

|  |
| --- |
| Lab Guide |
| Custom Pits Training |
| Peach Fuzzer |



# Peach Custom Pits Training Lab Guide

This document will guide you through each of the labs in the Custom Pits Training course. The first lab will start with very detailed instructions that will guide you through the changes to be made. As the labs progress the instructions will become less detailed in areas you are already familiar with.

**Lab Table of Contents**

[Lab: File Fuzzing 3](#_Toc449815663)

[Lab: Network Server Fuzzing 19](#_Toc449815664)

[Lab: REST API Fuzzing 22](#_Toc449815665)

[Lab: Network Client Fuzzing 23](#_Toc449815666)

## Lab: File Fuzzing

In this lab, we will progressively build a PNG file fuzzer that will target first a command line application and later an application with a GUI. This lab will be completed in five stages. Each stage will introduce Peach features and complexity to the fuzzer until we end up with a basic smart fuzzer for PNG.

### Part A – Simple Dumb PNG Fuzzer

First, we are going to create a simple dumb PNG fuzzer. This fuzzer will place the contents of the sample PNG file into a Blob data element, mutate it and then launch a command line PNG tool.

The tasks for this section are:

1. Modify the png\_dumb.xml template to load a sample PNG file, sample.png, into a DataModel that contains only a Blob element.
2. Write the mutated file out to a file called peach.png.
3. Launch a command line program bin\pngcheck.exe, which accepts a single parameter that indicates the PNG to validate - use peach.png.

The following are detailed steps for completing your tasks.

**Start the virtual machine**

If the virtual machine is not already running please start it now using the VMWare Player application.

**Open up the template**

1. Open folder c:\labs\png
2. Double click on png\_dumb.xml to load into editor

**Modify the** DataModel

1. Locate and rename the DataModel element to “Png”.

<DataModel name="**Png**">

</DataModel>

1. Add a Blob data element to the “Png” DataModel.

<DataModel name="Png">

**<Blob />**

</DataModel>

1. Add an Action to the initial State that will write out the PNG data and also loads our sample.png file.

<StateModel name="TheState" initialState="Initial">

<State name="Initial">

**<!-- Write the PNG file -->**

**<Action type="output">**

**<DataModel ref="Png"/>**

**<!-- Load sample data into our data model -->**

**<Data fileName="sample.png" />**

**</Action>**

</State>

</StateModel>

1. Add an Action to the StateModel that will cause the file we are writing to be closed.

<StateModel name="TheState" initialState="Initial">

<State name="Initial">

<!-- Write the PNG file -->

<Action type="output">

<DataModel ref="Png"/>

<!-- Load sample data into our data model -->

<Data fileName="sample.png" />

</Action>

**<!-- Close the file we are writing -->**

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

</State>

</StateModel>

1. Modify the StateModel to include a call-based Action this will cause the program to run.

<!-- Our Simple State model -->

<StateModel name="TheState" initialState="Initial">

<State name="Initial">

<!-- Write the PNG file -->

<Action type="output">

<DataModel ref="Png"/>

<Data fileName="c:\labs\png\sample.png" />

</Action>

<!-- Close the file we are writing -->

<Action type="close" />

**<!-- Launch the test application -->**

**<Action type="call" method="ScoobySnacks" publisher="Peach.Agent" />**

</State>

</StateModel>

1. Modify the Publisher to be of type File.

<Test name="Default">

<Agent ref="LocalAgent"/>

<StateModel ref="TheState"/>

**<Publisher class="File">**

**<Param name="FileName" value="peach.png"/>**

**</Publisher>**

</Test>

1. Modify the Agent, add a new Monitor of type WindowsDebugger.

<Agent name="LocalAgent">

**<Monitor class="WindowsDebugger">**

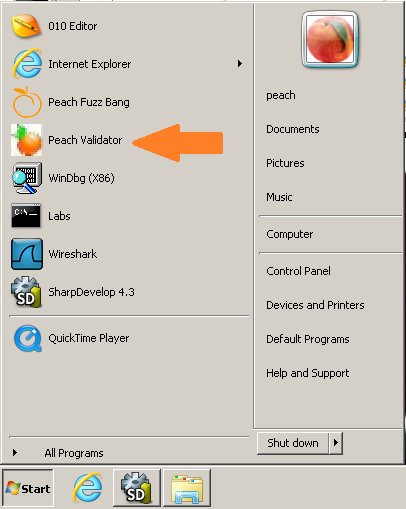
**<Param name="Executable" value="bin\pngcheck.exe peach.png"/>**

