GRAPEVINE: AUTOMATED MAC OS X SYSCALL FUZZER GENERATOR SECURITY RESEARCH AND DEVELOPMENT PROJECT REPORT SINGAPORE POLYTECHNIC

DISM 3A 90

SP Supervisor: Mr. Samson Yeow
CSIT Supervisor: Ms. Yong Ru Fui

Jeremy Heng (P1000720)
Ku Wee Kiat (P1030284)
Goh Kee Chin (P1030338)
Gerald Tan (P1030507)

August 6, 2012

Abstract

Fuzzers built to test for vulnerabilities and reveal bugs in the Mac OS Xs XNU Kernel are scarce, in both the open and closed source worlds. In addition, existing fuzzers lack the intelligence to adapt to a rapidly changing environment such as a regularly updated operating system kernel. Existing fuzzers do not take into account the semantics of system calls or the order in which these system calls are called. This report details the various stages of developing a highly abstractable automated fuzzer designed for the discovery of vulnerabilities and bugs in the Mac OS X's XNU kernel through the process of 'fuzzing'.

Contents

1 Acknowledgements

We would like to thank Ms. Yong Ru Fui and our CSIT supervisors for our extensive communication and meetings during which much well-received advice was given with regards to the direction of the project. Their feedback on the progress of the project was extremely crucial.

We would also like to thank Mr. Samson Yeow for the guidance on the business and marketing aspects of the project as well as for providing an alternative point of view as an assessor with application security as a focus.

We would also like to show appreciation to the freemusicarchive.org for making hundreds of music pieces licensed under the Creative Commons available for use in derivative works as well as Slinte whose Irish tune we have used for our commercial.

2 Abstract

An abstract of the identified problems and desired outcomes follows.

2.1 Summary of Problem and Needs

Fuzzers built to test for vulnerabilities and bugs in the Mac OS Xs XNU Kernel are scarce, in both the open and closed source worlds. In addition, existing fuzzers lack the intelligence to adapt to a rapidly changing environment such as a regularly updated operating system kernel. Existing fuzzers do not take into account the semantics of system calls or the order in which these system calls are called.

2.2 Summary of Objectives

The objective is to create a tool that fulfills the following requirements:

a) is capable of discovering bugs in the Mac OS X's XNU kernel through the process of fuzzing;

b) automatically and dynamically generate the system calls for execution based on attached semantic rules; and

c) provides a means by which further investigation may be carried out in the event of a detected failure of the kernel (e.g. kernel panic through corrupted kernel memory).

2.3 Summary of Security Functions

The tool would comprise of the following security-related functions:

a) parse system call header files;

b) generate fuzzing input to system calls;

c) logging of syscall results and fuzz data input

d) crash detection and automatic recovery of virtual machine; and

e) stateful fuzzing.

2.4 Summary of Implementation

The software will be deployed in a client-server model. The host controller, or ghost, will control multiple instances of fuzzing daemons, gfuzzd. Each gfuzzd instance is run in a virtual machine that ghost has control over.

It will be implemented with as many generic parts as possible so as it allow it to be reused in situations beyond the XNU kernel. Parts that are possibly abstractable and user controlled are: the mechanism by which system calls are made, the fuzz input generation, handling of how the host controller will react to certain events such as losing a connection to an instance, or handling received data.

In addition, shared components such as syscall profiling and the generators should only be coded once as a module and be used (and be reused) by other components that share this dependency as opposed to implementing a separate module for use in each.

3 Technical Definitions and Acronyms

System Call (Syscall),

: An exposed function to which userland programs may make requests for the kernel to do work.

Kernel

: The interfacing layer between hardware and software.

Userland

: Userland programs run at a level higher than the kernel. That is, it relies on the kernel to do work and achieves this through system calls.

Generator

: In the context of the Grapevine project, a generator is a user-subclassable class that provides the data to fuzz with to the Calling Mechanism. It also accepts a return value after the syscall is called with the generated data and may act on it to modify future data yields.

Fuzz,

: The process of putting possibly random input in a system through exposed interfaces in order to test for unexpected outcomes.

Calling Mechanism,

: In the context of the Grapevine project, a calling mechanism is a user-subclassable class that decides how fuzz input is run. The default calling mechanism in this project is one that calls XNU system calls. A calling mechanism that sends HTTP methods to fuzz a web application is also possible under the framework.

Callback

: A event handler whose reference is stored in advance and possibly called at a later time in response to conditions.

