Gokul Caushik
CS 569

Peach Fuzzer w/ SWARM?

The goal of this project was to see the effectiveness of SWARM by building or using a fuzzer, incorporating SWARM into it, and testing whether the addition of SWARM makes the fuzzer better at finding bugs.
The fuzzer I picked was the Peach fuzzer, and I came across it as I was looking for open source fuzzers. Peach is one of the more popular open source fuzzers out there today.
You can get the source code for it from sourceforge; I downloaded Peach version 3.0.
I will cover all the steps I took in this whole project, even though I had started the project before this term I still think it is good to have all the work I did documented, so that is the reason why it is in this report.

You are already familiar with the concept of SWARM testing so I won't go into what it is.
My plan was to fuzz HTML, using the Peach fuzzer to generate sets of HTML files and running them on Firefox.
One set of the generated files would include all the features of HTML.
To incorporate SWARM, I would have another set of files generated that will only include a subset of the features of HTML.
Then I would run the fuzzer on both sets of files and compare the amount of crashes/bugs.

The first step was to identify the features I wanted to test, and for HTML it would obviously be based on tags.
Here is a list of possible features:

Tags used for text (font, text color, etc.)
Java applets
Client side scripts
Forms
Media (Audio, Video, other plugins)
Images/Figures
Menus
Drop-Down Lists
Description Lists
Ordered/Unordered Lists
Presentational Attributes (such as subscript, superscript, etc.)
Headers/Footers
Phrase Tags (if you want to embed code, variables, etc.)
Graphics
....
....
and more.

Regular random testing will generate files which have all these features, but SWARM testing will generate files that have a subset of these features.
The next step would be to have Peach generate these files.

Peach Fuzzer Overview

Before I go into the challenges I had when trying to run Peach and generate files, I want to talk about the Peach fuzzer in general, because I spent an inordinate amount of time trying to figure out the structure of the fuzzer and how it worked.
I spent a lot of time reading the Peach documentation (which is not great by any means) and whatever I understood I want to relay it here.
It is absolutely essential to understand how this thing works or I was going to have a lot of trouble fuzzing.

Peach pit file

The pit file is an XML document that you feed to Peach as a command line argument, and this file tells Peach exactly what data it is that you are fuzzing, what actions Peach takes in order to fuzz the data, what target application to run the fuzzed data on, and what to use for monitoring and logging faults.
I had to make a pit file that fuzzes HTML with the target application being Firefox.
In the peach/samples folder after installation there are many sample pit files that you can run.
Those were helpful for me to create my own pit file.
My pit file is named Firefox.xml.

Now I will explain what goes in the pit file.

Data Model

The data model describes the data to be fuzzed.
The data can be in the form of Strings, Blocks, and Blobs.
Now, looking at the documentation and the examples, most of the examples show data models that have a very fixed structure and are easy to model.
File structures like zip, wav, etc. have a very well defined structure and you know that every file of that type will have the same structure.
The issue with HTML is you only have a small set of tags that are required for any HTML file, but other than that there could be huge variability in tags from one HTML file to the next.
I don't know how to capture that in the data model.

Instead I used a string analyzer which will take all the text and transform it into a tokenized tree.
I read that this is a faster way to create a data model, if it is difficult to specify a precise data model you can just use this.
I needed to do this because I was taking too long racking my head over the data model, I wanted to get the fuzzer to run.

```
<DataModel name="HTML">
        <String type="utf8">
                <Analyzer class="StringToken" />
        </String>

</DataModel>
```

State Model

The state model tells the fuzzer what actions to perform.
It is used to specify how the fuzzer sends and receives the data to fuzz.
Maybe you want to send the data through a socket, or a file for the fuzzer to open, or whatever it may be.
In the case of HTML I felt that the data should be sent as a file which the fuzzer will open.
Actions are then carried out by the Publisher, which I will get to later.

I used 3 actions in my state model.
The first is to send the HTML file to the Publisher to be read.
I put a file name in there to begin with which is just a random HTML file.
My thinking is that the fuzzer will take this file and mutate it.
The next action is to close the file, which is trivial.
And the last action is to call the Publisher, and I just chose a default publisher called Peach.Agent.
From what I read you can use a generic name for the method, so my method name is “Start”.

```
<StateModel name="State" initialState="Initial">
        <State name="Initial">
                <Action type="output">
                        <DataModel ref="HTML" />
        <!--
            Peach will use this file as a starting point for generating fuzzed
            files. It will therefore need to figure out how this file can be
            interpereted based on the DataModel specified above (ref="HTML").
        -->
                        <Data fileName="/home/gokul/1.html" />
                </Action>
                <Action type="close" />

                <Action type="call" method="Start" publisher="Peach.Agent"/>
        </State>
</StateModel>
```

Agents and Monitors

Agents and monitors are used for debugging purposes to analyze crashes.
Peach gives a huge list of different monitors to use, depending on what you are doing, whether it is debugging, or detecting crashes, or checking for memory leak, etc.
Every peach pit file should have these so I just have a local agent as I said before, and I am monitoring Firefox for crashes, so I used the Process monitor to watch the Firefox process.
The XML for this looks fairly straightforward.

```
<Agent name="LocalAgent">
        <Monitor class="Process">
                <Param name="Executable" value="/usr/bin/firefox" />
                <Param name="Arguments" value="/home/gokul/1.html" />
        </Monitor>
</Agent>
```

As you can see I used the binary for Firefox as the process to launch, and the argument tells the monitor to launch my HTML file in the state model (the one that should be getting mutated).

