

# 第 17 章 文件类型漏洞挖掘 与 Smart Fuzz

## 17.1 Smart Fuzz 概述

### 17.1.1 文件格式 Fuzz 的基本方法

不管是 IE 还是 Office，它们都有一个共同点，那就是用文件作为程序的主要输入。从本质上来说，这些软件都是按照事先约定好的数据结构对文件中不同的数据域进行解析，以决定用什么颜色、在什么位置显示这些数据。

不少程序员会存在这样的惯性思维，即假设他们所使用的文件是严格遵守软件规定的数据格式的。这个假设在普通的使用过程中似乎没有什么不妥——毕竟用 Word 生成的.doc 文件一般不存在什么非法的数据。

但是攻击者往往会挑战程序员的假定假设，尝试对软件所约定的数据格式进行稍许修改，观察软件在解析这种“畸形文件”时是否会发生错误，发生什么样的错误，以及堆栈是否会被溢出等。

文件格式 Fuzz（File Fuzz）就是这种利用“畸形文件”测试软件鲁棒性的方法。您可以在 Internet 上找到许多用于 File Fuzz 的工具。抛开界面、运行平台等因素不管，一个 File Fuzz 工具大体的工作流程包括以下几步，如图 17.1.1 所示。

- （1）以一个正常的文件模板为基础，按照一定规则产生一批畸形文件。
- （2）将畸形文件逐一送入软件进行解析，并监视软件是否会抛出异常。
- （3）记录软件产生的错误信息，如寄存器状态、栈状态等。
- （4）用日志或其他 UI 形式向测试人员展示异常信息，以进一步鉴定这些错误是否能被利用。

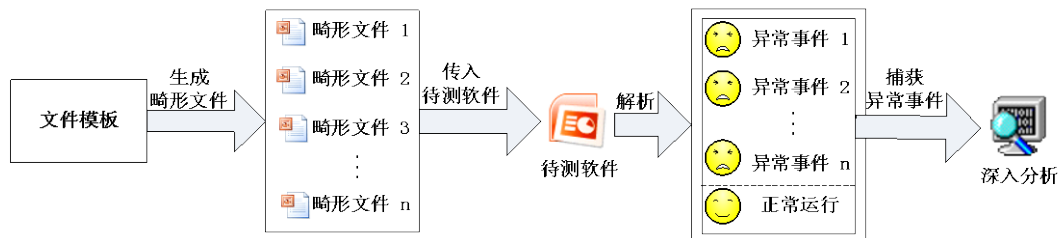


图 17.1.1 Fuzz 的一般步骤

## 17.1.2 Blind Fuzz 和 Smart Fuzz

Blind Fuzz 即通常所说的“盲测”，就是在随机位置插入随机的数据以生成畸形文件。然而现代软件往往使用非常复杂的私有数据结构，例如 PPT、word、excel、mp3、RMVB、PDF、Jpeg、ZIP 压缩包，加壳的 PE 文件。数据结构越复杂，解析逻辑越复杂，就越容易出现漏洞。复杂的数据结构通常具备以下特征：

- 拥有一批预定义的静态数据，如 magic，cmd id 等
- 数据结构的内容是可以动态改变的
- 数据结构之间是嵌套的
- 数据中存在多种数据关系（size of，point to，reference of，CRC）
- 有意义的数据被编码或压缩，甚至用另一种文件格式来存储，这些格式的文件被挖掘出越来越多的漏洞……

对于采用复杂数据结构的复杂文件进行漏洞挖掘，传统的 Blind Fuzz 暴露出一些不足之处，例如：产生测试用例的策略缺少针对性，生成大量无效测试用例，难以发现复杂解析器深层逻辑的漏洞等。

针对 Blind Fuzz 的不足，Smart Fuzz 被越来越多地提出和应用。通常 Smart Fuzz 包括三方面的特征：面向逻辑（Logic Oriented Fuzzing）、面向数据类型（Data Type Oriented Fuzzing）和基于样本（Sample Based Fuzzing）。

**面向逻辑：**测试前首先明确要测试的目标是解析文件的程序逻辑，而并不是文件本身。复杂的文件格式往往要经过多“层”解析，因此还需要明确测试用例正在试探的是哪一层的解析逻辑，即明确测试“深度”以及畸形数据的测试“粒度”。明确了测试的逻辑目标后，在生成畸形数据时可以具有针对性的仅仅改动样本文件的特定位置，尽量不破坏其他数据依赖关系，这样使得改动的数据能够传递到要测试的解析深度，而不会在上层的解析器中被破坏。

图 17.1.2 是对一个 PPT 文件进行面向逻辑测试和盲测的比较。可以看出，盲测中生成的大部分畸形文件都被无情地阻断在第一层解析器 OLE Parser 上，而面向逻辑测试生成的畸形文件则可以顺利通过 OLE Parser 到达下一层深度。

**面向数据类型测试：**测试中可以生成的数据通常包括以下几种类型。

- 算术型：包括以 HEX、ASCII、Unicode、Raw 格式存在的各种数值。
- 指针型：包括 Null 指针、合法/非法的内存指针等。
- 字符串型：包括超长字符串、缺少终止符(0x00)的字符串等。
- 特殊字符：包括#，@，‘，<，>，/，\，../ 等。

面向数据类型测试是指能够识别不同的数据类型，并且能够针对目标数据的类型按照不同规则来生成畸形数据。跟 Blind Fuzz 相比，这种方法产生的畸形数据通常都是有效的，能够大大减少无效的畸形文件。

**基于样本：**测试前首先构造一个合法的样本文件（也叫模板文件），这时样本文件里所有数据结构和逻辑必然都是合法的。然后以这个文件为模板，每次只改动一小部分数据和逻辑来



生成畸形文件，这种方法也叫做“变异”（Mutation）。对于复杂文件来说，以现成的样本文件为基础进行畸形数据变异来生成畸形文件的方法要比上面两种的难度要小很多，也更容易实现。但是这种方法不能测试样本文件里没有包含的数据结构，比如一个文件格式包含 18 种数据区块(Chunk)，而给定的样本文件中只用到了其中的 10 种，那么基于样本测试方法只会修改这 10 种区块的数据来产生畸形文件，测试不到其他 8 种数据对应的解析逻辑。为了提高测试质量，就要求在测试前构造一个能够包含几乎所有数据结构的文件（比如文字、图像、视频、声音、版权信息等数据）来作为样本。

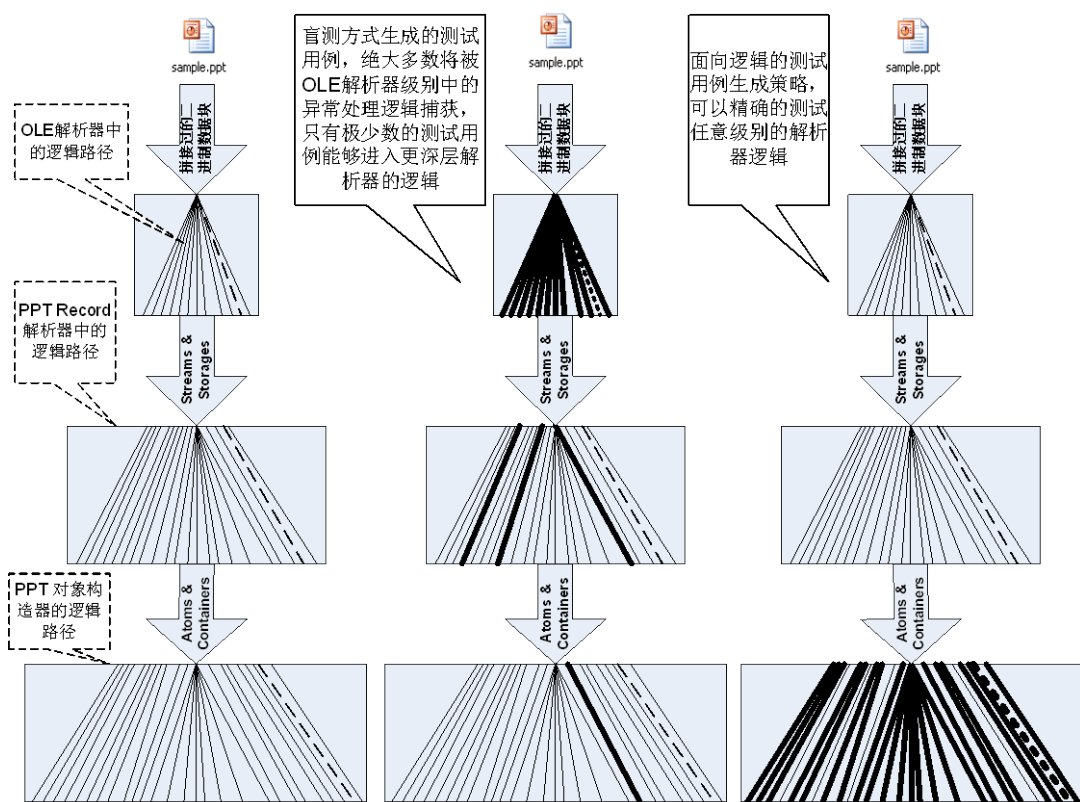
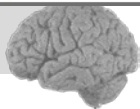