Signal

: Certain actions or errors cause signals to be thrown by the operating system. For example, a program with a segmentation fault problem will throw a SIGSEGV signal.

4 Foreword

4.1 Introduction

Group 90: Automated Syscall Fuzzer Generator. Fuzzing, when put simply, is the process of finding errors and bugs in software by supplying a mechanism that accepts input with unexpected data.

A metaphor: pumping water into a car instead of petrol. Doubtless, this would have a serious effect on the car's combustion system if it does not handle this faux fuel appropriately. Hence, in terms of software assurance and security, fuzzing is of grave importance.

The concept of fuzzing has only recently begun to capture widespread attention. Many vulnerabilities have been uncovered in popular applications which we use everyday like Microsoft Word, Microsoft Excel or web browsers such as Mozilla Firefox, Google Chrome, Safari.

This project is about fuzzing, arguably, the most important piece of software – the operating system kernel.

4.2 Background

Contemporary fuzzers in the market require much human interaction. For example, after an unrecoverable error occurs such as a kernel panic, users have to manually restart the machine and run the fuzzer again. Or, if a kernel is updated and changes were made to the system call list, the user would have to manually modify the fuzzer software to reflect the changes.

5 Requirement Analysis

5.1 Proposed Application

The proposed application is a fuzzing framework that may be extended to run on multiple platforms and to test a variety of systems including desktop applications, kernels, and web applications. In its shipped incarnation, it would able to execute system calls with input that can be generated by a wide variety of techniques.

The fuzzing framework also includes a VirtualBox Virtual Machine (VM) controller that allows the user to control VirtualBox VMs that run the fuzzer instances. The framework includes a logging component that logs the generated fuzz input and corresponding return values from the execution of the syscall.

5.2 User Requirements

The project's main target audience are information security researchers seeking a solution to fuzzing the kernel in order to fix holes or developers interested in assessing the general reliability of an application.

Through meetings and talks with the external supervisors, we have identified the following main requirements:

- capability to auto generate fuzzers based on the parsing of the kernels syscall headers;
- capability to fuzz OSX's XNU Kernel; and
- capability to monitor crashes in order to reproduce them for further analysis.

6 Requirement Specification

6.1 Physical Environment

Our assumptions about the physical environment when designing the system, is that it will be deployed in an isolated local area network (LAN) of one or two Intel machines running a Linux-based distribution. The benefits of such a setup comes with the advantages of developing on *nix-like operating systems, namely a highly configurable environment and excellent scripting support.

In addition, an isolated LAN network would lower the risk of external attacks while fuzzing is taking place and ensure a controlled environment for fuzzing. This is especially true when the fuzzer is extended to fuzz network-based systems such as listening services (e.g. ssh).

6.2 Core Functionality

The system should be able to parse XNU system call header files to identify all syscalls and the data types of the parameters. From this analysis, it should produce a logical representation, or a syscall profile, of these syscalls. Fuzz input generators should be then able to generate syscall fuzzing rules from the syscall profile improving the dynamics of the fuzzer.

The generator should be configurable and accommodate different fuzzing techniques. In the event that a crash was successfully cause, the system should capture and log faults or errors in order to reproduce them for further analysis.

6.3 Generation of Data

The generated fuzz input data might be of different data types to further expand the process with different and deeper fuzzing techniques. Integers, strings, binary data, and C structs can be generated and passed for execution in the calling mechanism. In return, the calling mechanism will call a function in the generator to pass on returned values after executing the syscall. This is so the generator may react dynamically.

6.4 Logging

Logging is an important part of post-fuzz analysis. In the event of a crash, we must be able to reproduce the crash and investigate the cause. Log entries should contain fuzz input data, return values from syscalls and signals like segmentation faults. The data type of return values is an integer. The data type of events is string. Log entries should be stored in a file or database for an indefinite amount of time.

6.5 Interfaces

With regards to user interactivity, the system will be controlled through a command line interface. A visual element is not imperative at this point as the system is intended to run on a sort of 'fire and forget' philosophy. However, a graphical interface to monitor the uptime of virtual machines would be useful.

Keeping the visual elements to a minimum has its advantages: a shell-only environment such as a headless server would run comfortably with a command line interface. For economical reasons, this set up is preferred.

6.6 Component Interaction

The system should be a client-server model with communication occuring between the host (client) and the server (virtual machines) through the network. Log entries will be sent from the virtual machines to a logger residing on the host or elsewhere, through the network.