Test

The final part of the pit file is the testing portion.
This is where the agent, the state model, publisher, and logger are configured.
You can also specify mutation testing strategies.

```
<Test name="Default">
        <Agent ref="LocalAgent"/>
        <StateModel ref="State"/>

        <!-- This publisher will write our file. -->
        <Publisher class="File">
                <Param name="FileName" value="fuzzed.html"/>
        </Publisher>

        <!-- To enable logging we must include this -->
        <Logger class="Filesystem">
                <Param name="Path" value="logs"/>
        </Logger>
</Test>
```

The agent I just referred to the agent I was using above, and the state model I just referred to the state model I was using above.

Publisher

The publisher is what carries out the actions in the state model.
Peach offers a large variety of publishers, it also give you the ability to create your own publishers.
I used the File publisher given by Peach, which reads and writes a file.
I wanted to use this publisher to read and write HTML files.
The HTML file written is called fuzzed.html.

Logger

The logger is there to log any crashes that may happen.
I used one because I want to be able to look at the crashes and what happened.
The logs are in a folder called “logs” in the peach main directory.

Test Results

I only ran for a limited number of iterations because every iteration it opened a new Firefox process and I didn't want too much on my system.
I captured the results in a file called “output”.
It shows the different mutations the fuzzer used, so I can verify that the fuzzer did indeed run, and it did fuzz my file.
However I couldn't detect any crashes.
I probably should let it run longer which I will do in the future...but I may have to change the pit file so that it doesn't launch a new process every iteration.

Here is how the output of one iteration looks like:

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

[*] Test 'Default' starting with random seed 29946.

[R1,-,-] Performing iteration

[1,-,-] Performing iteration
[*] Fuzzing:
HTML.DataElement_0.DataElement_0.DataElement_3.DataElement_6.DataElement_9.DataElement_12.DataElement_15.DataElement_18.DataElement_21.DataElement_24.DataElement_27.DataElement_30.DataElement_33.DataElement_36.DataElement_39.DataElement_42.DataElement_45.DataElement_48.DataElement_51.DataElement_54.DataElement_57.DataElement_60.DataElement_63.DataElement_66.DataElement_69.DataElement_72.DataElement_75.DataElement_78.DataElement_81.DataElement_84.DataElement_87.DataElement_90.DataElement_93.DataElement_96.DataElement_99.DataElement_102.DataElement_105.DataElement_108.DataElement_111.DataElement_114.DataElement_117.DataElement_120.DataElement_123.DataElement_126.DataElement_129.DataElement_132.DataElement_135.DataElement_138.DataElement_141.DataElement_144.DataElement_147.DataElement_150.DataElement_153.DataElement_156.DataElement_159.DataElement_162.DataElement_165.DataElement_168.DataElement_171.DataElement_174.DataElement_177.DataElement_180.DataElement_183.DataElement_186.DataElement_189.DataElement_192.DataElement_195.DataElement_198.DataElement_201.DataElement_204.DataElement_207.DataElement_210.DataElement_213.DataElement_216.DataElement_219.DataElement_222.DataElement_225.DataElement_228.DataElement_231.DataElement_234.DataElement_237.DataElement_240.DataElement_243.DataElement_246.DataElement_249.DataElement_252.DataElement_255.DataElement_258.DataElement_261.DataElement_264.DataElement_267.DataElement_270.DataElement_273.DataElement_276.DataElement_279.DataElement_282.DataElement_285.DataElement_288.DataElement_291.DataElement_294.DataElement_297.DataElement_300.DataElement_303.DataElement_306.DataElement_309.DataElement_312.DataElement_315.DataElement_318.DataElement_321.DataElement_324.DataElement_327.DataElement_330.DataElement_