图 17.1.2 对 PPT 文件进行面向逻辑测试与盲测的比较

题外话：以上这三种 Fuzz 特性并不是相互独立的，而是可以同时使用的。通常，在一个好的 Smart Fuzz 工具中，这三种特性都会被包含。



## 17.2 用 Peach 挖掘文件漏洞

### 17.2.1 Peach 介绍及安装

Peach 是一款用 Python 写的开源的 Smart Fuzz 工具，它支持两种文件 Fuzz 方法：基于生长(Generation Based)和基于变异(Mutation Based)。基于生长的 Fuzz 方法产生随机或启发性数据来填充给定的数据模型，从而生成畸形文件。而基于变异的 Fuzz 方法在一个给定的样本文件基础上进行修改从而产生畸形文件。

Peach 的安装文件可以在 <http://peachfuzzer.com/PeachInstallation> 页面下载，有 exe 版本和 Python 源码两种版本。在 Windows 环境下安装方法如下。

#### exe 版本

- 安装 Debugging Tools for Windows，为了避免兼容问题，推荐使用 WinDbg 6.8.4 版本。最好再下载操作系统对应的 Windows 符号包（Windows Symbol Packet，<http://www.microsoft.com/whdc/devtools/debugging/symbolpkg.mspx>），安装后运行 WinDbg，按下 Ctrl+S 配置 Symbol Search Path，加入 Windows 符号包路径，最后保存退出。
- 下载并安装 Peach Installer，有 32 位(x86)和 64 位(x64)两种版本。
- 如果需要进行网络协议 Fuzz 的话，安装 Wireshark（<http://www.wireshark.org/>）或者 Winpcap（<http://www.winpcap.org/>）。

#### python 版本

- 安装 Debugging Tools for Windows，与安装 exe 版本中的第一步相同。
- 安装 Python2.5 或者 2.6。
- 下载 Peach 源码包至本地主机并解压，解压路径中最好不要包含中文字符和空格。
- 解压后在 dependencies/src 目录下有很多 Peach 运行时需要的包（packet），需要分别安装。安装方法是在命令行下分别进入各个包的目录，敲入命令“python setup.py install”。

安装完后，在命令行下进入 Peach 的安装目录，运行：

```
bin\peach.exe samples\HelloWorld.xml    (exe 版本)
python peach.py samples\HelloWorld.xml  (python 版本)
```

这是一个 HelloWorld 例程，如果可以正常运行，则表示程序已经正确安装。

题外话：由于 Python 是跨平台的，所以 Peach 当然也可以在其他操作系统下运行。如果要在其他操作系统环境下安装，请参照 <http://peachfuzzer.com/PeachInstallation>。



## 17.2.2 XML 介绍

Peach 使用 XML 语言来定义数据结构, 这种定义数据结构的文件被叫做 Peach Pit 文件。在学习写 Peach Pit 文件之前, 我们先来对 XML 进行一个简单的了解。

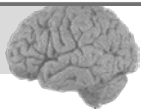
XML 是“Extensible Markup Language”的缩写, 即可扩展标记语言, 它与 HTML 一样都是标准通用标记语言 (Standard Generalized Markup Language, SGML)。XML 是 Internet 环境中跨平台的、依赖于内容的技术, 是当前处理结构化文档信息的有力工具。XML 是一种简单的数据存储语言, 虽然它比二进制数据要占用更多的空间, 但 XML 可读性很好, 也易于掌握和使用。

以下面这个 XML 文件为例:

```
<?xml version="1.0" encoding="GB2312"?>
<bookstore>
  <!-- This is a comment -->
  <book catalog="Programming">
    <title lang="cn">XML 入门</title>
    <author>Erik T.Ray</author>
    <price>42.00</price>
  </book>
  <book catalog="Networking">
    <title lang="cn">TCP/IP 详解</title>
    <author>W.Richard Stevens</author>
    <price>45.00</price>
  </book>
</bookstore>
```

XML 文件分为文件序言 (Prolog) 和文件主体两大部分。文件序言位于 XML 文件的第一行, 它告诉解析器该如何工作。文件序言中, version 标明此 XML 文件所用的标准版本号, 必须要有; encoding 标明此 XML 文件的编码类型, 如果是 Unicode 编码时则可以省略。文件主体分为如下几部分。

- 元素 (Element): 元素是组成 XML 文档的最小单位, 一个元素有一个标识来定义, 包括开始和结束标识以及其中的内容。上例中, <title lang="cn">XML 入门</title>就是一个元素。
- 标签 (Tag): 标签是用来定义元素的。在 XML 中, 标签必须成对出现, 将数据包围在中间。比如元素 <title lang="cn">XML 入门</title>中, <title>就是标签。另外, 在 XML 中可以在一个标签中同时表示起始和结束标签, 即在大于符号之前紧跟一个斜线 (/), 例如 XML 解析器会将 <tag/> 翻译成 <tag></tag>。
- 属性 (Attribute): 属性是对标签的进一步描述, 一个标签可以有多个属性。比如元素 <title lang="cn">XML 入门</title>中, lang="cn" 就是标签 <title> 的属性。
- 父元素 (Parent Element) 和子元素 (Child Element): 父元素是指包含其他元素的元素,



被包含的元素成为它的子元素。上个例子中，<book>是父元素，<title>、<author>、<price>是它的子元素。

- 根元素（Root Element）：又称文档元素，它是一个完全包含文档中其他所有元素的元素。根元素的起始标记要放在所有其他元素的起始标记之前，而根元素的结束标记要放在所有其他元素的结束标记之后。上个例子中，<bookstore>就是根元素。
- 注释（Comment）：在 XML 中，注释的方法与 HTML 完全相同，使用"<!--"和"-->"将注释文本括起来即可。

### 17.2.3 定义简单的 Peach Pit

Peach 所使用的 Peach Pit 文件包含了以下 5 个模块：

- GeneralConf
- DataModel
- StateModel
- Agents and Monitors
- Test and Run Configuration

下面分别介绍这 5 个模块的定义方法，并完成一个简单的 HelloWorld 程序。

题外话：在这之前，我们需要准备一个好用的 XML 文件编辑器，Visual Studio，Open XML Editor 或者 Notepad++ 都是不错的选择。这里我使用的是 Notepad++，它集成了数十种语言的语法着色方案，并且，它安装完后只有 10MB 左右。

首先，我们先搭好一个 XML 框架，下面要写的所有元素都要被包含在根元素<Peach>里。

```
<?xml version="1.0" encoding="utf-8"?>
<Peach xmlns=http://phed.org/2008/Peach xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
      xsi:schemaLocation=http://phed.org/2008/Peach ../peach.xsd >
  <!-- add elements here -->
</Peach>
```

其中，Peach 元素的各个属性基本是固定的，不要轻易改动。

#### （1）GeneralConf

GeneralConf 是 Peach Pit 文件的第一部分，用来定义基本配置信息。具体来说，包括以下三种元素。

- Include：要包含的其他 Peach Pit 文件。
- Import：要导入的 python 库。
- PythonPath：要添加的 python 库的路径。

要注意的是，所有的 Peach Pit 文件都要包含 default.xml 这个文件。

在 HelloWorld 中，GeneralConf 部分只需写入如下内容。





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

## (2) DataModel

DataModel 元素用来定义数据模型，包括数据结构和数据关系等。一个 Peach Pit 文件中需要包含一个或者多个数据模型。DataModel 可以定义的几种常用的数据类型如下。

- String: 字符串型。
- Number: 数据型。
- Blob: 无具体数据类型。
- Block: 用于对数据进行分组。

比如:

```
<DataModel name="HelloData">
  <String name="ID" size="32" value="RIFF" isStatic="true" />
  <Block name="TypeAndData">
    <Number name="Type" size="16" />
    <Blob name="Data" />
  </Block>
</DataModel>
```