6.7 Developer Requirements

In order to develop such a framework, there are certain system development requirements. Knowledge of the Python programming language as well as C is crucial. In addition, knowledge on fuzzing techniques and Linux/XNU kernels is crucial for creating fuzzer generators.

6.8 Resources

Development will be carried out by a four person team with varying skill and knowledge in the field. It will be done with the assistance of two Macbooks running OS X

and three laptops running Linux.

It is assumed that users will have at least one Intel machine running Linux and are familiar with using the operating system.

6.9 Software Requirements

The following software is required for usage of the framework:

- Python 2.6/2.7 (On both the physical and virtual machines.);
- VirtualBox with installed extensions and vboxapi python bindings; and
- A VirtualBox virtual machine running OS X.

6.10 Security

Machines should be run in a isolated local network. No further access controls are required but may be explored for further expansion. Access controls may be put in place in the future to prevent compromise of any of the fuzzing machines or the host controller machine.

6.11 Documentation and Quality Assurance

The team should provide full documentation on using the tool, fuzzing techniques, and details on any crashes found. In addition, documentation on how the system is built and how it works would better equip a user for expansion of the program through user defined generators, calling mechanisms, and event handling callbacks.

7 System Design

This section details how the application design has met the requirements specified.

7.1 System Architecture

The system architecture may be broken up into two parts: system hardware and system software.

7.1.1 System Hardware

One to two Linux Intel machines running in an isolated local area network.

7.1.2 System Software

- Python 2.6/2.7 (On physical and virtual machines);
- Python Twisted module;
- VirtualBox, extensions, and vboxapi python bindings (On physical machines); and
- OS X virtual machine.

The virtual machine should be accessible from the physical machine via ipv4 networking. There should be no firewall blocking or restricting any ports on both the physical and virtual machines.

7.1.3 System Hierarchy

There are two main entry points to the framework: *a*) ghost.py, the host controller; and *b*) and gfuzzd.py, the fuzzer daemon.

If one were to expand the hierarchy of these two separate but very related applications, we would obtain the following tree:

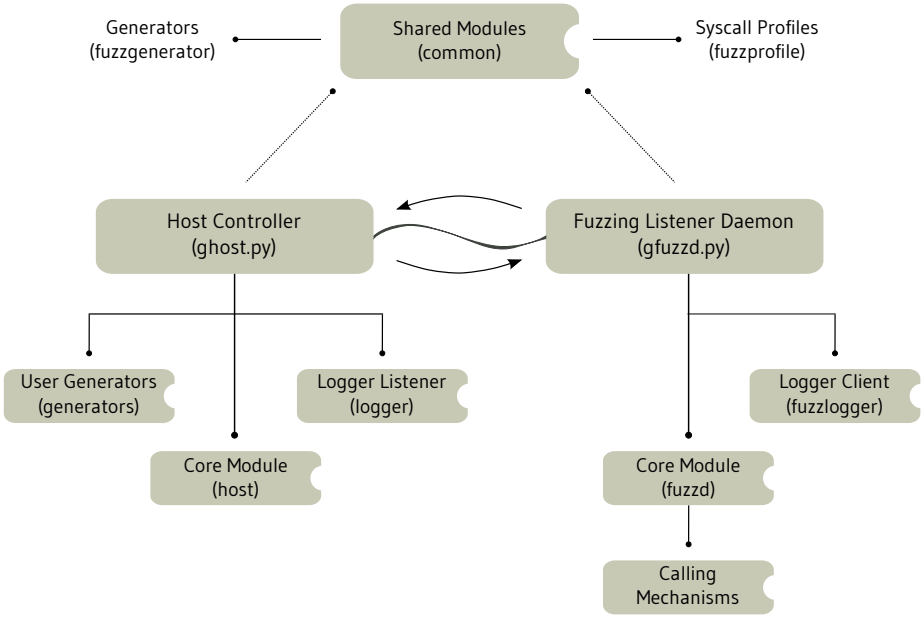

Figure 1: Class Hierarchy

7.2 Description of Major Functionality

A large feature of the system is the versatility of its main function: to fuzz. Hence, the designed system expands on this idea to include more than one platform, more than one target system, and more than one method of fuzzing. Figure 2 depicts the overview of two extremely versatile pluggable components of the main listener framework, gfuzzd.

7.2.1 Logging

ghost sends the settings for the location of the logger to a gfuzzd instance when the user issues a 'log <ip of logger><port>' command to a connected gfuzzd instance. gfuzzd will then send all future log entries to the user-defined address.