333.DataElement_336.DataElement_339.DataElement_342.DataElement_345.DataElement_348.Data
Element_351.DataElement_354.DataElement_357.DataElement_360.DataElement_363.DataElement_
366.DataElement_369.DataElement_372.DataElement_375.DataElement_378.DataElement_381.Data
Element_384.DataElement_387.DataElement_390.DataElement_393.DataElement_396.DataElement_
399.DataElement_402.DataElement_405.DataElement_408.DataElement_411.DataElement_414.Data
Element_417.DataElement_420.DataElement_423.DataElement_426.DataElement_429.DataElement_
432.DataElement_435.DataElement_438.DataElement_441.DataElement_444.DataElement_447.Data
Element_450.DataElement_453.DataElement_456.DataElement_459.DataElement_462.DataElement_
465.DataElement_468.DataElement_471.DataElement_474.DataElement_477.DataElement_480.Data
Element_483.DataElement_486.DataElement_489.DataElement_492.DataElement_495.DataElement_
498.DataElement_501.DataElement_504.DataElement_507.DataElement_510.DataElement_513.Data
Element_516.DataElement_519.DataElement_522.DataElement_525.DataElement_528.DataElement_
531.DataElement_534.DataElement_535.DataElement_4056.DataElement_4059.DataElement_4060
[*] Mutator: UnicodeStringsMutator
[*] Fuzzing:
HTML.DataElement_0.DataElement_0.DataElement_3.DataElement_6.DataElement_9.DataElement_
12.DataElement_15.DataElement_18.DataElement_21.DataElement_24.DataElement_27.DataElement
_30.DataElement_33.DataElement_36.DataElement_39.DataElement_42.DataElement_45.DataElemen
t_48.DataElement_51.DataElement_54.DataElement_57.DataElement_60.DataElement_63.DataEleme
nt_66.DataElement_69.DataElement_72.DataElement_75.DataElement_78.DataElement_81.DataElem
ent_84.DataElement_87.DataElement_90.DataElement_93.DataElement_96.DataElement_99.DataEle
ment_102.DataElement_105.DataElement_108.DataElement_111.DataElement_114.DataElement_117.
DataElement_120.DataElement_123.DataElement_126.DataElement_129.DataElement_132.DataElem
ent_135.DataElement_138.DataElement_141.DataElement_144.DataElement_147.DataElement_150.D
ataElement_153.DataElement_156.DataElement_159.DataElement_162.DataElement_165.DataEleme
nt_168.DataElement_171.DataElement_174.DataElement_177.DataElement_180.DataElement_183.Da
taElement_186.DataElement_189.DataElement_192.DataElement_195.DataElement_198.DataElement
_201.DataElement_204.DataElement_207.DataElement_210.DataElement_211.DataElement_2673.Dat
aElement_2676
[*] Mutator: DataElementSwapNearNodesMutator

You can see the different mutators being used, and I am impressed that Peach has so many different mutators.
It is definitely something I can experiment more with.

Difficulties and SWARM

Right now I am not sure how to incorporate SWARM into the pit file.
I think one way to do that is to manually create pit files that have less features, but I remember you saying that it isn't the best idea.
It would be much better to do this in an automated way.
However I cannot figure out how to get the fuzzer to generate files by picking and choosing features.
From what I understand Peach gives the user the ability to do file generation and file mutation.
This term I focused more on the file mutation aspect, which in my opinion was easier for a beginner to understand.
But in order to incorporate SWARM I would also need to know the file generation aspect.
I need more expertise with the tool in order to do that.

Conclusion and what I learned

I know this project didn't reach the final goal that I had when I started (which is to incorporate SWARM), but that doesn't mean I am done with the project.
This project goes beyond school.
As far as where this project is headed, have I reached a dead end here?
Did I choose the correct fuzzer for this project?
These are questions I have to ask myself.
I was happy to learn about Peach, but I may have to look into other fuzzers too.
Or even making my own.
I think I put a lot of emphasis on the fuzzer, instead of the goal which is SWARM.

But for now it doesn't take away from the fact that I have learned much already, and there is also much more to be learned.
I knew in the beginning that this is going to be an ambitious project, and it definitely proved itself to be that.
The reason it is an ambitious project is because of the fuzzer I chose.
This fuzzer is very difficult to get your head wrapped around, much so that the company that owns Peach, Deja vu Security, offers conferences solely devoted to training people on Peach.

I liked when I did this work in summer because I could spend all my time on this, but with other classes it is not a simple task.
But considering how difficult this fuzzer is to understand, I think I did well to get this far.
Everything I did for this project, I did not have any previous experience in, I had to pick it up on the fly.
At this point I can get the Peach fuzzer to run and do some fuzzing.
I got knowledge on how to create a Peach pit file.
Do I know all the details? Far from it.
However I think I know enough basics that I can go deeper now.
Before this term I didn't even know anything about Peach pit files.
Before this term I didn't have any understanding of the Peach framework.
I like Peach and I am not done learning.

Testing is always going to be one of my interests and it should always be an area I will be involved in, so nothing I did was wasted.
Fuzzing is a very important concept to know and learning to tackle a difficult framework like Peach is worth my time.
I hope I can gain more insight because this is something I can bring to wherever I go to work.
Peach is a quite powerful fuzzer, but I would say it is unknown in the industrial world.
However I think it could be useful to know, because it will make me a better tester for sure.