要注意的是，size 的单位是 bit。上面的例子中，“ID”的“size”为 32，表示“ID”的长度为 4 字节（1 byte = 8 bits），刚好它的值“RIFF”也是 4 个字节。

在 HelloWorld 程序中，仅定义一个值为“Hello World!”的 String 类型数据。

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

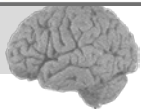
## (3) StateModel

StateModel 元素用于描述如何向目标程序发送 / 接收数据。StateModel 由至少一个 State 组成，并且用 initialState 指定第一个 State；每个 State 由至少一个 Action 组成，Action 用于定义 StateModel 中的各种动作，动作类型由 type 来指定。Action 支持的动作类型包括 start、stop、open、close、input、output、call 等。下面是一个例子：

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

<Action type="output">
  <DataModel ref="SomeDataModel" />
  <Data name="sample" filename="sample.bin"/>
</Action>

<Action type="call" method="DoStuff">
```



```
<Param name="param1" type="in">
  <DataModel ref="Param1DataModel" />
</Param>
</Action>

<Action type="close" />
```

上例中，第一个 Action 描述了一个输入型动作，表示按照数据模型 InputModel 产生数据并作为输入数据；第二个 Action 描述了一个输出型动作，表示按照数据模型 SomeDataMode 产生数据并输出到文件 sample.bin 中；第三个 Action 描述了一个调用动作，表示调用函数 DoStuff，并且将按照数据模型 Param1DataModel 产生的数据作为函数 DoStuff 的参数；第四个 Action 描述了一个关闭程序的动作。

当代码中存在多个 Action 时，则从上至下依次执行。

在 HelloWorld 程序中，我们只需要接收数据模型“HelloWorldTemplate”中的数据，所以写出如下的 StateModel。

```
<StateModel name="State" initialState="Statel" >
  <State name="Statel" >
    <Action type="output" >
      <DataModel ref="HelloWorldTemplate"/>
    </Action>
  </State>
</StateModel>
```

#### （4）Agent

Agent 元素用于定义代理和监视器，可以用来调用 WinDbg 等调试器来监控程序运行的错误信息等。一个 Peach Pit 文件可以定义多个 Agent，每个 Agent 下可以定义多个 Monitor。下面是一个例子：

```
<Agent name="LocalAgent" location="http://127.0.0.1:9000">
  <Monitor class="debugger.WindowsDebugEngine">
    <Param name="CommandLine" value="notepad.exe fileName" />
  </Monitor>
  <Monitor class="process.PageHeap">
    <Param name="Executable" value="notepad.exe" />
  </Monitor>
</Agent>
```

上例中，第一个 Monitor 类型为 debugger.WindowsDebugEngine，是调用 WinDbg 来执行下面的“notepad.exe filename”命令的。第二个 Monitor 类型为 process.PageHeap，意思是为 notepad.exe 开启页堆调试（Page Heap Debug），这在大多数 Windows Fuzzing 中都是很有用的。

在 HelloWorld 程序中，我们不需要启用 WinDbg 调试，所以无需配置 Agent。





### (5) Test and Run configuration

在 Peach Pit 文件中, Test and Run configuration 包括 Test 和 Run 两个元素。

Test 元素用来定义一个测试的配置, 包括一个 StateModel 和一个 Publisher, 以及 includeing/excluding、Agent 信息等。其中 StateModel 和 Publisher 是必须定义的, 其他是可选定义的。下面是一个 Test 配置的例子。

```
<Test name="TheTest">
  <Exclude xpath="//Reserved" />
  <Agent ref="LocalAgent" />
  <StateModel ref="TheState" />
  <Publisher class="file.FileWriter">
    <Param name="fileName" value="FuzzedFile"/>
  </Publisher>
</Test>
```

先对 Publisher 做一下介绍。Publisher 用来定义 Peach 的 IO 连接, 可以构造网络数据流 (如 TCP, UDP, HTTP) 和文件流 (如 FileWriter, FileReader) 等。上例中的 Publisher 定义表示将生成的畸形数据写到 FuzzedFile 文件中。

在 HelloWorld 程序中, 需要做的仅仅是把生成的畸形数据显示到命令行, 所以 Publisher 用的是标准输出 stdout.Stdout。

```
<Test name="HelloWorldTest">
  <StateModel ref="State"/>
  <Publisher class="stdout.Stdout" />
</Test>
```

现在到了最后一步, Run 的配置。Run 元素用来定义要运行哪些测试, 包含一个或多个 Test, 另外还可以通过 Logger 元素配置日志来捕获运行结果。当然, Logger 也是可选的。

```
<Run name="DefaultRun">
  <Test ref="TheTest" />
  <Logger class="logger.Filesystem">
    <Param name="path" value="c:\peach\logtest" />
  </Logger>
</Run>
```

上例表示程序运行 “TheTest” 这个测试, 并且把运行日志记录到 C:\peach\logtest 目录下。在 HelloWorld 程序中, 只需要在 Run 配置中放入之前定义好的 HelloWorldTest 就可以了。

```
<Run name="DefaultRun">
  <Test ref="HelloWorldTest" />
</Run>
```

将文件保存到 peach 目录下, 改名为 MyHelloWorld.xml, 然后运行:

```
bin\peach.exe MyHelloWorld.xml (exe 版本)
python peach.py MyHelloWorld.xml (python 版本)
```

如果没有错误的话,程序运行后会根据 DataModel 中定义的数据模型产生畸形数据,并将其显示到控制台。您将会看到命令行中不断显示出大量的乱码,大概几分钟后程序会运行完毕。如图 17.2.1 所示。

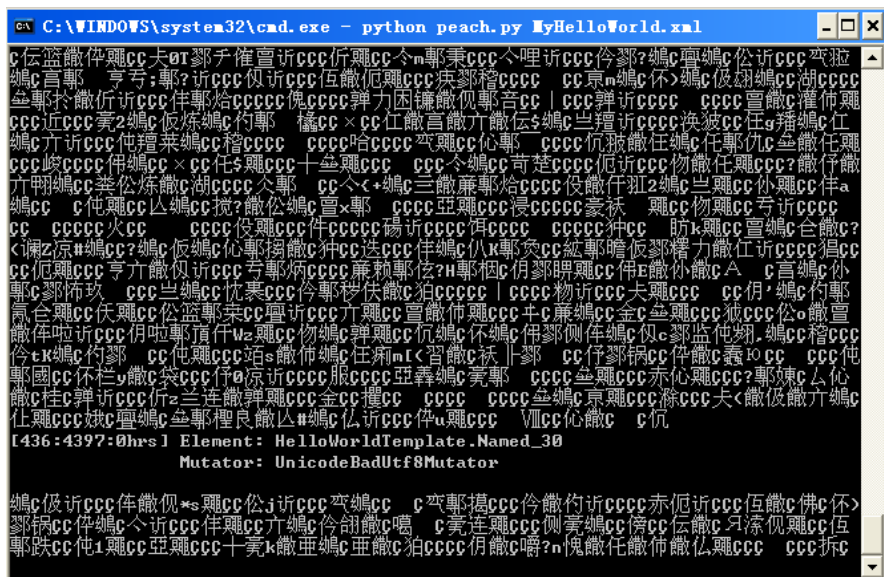


图 17.2.1 MyHelloWorld 例程运行结果

## 17.2.4 定义数据之间的依存关系

在 Smart Fuzz 中,并不是所有的数据都是可以随意产生的,比如数据校验值、数据长度等字段都是要进行计算才能得来的。如果让我们自己去计算这些数据的话,那将是一个费事、烦琐的工作。幸运的是,在 Peach Pit 文件中,可以用 Relation 元素来表示数据长度、数据个数以及数据偏移等信息。其格式为:

```
<Relation type="size" of="Data" />
<Relation type="count" of="Data" />
<Relation type="offset" of="Data" />
```

同样,数据校验值也可以通过 Fixup 元素来表示。Fixup 支持的校验类型包括 CRC32、MD5、SHA1、SHA256、EthernetChecksum、SspiAuthentication 等,具体细节可以参考 Peach 的官方文档。Fixup 的格式为:

```
<Fixup class="FixupClass">
  <Param name="ref" value="Data"/>
</Fixup>
```



其中 FixupClass 可以为 checksums.Crc32Fixup、checksums.SHA256Fixup 等。  
下面看一个例子，假定有如下的一个数据模型，如表 17-2-1 所示。

表 17-2-1 数据模型示例

Offset	Size	Description
0x00	4 bytes	Length of Data

续表

Offset	Size	Description
0x04	4 bytes	Type
0x08	Data	
after Data	4 bytes	CRC of Type and Data

可以看出，这里有两个数据需要定义依存关系。第一个是首 4 个字节的数据，其表示 Data 数据段的长度，可以用<Relation type="size" of="Data"/>这样的依存关系来表述。第二个是最后 4 个字节的数据，其表示 Type 和 Data 两个数据段的 CRC 校验，可以用<Fixup class="checksums.Crc32Fixup"/>这样的依存关系来表述。

考虑到需要将 Type 和 Data 这两个数据段合并到一起作为 Fixup 的参数，我们可以增加一个名为“TypeAndData”的 Block，将 Type 和 Data 放到该 Block 里，这样便可以用 TypeAndData 作为 Fixup 的参数。

DataModel 可以这样定义：

```
<DataModel name="HelloData">
  <Number name="Length" size="32">
    <Relation type="size" of="Data"/>
  </Number>
  <Block name="TypeAndData">
    <String name="Type" size="32"/>
    <Blob name="Data"/>
  </Block>
  <Number name="CRC" size="32">
    <Fixup class="checksums.Crc32Fixup">
      <Param name="ref" value="TypeAndData"/>
    </Fixup>
  </Number>
</DataModel>
```

## 17.2.5 用 Peach Fuzz PNG 文件

学习了 Peach Pit 文件之后，我们来进行一次简单的实战——使用 Peach 对 PNG 文件进行 Fuzz 测试。

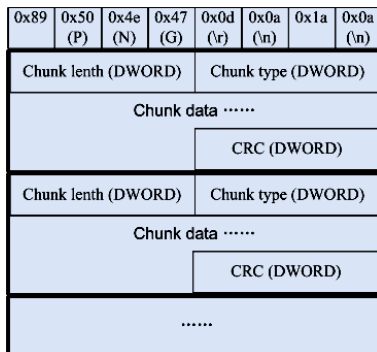


图 17.2.2 PNG 文件格式

首先来看一下 PNG 的文件格式，如图 17.2.2 所示。

一个 PNG 文件最前面是 8 个字节的 PNG 签名，十六进制值为 89 50 4E 47 0D 0A 1A 0A。随后是若干个数据区块(Chunk)，包括 IDHR、IDAT、IEND 等。每个区块的格式如表 17-2-2 所示。

表 17-2-2 PNG 文件 Chunk 格式

Name	Size	Description
Length	4 bytes	Length of data field
Type	4 bytes	Chunk type code
Data		Data bytes
CRC	4bytes	CRC of type and data

由此，可将 Chunk 的 DataModel 定义如下：

```
<DataModel name="Chunk">
  <Number name="Length" size="32" signed="false">
    <Relation type="size" of="Data" />
  </Number>
  <Block name="TypeAndData">
    <String name="Type" size="4"/>
    <Blob name="Data" />
  </Block>
  <Number name="CRC" size="32">
    <Fixup class="checksums.Crc32Fixup">
      <Param name="ref" value="TypeAndData" />
    </Fixup>
  </Number>
</DataModel>
```

首先，不去考虑每个 Chunk 的具体结构，而将 PNG 简单地认为是由一个 PNG 签名和若干结构相同的 Chunk 组成。那么可以在 Chunk 数据模型之后将 PNG 文件的 DataModel 进行如下定义：

```
<DataModel name="Png">
  <Blob name="pngMagic" isStatic="true" valueType="hex" value="89 50 4E 47 0D 0A 1A 0A" />
  <Block ref="Chunk" minOccurs="1" maxOccurs="1024" />
</DataModel>
```

minOccurs="1" maxOccurs="1024" 表示该区块最少重复 1 次，最多重复 1024 次。

然后开始配置 StateModel：第一步需要修改文件生成畸形文件；第二步需要把该文件关闭；第三步需要调用适当的程序打开生成的畸形文件。在这里，我们使用 pngcheck (<http://www.libpng.org/pub/png/apps/pngcheck.html>) 来作为打开畸形文件的程序。StateModel 定



义如下：

```
<StateModel name="TheState" initialState="Initial">
  <State name="Initial">
    <!-- Write out our png file -->
    <Action type="output">
      <DataModel ref="Png"/>
      <!-- This is our sample file to read in -->
      <Data name="data" fileName="sample.png"/>
    </Action>
    <!-- Close file -->
    <Action type="close"/>
    <!-- Launch the target process -->
    <Action type="call" method="D:\pngcheck.exe">
      <Param name="png file" type="in">
        <DataModel ref="Param"/>
        <Data name="filename">
          <!-- Name of Fuzzed output file -->
          <Field name="Value" value="peach.png"/>
        </Data>
      </Param>
    </Action>
  </State>
</StateModel>
```

在 call 动作中我们引用了一个叫做“Param”的数据模型，这个数据模型用来存放传递给 pngcheck.exe 的参数，即畸形文件的文件名。所以“Param”需要包含一个名为“Value”的字符型静态数据。我们需要在 StateModel 之前定义该数据模型。

```
<DataModel name="Param">
  <String name="Value" isStatic="true" />
</DataModel>
```

然后需要在 Test 元素中配置 Publisher 信息。在这里需要使用 FileWriterLauncher，它能够在写完文件之后使用 call 动作启动一个线程。它的参数应当是生成的畸形文件。

```
<Test name="TheTest">
  <StateModel ref="TheState"/>
  <Publisher class="file.FileWriterLauncher">
    <Param name="fileName" value="peach.png"/>
  </Publisher>
</Test>
```

最后在 Run 信息配置中指定要运行的测试名称。

```
<Run name="DefaultRun">
  <Test ref="TheTest" />
```

```
</Run>
```

至此，Peach Pit 文件就配置完毕了。将其命名为 `png_dumb.xml` 并和 `sample.png` 保存在 Peach 目录下。运行 Fuzzer：

```
bin\peach.exe png_dumb.xml （exe 版本）  
python peach.py png_dumb.xml （python 版本）
```

程序会根据 Peach Pit 文件的配置以及输入的样本 `sample.png` 生成畸形文件，并将该畸形文件传到 `pngcheck.exe` 中。如图 17.2.3 所示。

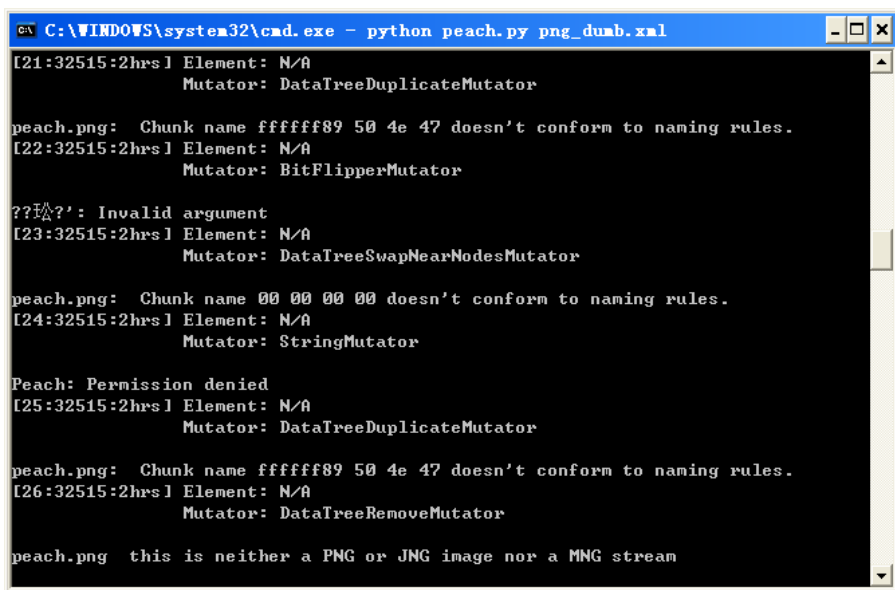


图 17.2.3 使用 `pngcheck.exe` 打开 PNG 测试用例

接下来对 `png_dumb.xml` 做一些改动，让程序调用 Windows 资源管理器打开畸形文件。

首先，在 StateModel 的 Action 中找到 `type="call"` 的这一行，并将后面的 `pngcheck` 地址改为 `explorer`。

```
<Action type="call" method="explorer">
```

然后在 Publisher 配置中将 `class` 改为 `file.FileWriterLauncherGui`，并为 Publisher 增加一个名为 `WindowName`、值为 `peach.png` 的参数。