**<Param name="StartOnCall" value="ScoobySnacks"/>**

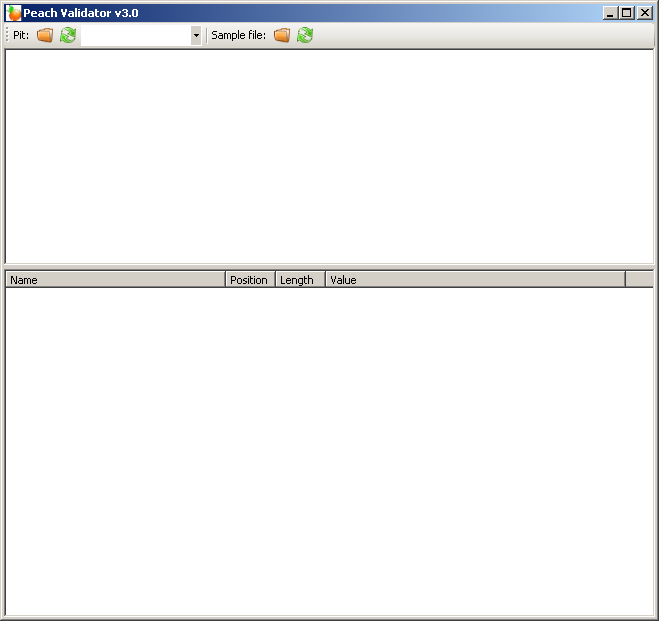
**</Monitor>**

</Agent>

1. Validate your modifications by using the Peach Validator, which can be found in Start Menu of your VM.



* 1. Click the Open Pit button, browse and load the png\_dumb.xml file located in c:\labs\png
  2. Select Png from the drop down box
  3. Browse and select the c:\labs\png\sample.png file
  4. Click the refresh button



**2**

**4**

**3**

**1**

Figure 1 - Peach Validator

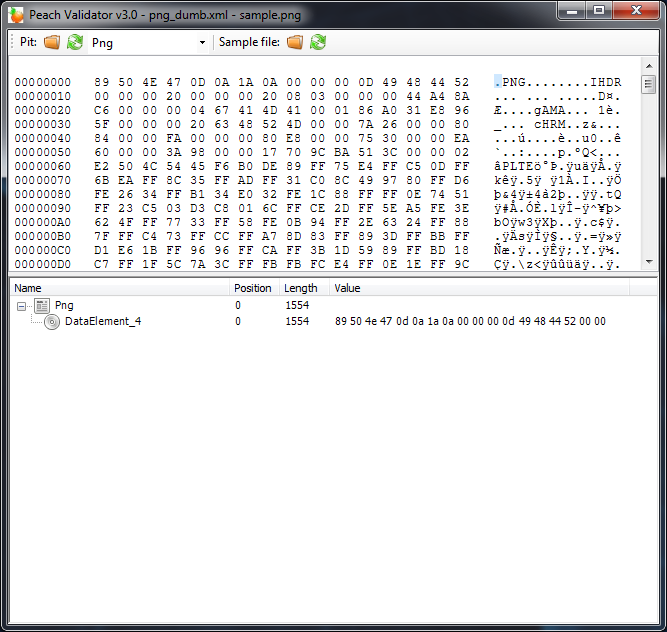


Figure 2 - Data loaded into Peach Validator

1. If everything validates, execute the fuzzer by double clicking on **Command Prompt** in the c:\labs\png folder. To accomplish this, type c:\peach\peach png\_dumb.xml in the command window as shown in **Figure 3**.

