“Fuzzing” Software Security Lab: Introduction to Security Testing using Fuzzers

The objective of this lab is to experiment with the concepts of fuzzing, that is performing security tests that operate on the data manipulated by the program in order to unearth bugs and potential vulnerabilities. You will test simple applications in the course of this lab.

Part 1: Introduction

Security testing is more or less precise depending on the capabilities of the tester:

- whitebox testing assumes that source code is available to the attacker. This is the most favorable situation, which may actually arise for all open-source software or for some code auditors. In this situation, the adversary can evaluate the consequences of a bug or of error in the code;
- greybox testing assumes the availability of the compiled binary only, but this binary might be instrumented for further investigations;
- blackbox testing finally assumes only that the tested can control the program’s input and observe the program’s output. This will generally be the case when attacking a distributed application.

Fuzzing consists in automatically feeding data to a program with the intent of causing the program to crash or expose a bug. When fuzzing, we perform two tasks: (1) we alter the program inputs, either through the generation of random or mutated values or based on a model of said inputs, and (2) we then compare the actual behavior of the software using such inputs thanks to an oracle (a specialized program) that determines what should be observed instead. A very basic technique for fuzzing (even though it is neither flexible nor reproducible) is to pipe /dev/urandom into a program’s input (you can try that on some of our test programs later on).

The fuzzing technique is commonly used by both white and black hats (security experts performing a code audit and hackers trying to find vulnerabilities). Fuzzing indeed makes it very simple to discover missing input validations that may crash an application and abnormal behaviors that may be exploited as another consequence of unanticipated inputs. The strength of fuzzing is that it is able to generate only limited types of ill-formed inputs and to recognize only a limited number of poor behaviors, but entirely automatically and as much as possible reproducibly (in order to further investigate the case). Fuzzing relies on a specific tool, a fuzzer is generally adapted to the target of the attacks. One can notably distinguish:

- environment variables that condition the execution of the program
- command line arguments/parameters
- parameters of system calls
- file content and format
- network protocols and APIs (notably for webapps), including their format (structure of messages), the encoding used (endianness, signature, range …), or the content provided. In the following, we will actually not consider this type of fuzzing.

Because of this very large number of targets, there also exists a variety of fuzzers that have been developed in the last 15 years (the first fuzzer, Barton Millers' "Operating System Utility Program Reliability – The Fuzz Generator", dating from 1988). For instance, Peach, SPIKE, Sulley, Codenomicon have become famous names in the area, but many more products exist.

Fuzzing therefore constitutes the first step in vulnerability assessment, even in the context of blackbox testing, and guides the investigations. Being a test, it is of course by no means exhaustive, but its coverage of the different behaviors exhibited by software is generally improved in smart fuzzing approaches.

Part 2: Fuzzing with Zzuf

You will first try out the C based fuzzer zzuf. You have to install the necessary packages for the lab using a Linux distrib (for instance running a virtual machine or your own Linux). Debian Stretch is known to work with zzuf, as well as Ubuntu 12.04 (for instance the old SEEDUbuntu virtual machine from the SEED labs). However, Ubuntu 16.04 has issues.

Now install zzuf as follows:

```
sudo apt-get install zzuf
```

Second install the following packages:

```
sudo apt-get install antiword

sudo apt-get install mplayer
```

Also download the *smiley.txt* and *test.wmv* file from the course's website, and the *Rapport.doc* file (a file from Polytech's web site).

In this part of the lab, we will experiment with zzuf in order to try out input fuzz testing. This fuzzer provides a very convenient interface in order to manipulate the data handled by a software, be they inputs, variables, or the content of files. With zzuf, the user is able to control the seed for the pseudo-random test generation. In this respect, we are able to consistently reproduce the exact same behavior. This may enable a deeper inspection of software, and notably to distinguish concurrency faults that would be exploited from other faults which are deterministic. You can get information on the different options of zzuf by calling:

```
zzuf -h
```

We will fuzz files used by different programs.

First of all, we will look at the cat program. This program displays the content of file. Examine the content of smiley.txt:

```
cat smiley.txt
```

Now run the same program but after intercepting its dataflows through zzuf, as follows:

```
zzuf –r 0.01 cat smiley.txt
```

The parameter –r determines the ratio of fuzzing performed (0.1 percent in this case). What do you observe if you change this value to 0.1? Now, the following command prevents the generation of non-printable characters:

```
zzuf –r 0.1 –R '\x00-\x20\x7f-xff' cat smiley
```

In what way does this change the output?

We will now examine how zzuf can fuzz a program like *file* which determines the content of a given file. First compare the result provided when calling:

```
file /bin/ls
```

and that provided when invoking zzuf:

```
zzuf –r0.01 file /bin/ls
```

Something's taking place behind the scenes. Now invoke zzuf again with the debug option (-d). What is the problem? How can you get rid of that problem using the exclude (-E) option?

Now that you've solved this problem, invoke zzuf again with another option: "-s0:5". This option triggers five runs of the fuzzer using pseudo-random generations of the 1 percent of modifications introduced. Why is the result different between runs?