```
<Publisher class="file.FileWriterLauncherGui">  
<Param name="fileName" value="peach.png" />  
<Param name="WindowName" value="peach.png" />  
</Publisher>
```

`FileWriterLauncherGui` 和 `FileWriterLauncher` 的区别在于，前者用于运行带界面的 GUI 程





序,并且在运行后会自动关闭窗口标题中含有 WindowName 的值的 GUI 窗口。

执行 Fuzzer,可以看到生成的各个 PNG 畸形文件逐一地被 explorer 打开,如图 17.2.4 所示。

为了捕获程序的异常,还需要配置一下 Agent and Monitor,调用 WinDbg 进行调试。在这之前,请确认已经安装了 WinDbg 6.8。



图 17.2.4 使用 explorer 打开 PNG 测试用例

题外话:到目前为止, Peach 与 WinDbg 6.12 是不兼容的,这会导致实验的失败。所以为了稳妥起见,建议您使用 WinDbg 6.8 版本来进行本实验。

首先,将 StateModel 中最后一个 Action 删掉,并添加这一行:

```
<Action type="call" method="ScoobySnacks" />
```

然后,在 StateModel 下面加入 Agent 配置:

```
<Agent name="LocalAgent">
  <Monitor class="debugger.WindowsDebugEngine">
    <Param name="CommandLine" value="explorer peach.png" />
    <Param name="StartOnCall" value="ScoobySnacks" />
  </Monitor>

  <Monitor class="process.PageHeap">
    <Param name="Executable" value="explorer"/>
  </Monitor>
</Agent>
```

然后,在 Test 配置的第一行加入:

```
<Agent ref="LocalAgent"/>
```

并且在 Publisher 的最后一行加入名为 debugger, 值为 true 的参数:

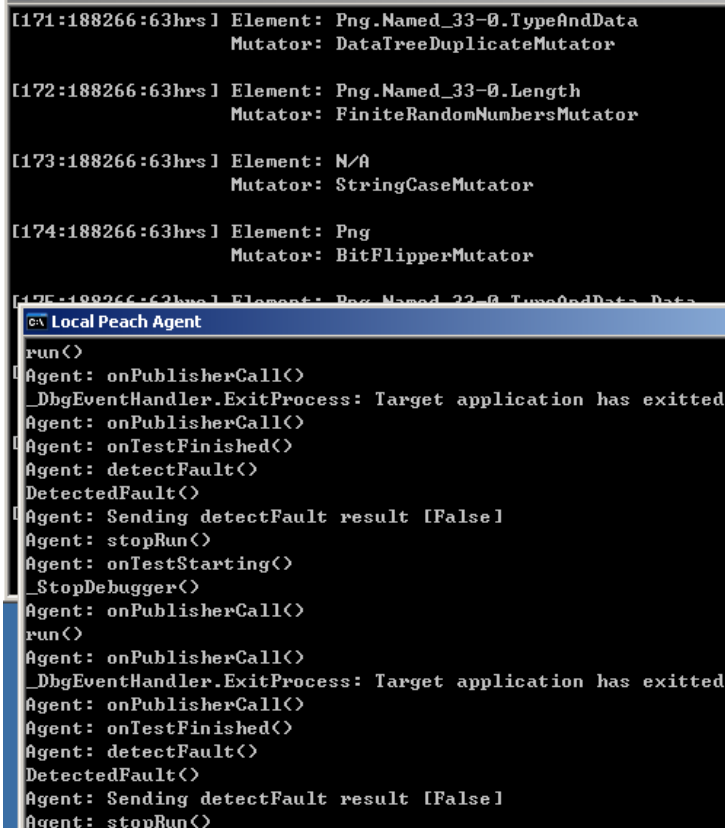
```
<Param name="debugger" value="true" />
```

最后在 Run 配置的 Test 元素后面加入日志配置:

```
<Logger class="logger.Filesystem">
```

```
<Param name="path" value="logs"/>
</Logger>
```

重新运行 Fuzzer, 如图 17.2.5 所示, Fuzzer 程序启动了一个 Local Peach Agent, 通过该 Agent 控制 WinDbg 进行调试并捕获异常事件。



```
[171:188266:63hrs] Element: Png.Named_33-0.TypeAndData
Mutator: DataTreeDuplicateMutator

[172:188266:63hrs] Element: Png.Named_33-0.Length
Mutator: FiniteRandomNumbersMutator

[173:188266:63hrs] Element: N/A
Mutator: StringCaseMutator

[174:188266:63hrs] Element: Png
Mutator: BitFlipperMutator

[175:188266:63hrs] Element: Png.Named_33-0.TypeAndData_Data

Local Peach Agent
run()
Agent: onPublisherCall()
_DbgEventHandler.ExitProcess: Target application has exited
Agent: onPublisherCall()
Agent: onTestFinished()
Agent: detectFault()
DetectedFault()
Agent: Sending detectFault result [False]
Agent: stopRun()
Agent: onTestStarting()
_StopDebugger()
Agent: onPublisherCall()
run()
Agent: onPublisherCall()
_DbgEventHandler.ExitProcess: Target application has exited
Agent: onPublisherCall()
Agent: onTestFinished()
Agent: detectFault()
DetectedFault()
Agent: Sending detectFault result [False]
Agent: stopRun()
```

图 17.2.5 开启 WinDbg 调试的 Fuzzing

## 17.3 010 脚本，复杂文件解析的瑞士军刀

### 17.3.1 010 Editor 简介

010 Editor 是一款非常强大的文本/十六进制编辑器，除了文本/十六进制编辑外，还包括文件解析、计算器、文件比较等功能，但它真正的强大之处还在于文件的解析功能。我们可以使用 010Editor 官方网站提供的解析脚本(Binary Template)对 avi、bmp、png、exe 等简单格式的文件进行解析，当然也可以根据需求来自己编写文件解析脚本。

下面以 PNG 文件解析为例，介绍 010 Editor 的文件解析功能。首先从官方网站上下载和安装 010 Editor (<http://sweetscape.com/010editor>)，然后到文件解析脚本下载页面中下载



PNGTemplate.bt。用 010 Editor 打开 PNG 文件，然后通过 Templates -> Open Template 菜单打开 PNGTemplate.bt，按 F5 键运行该脚本，就可以在 Template Results 窗口中看到该 PNG 文件的解析结果。如图 17.3.1 所示。

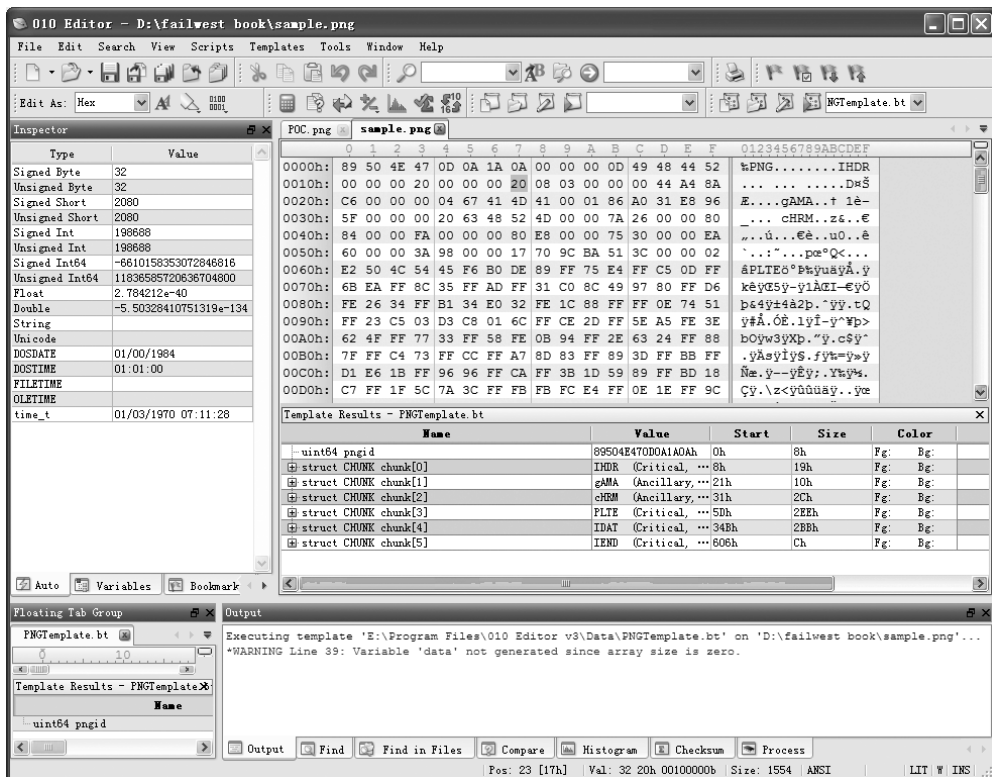


图 17.3.1 010 Editor 的文件解析功能

## 17.3.2 010 脚本编写入门

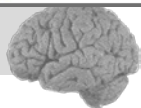
学过 C/C++ 的您会发现 010 Editor 的文件解析脚本（即 010 脚本）看起来跟 C/C++ 的结构体定义比较相似。然而文件解析脚本不是结构体，而是一个自上而下执行的程序，所以它可以使用 if、for、while 等语句。

在 010 脚本中，声明的每个变量都对应着文件的相应字节。比如以下声明：