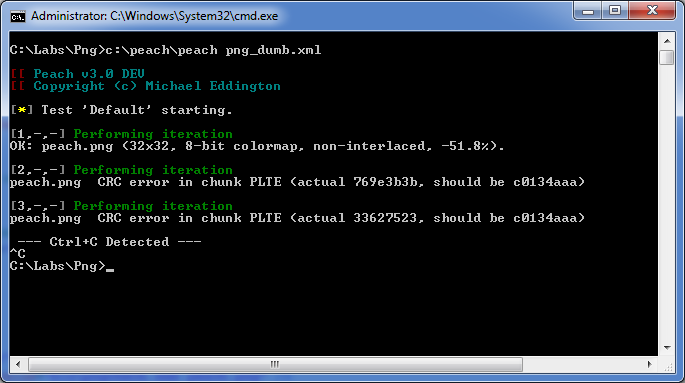


Figure 3 - Running the fuzzer

### Part B – Dealing with the PNG checksum

We will now expand the png\_dumb.xml fuzzer to respect the required CRC32 checksums that exist in the PNG format. We will still try and keep the fuzzer as dumb as possible.

The tasks for Part B of this lab are as follows:

1. Use the PNG specification and the 010 Editor template, PNG12Template.bt, to determine where the CRC32 checksums are in the file format.
2. Modify the png\_dumb.xml fuzzer to understand PNG chunks and perform a CRC32 of the required fields using a Fixup element.

**Open the PNG specification and sample file in the 010 Editor:**

1. Double click the **PNG Specification** shortcut in the C:\labs\png folder.
2. Browse to and click the 3. File Structure to load the format section of the specification.

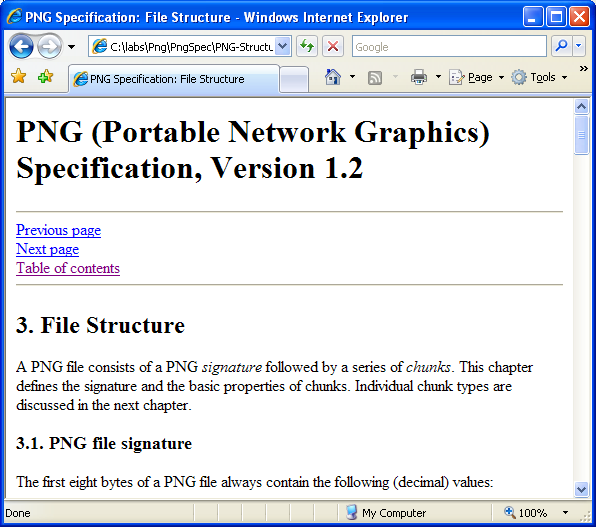


Figure 4 - PNG Specification, Section 3

1. Read sections 3.1 and 3.2 of the PNG Specification
2. Next, load the sample files in the 010 Editor, which can be located in the Windows Start Menu.

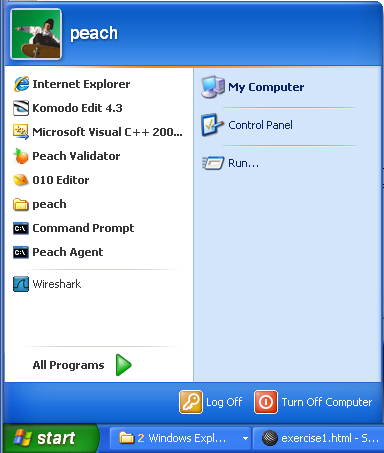


Figure 5 - Start menu showing 010 Editor

1. Load the “PNG12Template.bt” by clicking the File -> Open menu and browsing to “c:\labs\png” and selecting “png12template.bt”.
2. Next, load c:\labs\png\sample.png by clicking the File -> Open menu.
3. Select the PNG12Template.bt template in the “Run Template” drop down box.