We will now illustrate vulnerabilities of a software used to access Word documents, antiword. This utility prints a text-only version of a Word document. Let's fuzz it:

```
zzuf –C10 –q –s0:1000 –r0.001:0.02 –M1000 antiword Formulaire\ mobilite\ Polytech.doc
```

What do you observe? In what respect can that be connected to the range used in the ratio (-r) option and to the seed?

Now make sure that the video file was transferred correctly by running the following command (can be interrupted with Ctrl-C):

```
mplayer test.wmv
```

Let us now try to fuzz the content of this video. We can crash the program by fuzzing the content of the header, but we can trigger subtler crashes or vulnerabilities by modifying the video payload. This can be realized by specifying the offset and range at which modifications will take place using the bytes (-b) option. Try it for yourself, for instance:

```
zzuf –q  –s5:1000 –r0.001:0.1 –S –c –b 30000-70000 mplayer test.wmv
```

What else do you observe now? Try with different values for parameters.

Now that you are familiar with the tool, experiment with other binaries at your convenience and report on your trials.

Part 3: Fuzzing with Fusil

We will also use the Python fuzzer Fusil. You therefore need to install a few Python dependencies first:

```
sudo apt-get update

sudo apt-get install python python-ptrace
```

Download the following package (Fusil will be the first fuzzer we use):

http://pypi.python.org/packages/source/f/fusil/fusil-1.4.tar.gz

then extract it and install it as follows:

```
tar xzvf fusil-1.4.tar.gz

cd fusil-1.4

sudo python setup.py install
```

We will also need a number of applications to test from. Download the programs *vulntest.c*, *tabfuz.py*, and *tabfuz2.py* from the course web page.

In this part of the lab, you will use a fuzzer named Fusil in order to fuzz parameters on the command line. This fuzzer provides a convenient way to execute and even script multiple tests and other operations that must be performed to support a particular security test, notably with respect to observing the application behavior. It also stores the results of security tests into a log that can be browsed after fuzzing in order to understand what happened.

Fusil is a framework that provides both mechanisms for generating tests (under the form of random or mutated data), also called ActionAgents in this framework, and for observing the behavior of the

program under test, also called Probes in Fusil. ActionAgents can modify the execution of a process (CreateProcess) but also modify files (MangleFile, AutoMangle) or messages exchanged over sockets. Probes can detect specific patterns in output logs (FileWatch), or monitor the behavior of the computer and of the execution environment (CpuProbe, ProcessTimeWatch), or the output (WatchStdout) among others. You will find more details about the Fusil fuzzer at the following web site:

http://fusil.readthedocs.io/

We will use only simple ActionAgents and Probes related to process execution and simple output monitoring in this lab.

Go back to the directory where you downloaded vulntest.c and compile it as follows:

```
gcc –o vulntest vulntest.c
```

and try to find the vulnerability in this code. Explain the results you obtain with the following commands:

```
./vulntest 100

./vulntest 260

./vulntest 8000
```

Now, we will apply Fusil to our code example. Run the following command:

```
sudo python tabfuz.py -- sessions=1
```

The tabfuz script, which is described in the following listing, only configures a single execution, as specified by the sessions parameter, with a fixed command-line argument (50).

```
#!/usr/bin/python
from fusil.application import Application
from fusil.process.create import ProjectProcess
from fusil.process.create import CreateProcess
from fusil.process.watch import WatchProcess
from fusil.process.stdout import WatchStdout


class Fuzzer(Application):
        def setupProject(self):
                process = ProjectProcess(self.project,
```

```
                    ['./vulntest', '50'])
                WatchProcess(process)
                WatchStdout(process)


if __name__ == "__main__":
        Fuzzer().main()
```

What is the result of running that script. What kind of vulnerability can we possibly find with this script?

Now we will test a more complete script tabfuz2.py as follows:

```
#!/usr/bin/python
from fusil.application import Application
from fusil.process.create import ProjectProcess
from fusil.process.create import CreateProcess
from fusil.process.watch import WatchProcess
from fusil.process.stdout import WatchStdout
import random

class printTabProcess(CreateProcess):

        def __init__(self, project):
                CreateProcess.__init__(self, project, ["./blahblah"])
                self.num = '0123456789'

        def chooseNumber(self, maxlength):
                number = ''
                i = random.choice(range(1, maxlength))
                while i > 0:
                        number += random.choice(self.num)
                        i = i - 1
                return number

        def createCmdLine(self):
```

```
                arguments = ["./vulntest"]
                arguments.append(self.chooseNumber(5))
                return arguments

        def on_session_start(self):
                self.cmdline.arguments = self.createCmdLine()
                self.createProcess()

class Fuzzer(Application):
        def setupProject(self):
                process = printTabProcess(self.project)
                WatchProcess(process)
                WatchStdout(process)

if __name__ == "__main__":
        Fuzzer().main()
```

This script is more complex as it updates with every new session (every run of the fuzzer). The script will select a new random number as a parameter for the vulntest program. Now execute this script:

```
Sudo python tabfuz2.py
```

This time, we have not constrained the parameter passed by the fuzzer nor restricted the number of sessions performed by the fuzzer. What is the end result? Can you reproduce one bug based on the output of the fuzzer (hint: which value raised the issue and how do you find it)? Does this help understand the vulnerability? How can you generate more bugs (hint: look at the --success parameter)? Explore other constructs of Fusil.