```
char header[4];
int numRecords;
```

这意味着，文件的首 4 个字节将会映射到字符数组 header 中，下 4 个字节则会映射到整型变量 numRecords 中，并最终显示在解析结果中。

然而，在编写 010 脚本时可能会遇到这种情况：需要定义一些变量，但是这些变量并不对应着文件中的任何字节，而仅仅是程序运行中所需要的，这时可以使用 local 关键字来定义变



量。比如以下声明：

```
local int i, total = 0;
int recordCounts[5];
for(i=0; i < 5; i++)
    total += recordCounts[i];
double records[total];
```

这样，i 和 total 就不会映射到文件中，也不会解析结果中显示出来。

另外，在数据的定义中，可以加上一些附加属性，如格式、颜色、注释等。附加属性用尖括号<>括起来。常用的属性包括以下几种：

```
<format=hex|decimal|octal|binary, fgcolor=<color>, bgcolor=<color>, comment=
"<string>", open=true|false|suppress, hidden=true|false,
read=<function_name>, write=<function_name> >
```

下面给出一个简单的实例。假设有一种文件格式如图 17.3.2 所示，我们可以看出，它由一个 Header 和若干个 Record 数据块组成。在 Header 中，numRecords 表示 Record 的个数，而在 Record 中，根据 Header 中 version 值的不同，data 的类型也不同。

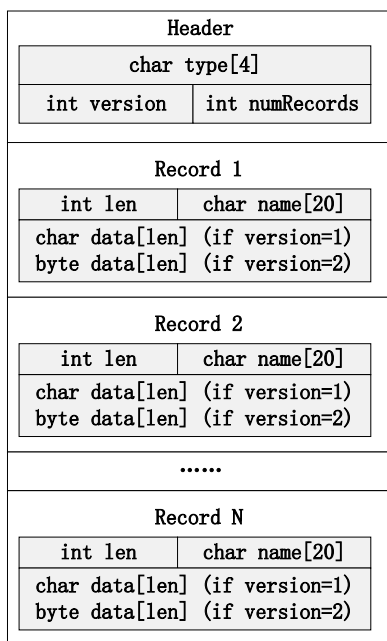


图 17.3.2 文件格式示例

根据文件格式，我们可以写出如下脚本：

```
struct FILE {
    struct HEADER {
        char    type[4];
```



```
        int    version;
        int    numRecords;
    } header;

    struct RECORD {
        int    len;
        char    name[20];
        if( file.header.version == 1 )
            char data[len];
        if( file.header.version == 2 )
            byte data[len];
    } record[ file.header.numRecords ];
} file;
```

### 17.3.3 010 脚本编写提高——PNG 文件解析

本节中我们将创建一个解析 PNG 文件的 010 脚本。首先来回顾一下图 17.2.2 中介绍过的 PNG 文件格式。由图 17.2.2 中可知，需要定义 PNG 签名和 Chunk 两种结构。先来定义 PNG 签名：

```
const uint64 PNGMAGIC = 0x89504E470D0A1A0AL;
```

接下来，参照 17.2.5 节介绍的 PNG 的 Chunk 格式，写下 Chunk 的结构定义：

```
typedef struct {
    uint32 length;
    char    ctype[4];
    ubyte    data[length];
    uint32 crc <format=hex>;
} CHUNK
```

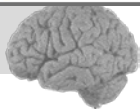
其中<format=hex>是 crc 的附加属性，表明该数据用十六进制来表示。

我们还需要定义 CHUNK 结构体的 read 函数，以便在显示解析结果时能够给出每个 Chunk 的名字，显然 ctype 的值可以作为 Chunk 的名字。此外，在 ctype 中，每个字节的第三位还分别标识了该 Chunk 的一些附加信息，如表 17-3-1 所示。

表 17-3-1 ctype 数据表述的附加信息

位 置	1	0
第 1 字节的第 3 位	Ancillary	Critical
第 2 字节的第 3 位	Private	Public
第 3 字节的第 3 位	ERROR_RESERVED	
第 4 字节的第 3 位	Safe to Copy	Unsafe to Copy

- Ancillary 表示该区块是辅助区块，这些区块是可有可无的；Critical 表示该区块是关键区块，这些区块是必需的。



- Private 表示该区块是不在 PNG 标准规格 (PNG specification) 区块之中的, 属于该 PNG 文件私有, 其名称的第二个字母是小写的; Public 表示该区块属于 PNG 标准规格区块, 其名称的第二个字母是大写的。
- Safe to Copy 表示该区块与图像数据无关, 可以随意复制到改动过的 PNG 文件中; Unsafe to Copy 表示该区块内容与图像数据息息相关, 如果对文件的 Critical 区块进行了增删改等操作, 则该区块也需要进行相应的修改。

把这些信息也加到 Chunk 的显示结果中去。在 CHUNK 结构定义下面写出如下函数:

```
string readCHUNK(local CHUNK &c) {  
    local string s;  
    s=c.ctype+" (";  
    s += (c.ctype[0] & 0x20) ? "Ancillary, "      : "Critical, ";  
    s += (c.ctype[1] & 0x20) ? "Private, "        : "Public, ";  
    s += (c.ctype[2] & 0x20) ? "ERROR_RESERVED, " : "";  
    s += (c.ctype[3] & 0x20) ? "Safe to Copy)"    : "Unsafe to Copy)";  
    return s;  
}
```

题外话: 将一个数与 0x20 (二进制为 0010 0000) 进行“按位与”运算即为取该数的第 3 位。如果您平时留心的话, 会发现在许多程序中都会用到这种按位运算的小技巧。

然后在上面定义好的 CHUNK 结构体后加上 read 属性, 即把 “}CHUNK;” 这一行改为:

```
}CHUNK <read=readCHUNK>;
```

最后写入解析“主函数”:

```
// -----  
// Here's where we really allocate the data  
// -----  
uint64 pngid <format=hex>;  
if (pngid != PNGMAGIC) {  
    Warning("Invalid PNG File: Bad Magic Number"); return -1;  
}  
while(!FEOF()) {  
    CHUNK chunk;  
}
```

另外, 由于 PNG 文件是按照 BigEndian 格式进行存储的, 所以我们要在脚本的第一行加入:

```
BigEndian();
```

至此终于大功告成。将编写好的 010 脚本进行保存, 打开一个 PNG 文件, 然后运行该脚本。可以看到类似于图 17.3.3 的解析结果。





从图 17.3.3 中可以看到, 该 PNG 文件包含了 6 个 Chunk: IHDR、gAMA、cHRM、PLTE、IDAT 和 IEND。当然, 根据 PNG 文件的不同, 解析结果也不尽相同。但是在任何 PNG 文件中, IHDR、PLET、IDAT 和 IEND 这 4 个 Chunk 是必不可少的。

下面我们来做一个有趣的实验。首先介绍一下实验环境, 如表 17-3-2 所示。

uint64 pngid	89504E470D0A1A0Ah	0h	8h
struct CHUNK chunk[0]	IHDR (Critical, Public, Unsafe to Copy)	8h	19h
uint32 length	13	8h	4h
char ctype[4]	IHDR	Ch	4h
char ctype[0]	73 'I'	Ch	1h
char ctype[1]	72 'H'	Dh	1h
char ctype[2]	68 'D'	Eh	1h
char ctype[3]	82 'R'	Fh	1h
ubyte data[13]		10h	Dh
uint32 crc	44A48AC6h	1Dh	4h
struct CHUNK chunk[1]	gAMA (Ancillary, Public, Unsafe to Copy)	21h	10h
struct CHUNK chunk[2]	cHRM (Ancillary, Public, Unsafe to Copy)	31h	2Ch
struct CHUNK chunk[3]	PLTE (Critical, Public, Unsafe to Copy)	5Dh	2EEh
struct CHUNK chunk[4]	IDAT (Critical, Public, Unsafe to Copy)	34Bh	2BBh
struct CHUNK chunk[5]	IEND (Critical, Public, Unsafe to Copy)	606h	Ch

图 17.3.3 PNG 文件解析结果

表 17-3-2 实验环境

计算机	VMware 虚拟机或者实体机
操作系统	Windows XP Professional Server Pack 3
GdiPlus.dll 版本	5.1.3102.5512

题外话: GdiPlus.dll 是 GDI 图像设备接口图形界面的相关模块, 属于 Microsoft GDI+, 它会在安装一些软件或补丁时出现, 并且出现在与所安装软件相同分区下。所以 gdiplus.dll 不一定在 C:\WINDOWS\system32 目录里, 在实验前您可能有必要搜索一下它的位置。

在之前的 PNG 解析结果中找到 IHDR Chunk 的 length 位置, 也就是第 9~12 字节, 通常情况下其值应该是 13(0x0D)。现在我们把它改为 0xFFFFFFFF4, 并将文件另存为 poc.png。如图 17.3.4 所示。

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	89	50	4E	47	0D	0A	1A	0A	FF	FF	FF	F4	49	48	44	52	%PNG....YyYyIHDR
0010h:	00	00	00	20	00	00	00	20	08	03	00	00	00	44	A4	8A	... ..[D=Š
0020h:	C6	00	00	00	04	67	41	4D	41	00	01	86	A0	31	E8	96	E]...gAMA...† 1è-
0030h:	5F	00	00	00	20	63	48	52	4D	00	00	7A	26	00	00	80	_... cHRM...zš...€
Template Results - MyPNGTemplate.bt																	
Name		Value												Start	Size		
uint64 pngid		89504E470D0A1A0Ah												0h	8h		
struct CHUNK chunk[0]		IHDR (Critical, Public, Unsafe to Copy)												8h	19h		
uint32 length		4294967284												8h	4h		
char ctype[4]		IHDR												Ch	4h		
ubyte data[13]														10h	Dh		
uint32 crc		44A48AC6h												1Dh	4h		

图 17.3.4 将 IHDR 的 length 修改为 0xFFFFFFFF4

在实验计算机中打开 poc.png, 或者仅仅是打开 poc.png 所在的文件夹, explorer.exe 的 CPU

占用率就上升到了将近 100%，使系统接近当机状态。只有强行结束或重启 explorer.exe 进程才能使系统恢复正常。如图 17.3.5 所示。

实际上，这是一个 gdiplus.dll 在处理 IHDR 时的整数溢出漏洞。该漏洞有很多种危害方式：

- 使得打开 poc.png 文件或者打开 poc.png 所在文件夹的未打补丁的用户死机
- 将 poc.png 挂到某个网页面上，将使得访问该页面的未打补丁的用户死机
- 将 poc.png 设为 QQ 或 MSN 头像，将使看到您头像的未打补丁的好友死机
- .....

由此可见，在解析文件的基础上进行漏洞的挖掘和测试比盲目地 Fuzz 要方便、有效很多。

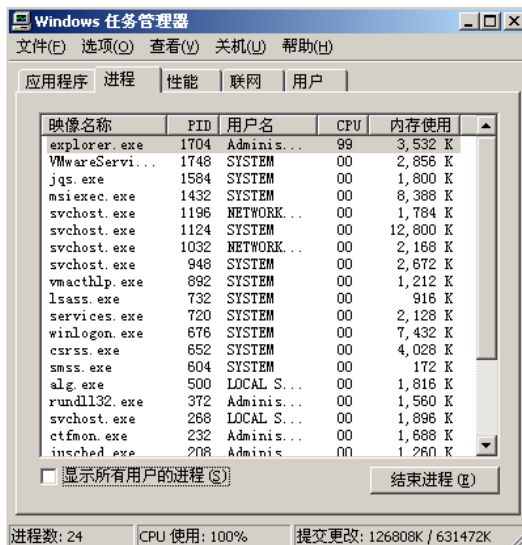


图 17.3.5 CPU 占用率 100%

### 17.3.4 深入解析，深入挖掘——PPT 文件解析

这一节我们将挑战一种更加复杂的文件类型解析。通过学习本节手工解析 PPT 的知识，您完全可以自动化这种手工挖掘的过程，并据此写出自己的 Smart Fuzz 工具。

Office 系列软件使用的文件格式可以分为两个系列。

**Office 97~Office 2003:** 使用基于二进制的文件格式，文件名后缀为 doc、ppt、xls 等。

**Office 2003 及更高版本:** 使用基于 XML 的文件格式，文件名后缀为 docx、pptx、xlsx 等。

本节主要讨论 PowerPoint 97~2003 所使用的二进制文件格式。如图 17.1.2 所示，PPT 文件的解析过程从逻辑上可以分为如表 17-3-3 所示的四层。

表 17-3-3 PPT 文件解析器逻辑分层

测试深度	解析逻辑	数据粒度	Fuzz 方法
Level1	OLE2 解析器	离散分布的 512 字节数据段	修改 OLE 文件头、FAT 区块、目录区块



			等位置的数据结构
Level2	PPT 记录解析器	流和信息库	修改流中的数据, 破坏记录头和数据的 关系
Level3	PPT 对象创建器	原子和容器	用负载替换原子数据

续表

测试深度	解析逻辑	数据粒度	Fuzz 方法
Level4	PPT 对象内部逻辑	原子记录内部的 integer、 bool、string 等类型数据	用相关的负载集修改字节数据

PPT 的有效数据被组织在基于 OLE2 的二进制文件中。OLE2 文件把数据组织成流(Stream)进行存储。数据流在逻辑上连续存储, 在硬盘上则离散存储, 如图 17.3.6 所示。

如果直接用十六进制编辑器打开一个 OLE2 文件的话, 将会看到许多大小为 512 字节的离散的数据块, 并且在文件的开头有一个存储了这些数据块索引的 FAT 表。如果熟悉 FAT32 文件系统的话, 您会发现 OLE2 的 FAT 表跟 FAT32 文件系统的索引是很相似的。

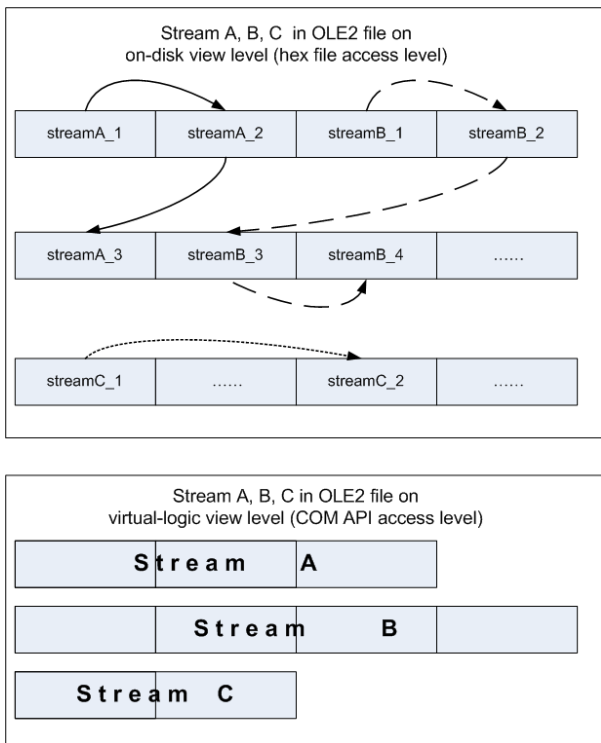


图 17.3.6 OLE2 文件格式

在本书的附带资料中, 我们给出了一个解析 OLE2 文件的 010 脚本 OLE2.bt。用 010 Editor 载入 PPT 文件, 运行 OLE2.bt, 可得到类似图 17.3.7 的解析结果。推荐您参照 OLE2 文件格式的详细介绍“Windows Compound Binary File Format Specification”来理解该脚本。

可以看到, 我们已经把看起来“杂乱无章”的二进制文件解析成了可读性较强的 OLE2 结构体, 可以比较容易地读出文件中的各个数据流。接下来我们要将 OLE2 进一步解析为 PPT。

PPT 的有效数据存储在 OLE2 的 stream 中, 通常一个 PPT 文件包括以下几种 stream, 如表 17-3-4 所示。

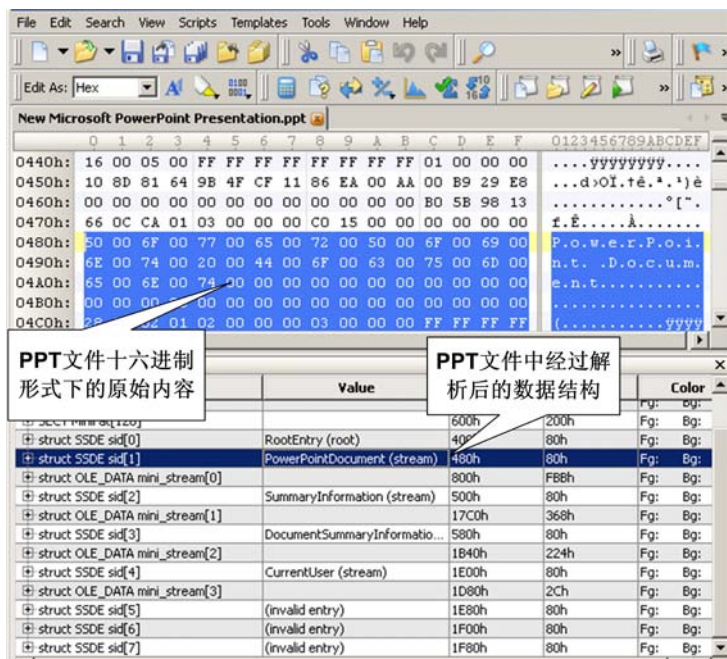


图 17.3.7 OLE2 解析结果

表 17-3-4 PPT 中常见数据流

数据流名称	说 明
Current User	包含最近打开演示文档的用户信息
PowerPoint Document	包含演示文档中的数据信息
Pictures(可选)	包含演示文档中的图像信息
Summary Information	包含演示文档的若干统计信息
Document Summary Information(可选)	包含演示文档的若干统计信息

在 PowerPoint Document 数据流中, 包含了我们感兴趣的大部分信息, 例如 font、color、text、position 等数据结构。所以下面着重讨论 PowerPoint Document 数据流的解析。

在 PowerPoint Document 数据流中, 数据以若干个“记录 (Record)”的形式存储。每个记录都包含一个 8 字节的 header, 根据 header 的不同, 记录又可分为“容器 (Container)”和“原子 (Atom)”两种类型。其中, 容器可以包含原子和其他容器, 而原子则包含了 PowerPoint 对象的真正数据。如图 17.3.8 所示。

微软已经公布了 PPT 文件中所有记录 (record) 的格式说明, 这就不难从 OLE2.bt 解析出

的数据流中进一步解析出所有 PPT 的原子和容器结构。如果您亲自动手编程，不难发现这实际上是一个树的遍历问题。限于篇幅，这里不再对解析过程做详细说明。在本书的随书资料中我们给出了一个简单的 O10 解析脚本 `ppt_parse.bt`。执行 `ppt_parse.bt` 脚本，即可得到 PPT 的解析结果，如图 17.3.9 所示。

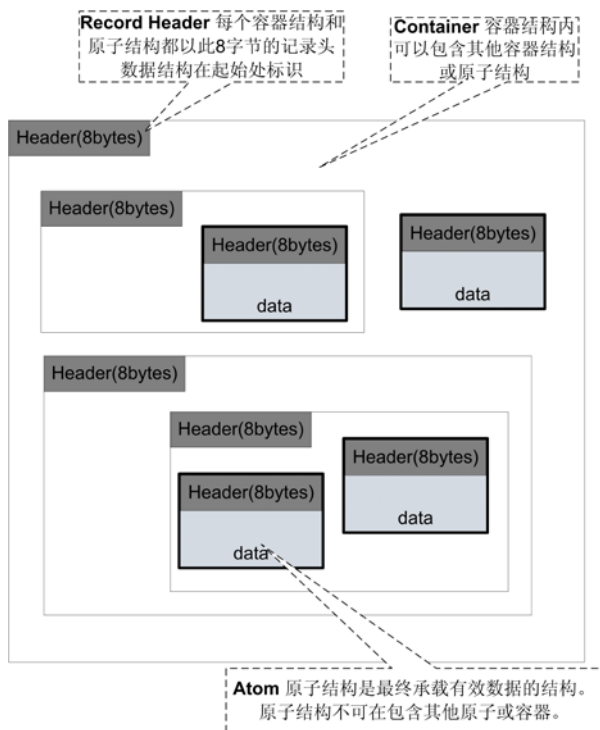
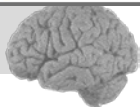


图 17.3.8 PPT 中的 Record 和 Atom