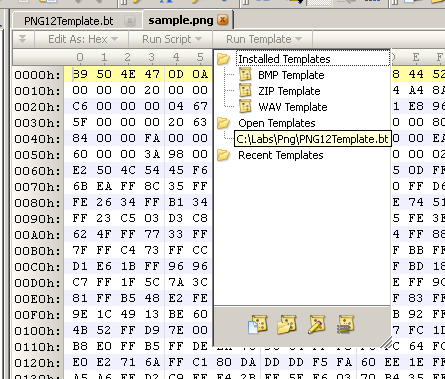


Figure 7 - Selecting PNG12Template

1. You should now see the template results below, double click on the tree view to expand.

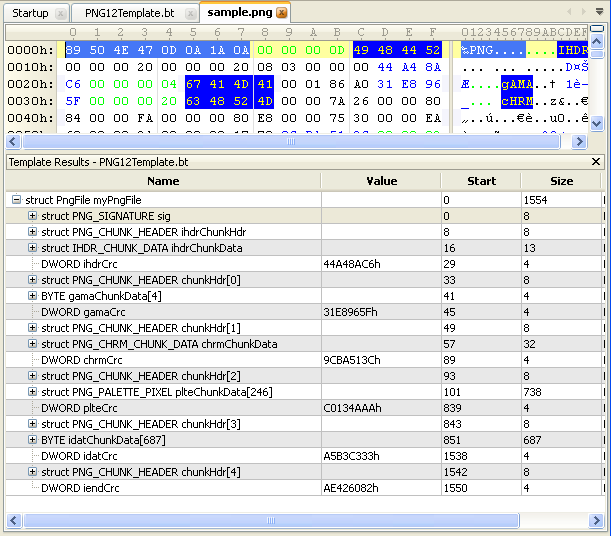


Figure 9 - Template results

PNG12Template will parse the sample PNG file and reveal its inner structure – acting as a good base reference for our DataModel.

Using the 010 Editor and the PNG specification we can see that a PNG file is simply a series of similarly defined Chunks where each Chunk starts with an 8 byte Chunk identifier (aka magic). Using Peach, we can define a generic Chunk DataModel that will act as a base for each Chunk type identified. More so, we can inform Peach that this base structure may occur more than once per PNG file. This will allow Peach to more effectively “crack” the PNG.

**Add new “Chunk” data model**

1. Before the existing **Png** DataModel, add a new DataModel called **Chunk**.
2. Using the PNG specification, add the following data elements to **Chunk**:
   1. Length
   2. Type
   3. Data
   4. CRC32
3. Create a size-of relation between **Length** and **Data** by adding a Relation element to **Length**.

**Add a Fixup to “CRC32” to perform the checksum**

1. Add a Fixup element to **CRC32** using the Crc class.

**Add PNG file signature and Chunks to “Png”** DataModel

1. Using the existing Blob, add the static PNG file signature from the PNG specification (see section 3.1).
2. Add a Block element that references the **Chunk** data model with a minOccurs of 1 and a maxOccurs of 1024.

**Validate the fuzzer using Peach Validator**

1. Start the Peach Validator (see Part A for instruction on how)
2. Load png\_dumb.xml and sample.png
3. Does the sample load correctly into the data model? If not go back and fix any problems.