Each set of syscall data is logged in gfuzzd.

The logger writes the log entries sent by the various gfuzzd instances directly to a file. Each log entry starts with "HOST (<ip>:<port>)" and is delimited by a string "\nNEWSET\n".

The design of log entries is further explained in the Log Transport Design section.

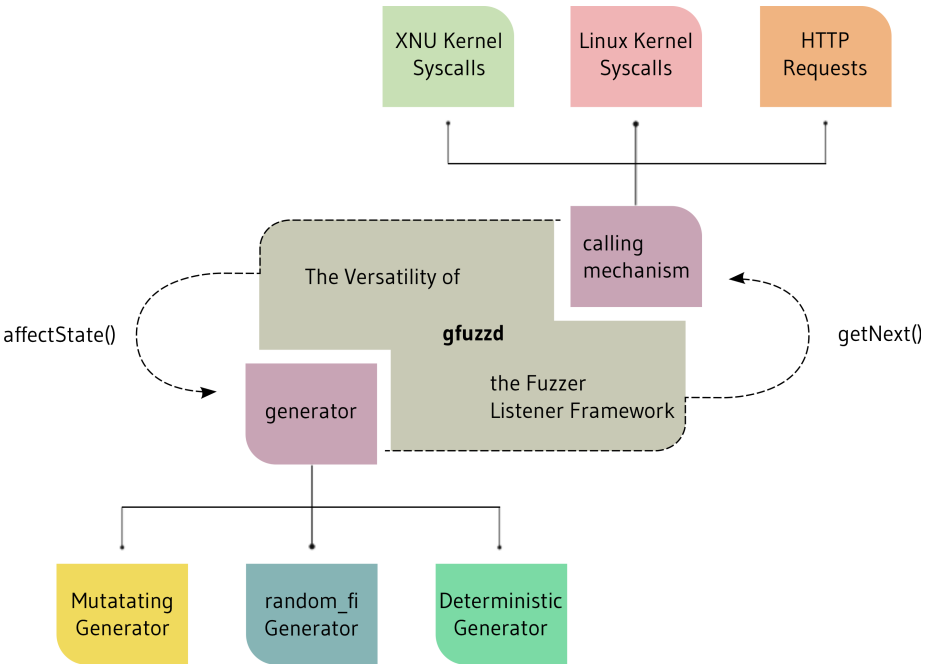

Figure 2: Versatility of gfuzzd

7.2.2 Parsing

The syscall.master file is parsed to obtain the BSD syscalls. The syscall_sw.c file is parsed to obtain the Mach syscalls. The parser is located in gfuzzd. Regular expressions are used to extract information when parsing.

For BSD syscalls, the function prototype, number, audit event, in files and comments are extracted. The function prototype for BSD syscalls includes, the return type, function name, syscall stub and arguments. Arguments contains the data type and name. All BSD syscalls are stored in a BSD syscall tree with branches based on conditionals defined by kernel compile time parameters.

For Mach syscalls, the number, name, argument count, munge_w and munge_d are extracted. Mach syscalls are stored in a collection.

The logical representation of syscalls are accessible to components such as the generator through a Syscall Profile, which is simply an object containing both the BSD syscall tree and the Mach syscall collection.

7.2.3 CallingMechanism

The CallingMechanism is an abstract class with a function that takes in a syscall number and some arguments. The user can implement different types of calling mechanisms, for example a HTTP CallingMechanism that makes a HTTP request.

The fuzzing framework has a built in XNUCallingMechanism which is used for calling XNU Kernel syscalls (BSD, Mach). It takes in a syscall number and some arguments which is then used to call the syscall. The program makes use of a built-in foreign function library in Python called ctypes. From the online python documentation on ctypes. http://docs.python.org/library/ctypes.html "It provides C compatible data types, and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python."

With ctypes, we can call syscalls in python code.

Loading C Library for use in Python

```
#import ctypes
from ctypes import *
#load OS X's C Library
libc = cdll.LoadLibrary(''libc.dylib'')
#OR
#load Linux's C Library
libc = cdll.LoadLibrary(  libc.so.6   )
```

Calling a Syscall with the loaded C Library

```
#libc.syscall calls the syscall with the given arguments
#return value of the syscall is stored in a variable, returnVal
returnVal = libc.syscall(syscallnumber, *args)
```

7.2.4 Generator

The generator is an abstract class which takes in a seed value and a profile. There are two abstract functions, getNext and affectState. Users can create a different generator for the generation of fuzz input data using various techniques.