```

New Microsoft PowerPoint Presentation.ppt
0F 00 02 E8 03 01 04 00 00 01 00 E9 03 28 00 00 00
0610h: 80 16 00 00 00 E0 10 00 00 E0 10 00 00 80 16 00 00
0620h: 05 00 00 00 0A 00 00 00 00 00 00 00 00 00 00 00
0630h: 01 00 00 00 00 00 00 00 01 0F 00 F2 03 60 01 00 00
0640h: 2F 00 C8 0F 0C 00 00 00 30 00 D2 0F 04 00 00 00
0650h: 00 00 00 00 0F 00 D5 07 98 00 00 00 00 00 B7 0F
0660h: 44 00 00 00 41 00 72 00 69 00 61 00 6C 00 00 00
0670h: BC 96 12 00 5F 3E 0B 3C BC 96 12 00 00 00 00 00
0680h: 30 00 D2 0F 54 A9 12 00 54 A9 12 00 F4 74 B0 00
0690h: BC 96 12 00 78 3A 0B 3C BC 96 12 00 00 00 00 00
06A0h: 0F 00 D5 07 00 00 04 00 10 00 B7 0F 44 00 00 00
06B0h: 8B 5B 53 4F 00 00 61 00 6C 00 00 00 BC 96 12 00

Q123456789ABCDEF
..e...e...
€.a...a...e...
.....
.....0...
./ë...0.0...
.....ö...
D...A...a...
>...>...
0.0.T0...T0...0t°
U...x...0U...
...ö...D...
<[SO...a...l...<...

Template Results - ppt_parse.bt

```

图 17.3.9 PPT 解析结果

下面我们再来做一个实验。在解析结果中找到 PSR\_FontEntityAtom FontEntityAtom 的一条记录(如图 17.3.9 中选中的部分),将其展开,可以看到 FontEntityAtom 的数据结构如表 17-3-5 所示。

表 17-3-5 FontEntityAtom 数据结构

位 置	类 型	名 称
0	uint2[32]	IfFaceName
64	ubyte	IfCharSet
65	ubyte	IfClipPrecision
66	ubyte	IfQuality
67	ubyte	IfPitchAndFamily

其中, `IfFaceName` 的内容是一个以 `NULL (0x0000)` 结尾的字符串, 它指定的所用的字体名。该字符串的长度不得超过 32 个字符。我们将 `IfFaceName` 的内容改成没有结束符 `NULL` 的长字符串, 比如 32 个 `0x1111`, 另存为 `test.ppt`。然后用 `PowerPoint` 打开 `test.ppt`, 选中与更改过的 `FontEntityAtom` 相关的文字, 可以看到该文字的字体名称变成了一堆乱码, 这便是畸形数据 (`Malformed Data`), 如图 17.3.10 所示。



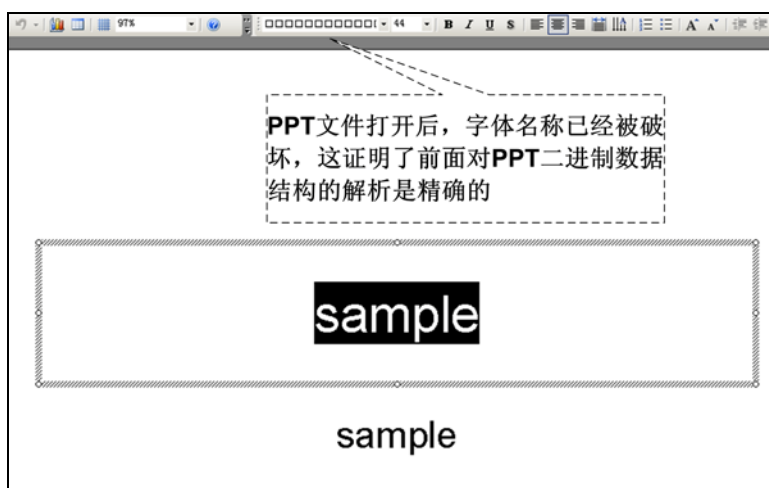


图 17.3.10 PowerPoint 打开 test.ppt 时脏数据的出现

通过这种方法，可以在 PPT 文件中构造出具有高度针对性的畸形数据，而避免 OLE2 层初级解析器的异常，将畸形数据直接送达 PowerPoint 解析器的最深层。这种深入解析，深入挖掘的思想已经被越来越多的 Smart Fuzz 测试系统所采用。