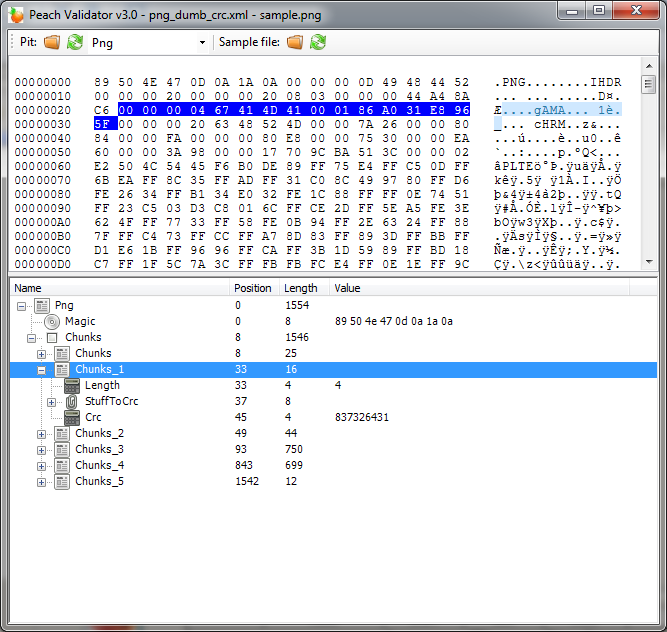


Figure 10 - Data loaded into Peach Validator

**Run the fuzzer and evaluate output**

1. Re-run the fuzzer (c:\peach\peach png\_dumb.xml) and evaluate the output.
2. Do we still get CRC errors or is the output different?

### Part C – Add monitor and logging support

Now that we are creating data that will make it into the parsers, we should configure an Agent, Monitors, and Loggers so we can detect any faults that may occur.

The tasks for Part C of this lab are as follows:

1. Modify the local Agent to use Page Heap Monitor to help us detect faults.
2. Additionally, configure a file system logger in the Test element to log the fuzzing results.

**Add** Agent **configuration**

1. Add the Page Heap monitor, PageHeap, which will take a single parameter named Executable with a value of pngcheck.exe.

<!-- Agent configuration to monitor our test app -->

<Agent name="LocalAgent">

<Monitor class="WindowsDebugger">

<Param name="Executable" value="bin\pngcheck.exe"/>

<Param name="Arguments" value=" c:\labs\png\peach.png"/>

<Param name="StartOnCall" value="ScoobySnacks"/>

</Monitor>

<Monitor class="PageHeap">

<Param name="Executable" value="pngcheck.exe"/>

</Monitor>

</Agent>

**Add a** Logger **configuration**

1. Add a Logger element to Test.
2. Configure the Logger to log information to the logs directory.

<!-- Log the results of our fuzzing run! -->

<Logger class="File">

<Param name="Path" value="logs"/>

</Logger>

**Run the modified fuzzer**

1. Run the fuzzer as previously shown.

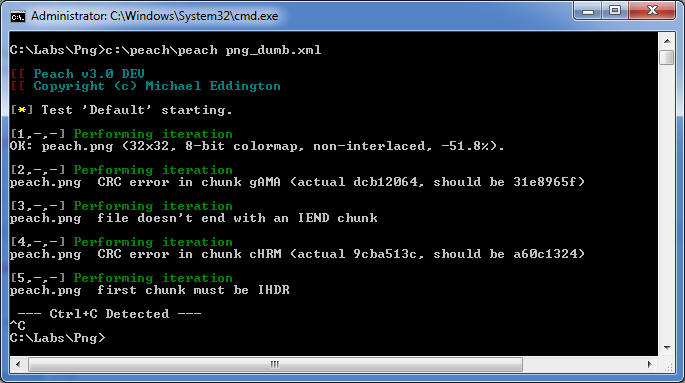


Figure 11 - Peach agent running

### Part D – Targeting a GUI application

Now that we have seen how to fuzz console applications, it’s time to try out a GUI image viewing application. After all, we are testing PNG images!

The tasks for Part D of this lab are as follows:

1. Modify Monitors to use the new application executable name VisualPng.exe.

\* Note: VisualPng.exe must have the **full path** to the Png in its argument. \*

<!-- Agent configuration to monitor our test app -->

<Agent name="LocalAgent">

<Monitor class="WindowsDebugger">

<Param name="Executable" value=" bin\VisualPng.exe"/>

<Param name="Arguments" value="c:\labs\png\peach.png"/>

<Param name="StartOnCall" value="ScoobySnacks"/>

</Monitor>

<Monitor class="PageHeap">

<Param name="Executable" value="VisualPng.exe"/>

</Monitor>

</Agent>

**Running the fuzzer**

If we have done everything correctly, when we run the fuzzer we should see the VisualPng application start and close with every round. Go ahead and run the fuzzer and see if it works.

### Part E – Adding some smarts

Now we will take our fuzzer another step by adding basic understanding of the PNG chunks in our sample file. For this lab, we will simplify and only add DataModels for the chunks that are used in our sample file.

The tasks for part E of this lab are as follows:

1. Using the 010 Editor and the “PNG12Template.bt” template script determine which chunks make up our sample PNG file.
2. Copy the contents of “png\_dumb.xml” into “png.xml” to use as a starting place.
3. Create data models for each of the chunk types in the PNG using 010 and the PNG Specification. The data models should extend the “Chunk” data model.
4. Use the “--debug” option for Peach and the Peach Validation UI to verify it works.

**Creating our smart PNG fuzzer**

This part of the lab will involve heavy class/teacher interaction as we walk through everything that needs to be done. Please follow the directions provided by the instructors to complete this part of the lab.

### Part F – Add Pit to Web UI

The last step is to add this pit to the web UI and run it.

1. Copy the PNG pit into the pits/Training/Labs (create if needed)
2. Remove existing agent configuration
3. Create a .config file for this pit. Reference the lecture slide deck for an example.
4. Verify the pit shows up in the web UI
5. Configure the pit and run it until you find a fault

Once you have found a fault move on to the next lab.

### What’s left?

We have created a fairly good PNG dumb fuzzer, and a fairly good PNG smart fuzzer. However, we stopped short of defining the entire PNG format. To take this example further, we would include other chunk types, ZIP compress, and other aspects of the PNG specification. This is not to say that the current fuzzer doesn’t find bugs. In fact, while building this lab issues were found in Windows XP SP3 using this PNG fuzzer.

## Lab: Network Server Fuzzing

In this lab, you will create a fuzzer that will act as a QuakeWorld client. Throughout, you will gain experience using the network based Publishers, slurp, various Monitors, and an introduction to fuzzing network servers.

### Part A - Understanding the QuakeWorld Protocol

The QuakeWorld protocol is a fairly simply protocol to grasp as it is comprised primarily of ASCII characters that provide a clear indication of what a given packet is attempting to communicate.

Before we start up the server and client, we need to identify our IP addresses. After loading the Peach **Target** Virtual Machine run the “ipconfig” command to locate your IP address.

One of the easiest means of understanding the QuakeWorld protocol is to watch it in action. To do so, load up the **Target** VM and launch the QuakeWorld Server:

**C:\Labs\quake\QuakeServer> qwsv.exe**

Next, switch to the **Peach** VM and start a capture within WireShark, which is in the Start Menu. Finally, run the QuakeWorld client:

**C:\Labs\quake\QuakeClient> qwc.bat safe**

Once prompted, perform the following:

1. Enter **connect SERVERIP** and press **Enter**. This will cause your client to connect to the QuakeWorld server. Note: replace SERVERIP with the Target virtual machines IP address.
2. Press **Esc**
3. Scroll down to **Quit** then press **Enter**
4. Press **Y** to exit the client.

At this point, we have all the info we need in WireShark to start building our fuzzer.

Spend a couple minutes reviewing the captures and try to answer the following question:

1. What is the basic structure of QuakeWorld packet?

Once you have a good understanding of this, open up the template found at **c:\labs\quake\quake.xml** and create a DataModel that can be reused for each packet sent by the server and client.

Once the basic structure is complete, go ahead and complete the StateModel and DataModels for the following packets:

1. GetChallenge
2. Challenge
3. Connect
4. Connected

**Note:** When defining the Publisher, you will pass it parameters that inform it which host and port to listen on. Currently, you must provide the publisher with the parameters in a specific order, first host, followed by the port parameter.

**Hint:**  You will need to use slurp to send the appropriate challenge back to the server. Also, when defining your Publisher, remember to list the **host** parameter before the **port** parameter ☺

Now we are ready to run the fuzzer:

**C:\peach\peach c:\labs\quake\quake.xml**

While the fuzzer is running, observe the QuakeWorld server window to get an idea of what occurring.

### Part B – Add Pit to Web UI

The last step is to add this pit to the web UI and run it.

1. Create a .config file for this pit. Reference the lecture slide deck for an example. Make sure to expose any items such as IP addresses and port numbers.
2. Verify the pit shows up in the web UI
3. Configure the pit and run it until you find a fault
   1. A remote agent running in the target VM
   2. Fault detection using a debugger for fault detection and automation
   3. Network capture for data collection

Once you have found a fault move on to the next lab.

## Lab: REST API Fuzzing

The goal of this lab is to fuzz a REST API service identifying SQL injection and stability issues. The lab is divided up into parts A through D. Complete each part in order.

In this lab all components will run on the Peach VM. The Target VM will not be used.

### Part A: Model the API

The goal of part A is to create a working data and state models that exercises the documented REST APIs in the target application. To this end you will need to review the Rest publisher documentation and the target rest API specification.

Resources:

* Peach developer documentation. Bookmarked in Chrome browser.
* Target REST API documentation in rest\_target.html
* Call examples from python in client.py

Rest Target:

* Starting: “python rest\_target.py”
* IP Address: 127.0.0.1
* Port: 5000

### Part B: Control iterations

The goal of part B is to enable control iterations and modify pit as needed to support. This part will use a new feature of state models called a final state. Currently we have used an initial state, but state models also support a final state that could be of use during this lab.

This part is completed when control iterations are enabled and no control iteration faults are reported.

### Part C: Monitor for SQL injection

The goal of part C is to add the ability to detect SQL injection vulnerabilities found during fuzzing. For this we will need to monitor the SQL database for SQL query compile errors. SQL query compile errors or syntax errors will only occur when SQL injection has been discovered. The method used to monitor for such errors is database dependent. Our lab target is using SQLite v3 and the application will send SQL error messages to the syslog facility.

SQLite v3’s error message: syntax error

Task: Modify pit to catch SQL injection errors using the Syslog monitor.

This part is completed when you have found one or more SQL injection faults.

### Part D: Monitor for 500 errors

The goal of part D is to monitor for 500 errors. Some REST targets use 500 status codes to indicate an abnormal error. These may be interesting to capture as faults. Read the Rest publisher documentation to discover how to enable this.

This part is completed when faults are found on 500 status codes.

### Part B – Add Pit to Web UI

The last step is to add this pit to the web UI and run it.

1. Expand the .config file for this pit as needed. Reference the lecture slide deck for an example. Make sure to expose any items such as IP addresses and port numbers.
2. Remove any Agent configuration
3. Install in the c:\peach\pits\Labs folder (create the folder if it does not exist)
4. Verify the pit shows up in the Library section of the Peach UI
5. Configure the pit and add monitoring
6. Run until you find a fault

Once you have found a fault ask the instructor to review you’re work and move on to the next lab.

## Lab: Network Client Fuzzing

In this lab you will write an evil Yahoo server that will target the Pidgin instant message client. Concept’s such as slurping, implementing both input and output based Actions, debugging Monitors, and server-based Publishers will be reinforced. Unlike the QuakeWorld protocol, the YMSG protocol contains many structures that are comprised of non-ASCII values. In this lab, you will develop the skills required to identify patterns in binary protocols and create effective fuzzers that target them.

### Part A – Understanding the YMSG Protocol

One of the easiest means of understanding the YMSG protocol is to watch it in action. To do so, we can leverage the fine work of the people that wrote a fairly complete WireShark parser for the YMSG protocol. You can find a series of packet captures under **c:\labs\yahoo\captures**. Spend a couple minutes reviewing the captures and try to answer the following questions:

1. What is the basic structure of an YMSG packet?
2. Where will we need to create Relations?

Once you have a good understanding of this, open up the template found at **c:\labs\yahoo\yahoo.xml** and create a DataModel that can be reused for each packet sent by the server and client.

Once you have the basic template created, use

**c:\labs\yahoo\capture\** **ymsg\_client\_logon\_empty\_buddy\_list\_with\_buddy\_notification.pcap**

To build out the following DataModels:

1. Client’s authentication request
2. Server’s challenge
3. Client’s response

Try not to create huge Blobs that contain a ton of data. This will limit the fuzzer’s ability to intelligently mutate the data. The goal of this lab is to provide experience in intelligently defining data structures.

**Note:** It may also be helpful to load up the YMSG 010 Editor template found at **c:\labs\yahoo\YMSG\_Parser.bt**. You’ll also find a web archive in the same folder that may provide some guidance on decoding the authentication scheme.

Once your DataModel’s validate, add the necessary StateModel, Publisher, Test, and Run sections to your PIT file.

**Phase B – Setting up the Pidgin and Testing your DataModel**

Now that we hopefully have the correct DataModel, Publisher, and StateModel setup, let’s start configuring the VMs to talk to our evil YMSG server.

1. Open up the **Target** VM and execute Pidgin from the Start Menu.
2. Click on **Account -> Manage**
3. Click **Add**
4. Select the **Yahoo** Protocol
5. Enter **bh\_peach\_fuzz2** for the username
6. Enter **blackhat123** for the password
7. Check **Remember Password**
8. Click the **Advanced** tab
9. Change the **Pager Server** to be **CLIENT\_IP** (this is the address of the Peach VM)
10. Click **Save**
11. Uncheck the **Enable** box
12. Click **Close** on the **Accounts** Window
13. Click on **Tools->Preferences** in the main Pidgin window
14. Set the **Show system tray icon** to **Never**
15. Make sure Offline Buddies is checked, click on **Buddies -> Show -> Offline Buddies**

Now, switch over to the **Peach** VM and perform the following:

1. Execute your fuzzer
2. Start WireShark

Now, switch back to the **Target** VM and perform the following:

1. In Pidgen, click on **Accounts** -> **Enable Account** -> **bh\_peach\_fuzz2 (Yahoo)**

If all went well, you should see the following packets in WireShark:

1. YMSG : Authentication (status=Default)
2. YMSG : Authentication (status=Server Ack)
3. YMSG: Authentication Response (status=Default)

If you do not see these packets, it more than likely means the values in your DataModel are incorrect. Compare them against the captures provided.

Once all your packets match, move on to the next phase.

### Part C – Getting Pidgin Some Buddies

Now - the final leg to get Pidgin online and a list of buddies. In this phase we will be defining two fairly large structures. You’ll be defining packets 4 and 5 from **ymsg\_client\_logon\_empty\_buddy\_list\_with\_buddy\_notification.pcap** and adding the necessary State elements to support them. Once this is complete Pidgin will think it’s authenticated to a YMSG server, display a list of buddies, and present a Buddy Request notification. To help visually identify that Pidgin is happily connected by preventing Peach from closing the connection immediately, add the following Action to the bottom of your StateModel:

<Action type="close" onStart="time.sleep(15)"/>

And the following python import to the top of your pit:

<Import import="time"/>

Once Pidgin is convinced it’s talking to a real YMSG server we can start to fuzz it.

### Part D – Fuzz It

Currently, no publisher exists that provides the ability to listen locally on a socket while performing remote process actions, such as killing or starting Pidgin. As such, we’ll want to execute Pidgin locally to our fuzzer so we can regain the ability to start and kill the process in an automated manner. To achieve this, perform the following on the **Peach** VM:

1. Start Pidgin and configure it in the same manner as we did earlier. Only this time, set the **Pager Server** to **127.0.0.1**
2. Update the publisher as follows:

<Publisher class="TcpListener">

<Param name="Host" value="0.0.0.0"/>

<Param name="Port" value="5050"/>

</Publisher>

1. Configure a WindowsDebugger monitor in your agent configuration to run pidgin.exe.
2. Run your fuzzer ☺!

At this point you should see pidgin opening, connecting, and closing automatically ☺ Fun Huh!?

THE END