The built-in generator, RandomFI, generates a set of data which includes random fuzz input data or pre-defined memory location (userland address, kernel address or unmapped address) and a random syscall number.

The getNext function generates a new set of data. The affectState function is designed to use the return value of the previous syscall that was called to affect the generation of fuzz input data.

7.2.5 Fuzzing

Fuzzing is done by calling the getNext function of the generator which will return a set of data. This set of data is described in the Calling Mechanism section above. Before using the calling mechanism to call the syscall, the set of data obtained from the generator is logged. After logging, the data is then used to call the syscall via the Calling Mechanism. The return value resulting from calling the syscall is logged as well. This return value can also used by the affectState function in the generator.

7.2.6 Signal Handling

Handles signals. When a signal is caught during fuzzing, it runs some code or ignores it so fuzzing can continue. If an interrupt signal is detected, the program will terminate. Or if an illegal instruction signal is received (SIGILL), the fuzzer can simply log that the signal was caught and carry on fuzzing unfettered.

7.2.7 Callbacks

Callbacks are extensively used in ghost. It allows users to define behaviour programmatically whenever certain events are triggered such as 'lost_connection', or 'data_received'. This allows for great flexibility when handling the gfuzzd instances.

For example, 'lost_connection' events might be interpreted as an indication that a kernel panic occurred. This may be inferred by checking that the virtual machine state is running and is awaiting a restart. Thus, one might handle that event in a callback with code that would restart the virtual machine and continue fuzzing.

7.2.8 Dumping VM Core/Memory

When a crash or kernel panic is detected, ghost will dump the virtualmachines memory using a function provided by the VirtualBox API, vboxapi. The resulting file is a virtualbox core dump. VirtualBox uses the 64-bit ELF format for its VM core files which can be inspected elfdump and GNU readelf or other similar utilities.

7.2.9 Dynamic Code Execution

gfuzzd is capable of loading arbitrary generators without having the class on the machine before hand. It uses a little python trickery and the fact that everything in python

is a first class object means that source code can be sent, uncompiled, over the network and then executed in a safe namespace. This makes maintainability of the system very easy.

7.2.10 Crash Detection

The host controller, ghost, will constantly monitor VMs it has connected to by pinging it at fixed intervals. If the VMs do not respond, it may have crashed.

7.2.11 Shell-like Interpreter for ghost

ghost is a command line application and is controlled by interacting with a shell. A list of commands follows:

* connect <ip of target><port>
  - Description: connects to the user-defined gfuzzd instance running on a virtual machine
* log <ip of logger><port>
  - Description: Sets the logger settings of the currently connected gfuzzd instance to the user-defined <ip of logger>and <port>
* fuzz
  - Description: The currently connected gfuzzd instance will begin the fuzzing process with the specified generator inputs called by the calling mechanism.
* stopfuzz
  - Description: The currently connected gfuzzd instance will stop fuzzing.
* dumpstate
  - Description: The currently connected gfuzzd isntance will dump its current state to ghost.
* loadgen
  - Description: Prepares the currently connected gfuzzd instance for fuzzing. Sends the gfuzzd instance the name of the generator to be used, a seed value and some python code.
* shutdown

  - Description: The currently connected gfuzzd instance will stop fuzzing and exit.

* help
  - Description: Prints the help screen.

7.2.12 VirtualBox Control

Using VirtualBox API, vboxapi, several functions can be carried out, such as turning on a machine and getting its memory dump files. The controller uses XPCOM (Cross Platform Component Object Model) and other VirtualBox interfaces to control almost all aspects of the virtual machine execution.

List of core functions integrated into the main program :

* dumpGuestCore
  - Description: Retrieve the machines memory and returns the machines core dump file.
* shutdownMachine
  - Description: Shutdowns a machine
* commandMachine
  - Description: Sends a command to the virtual machine that is able to alter and get the several states of the machine and its session. It also manages the machines interface session, preventing a collision and crashing the virtual machine. It also performs key functions such as capturing a snapshot and restoring one.
* activateMachine
  - Description: Activate a machine in either a GUI (Graphical User Interface) or VRDP (VirtualBox Remote Desktop Protocol) .

A web-based monitoring GUI has also been implemented and it uses AJAX (Asynchronous JavaScript and XML) and JSON (JavaScript Object Notation) to communicate with the host. It displays key information such as the virtual machines session status and its state.

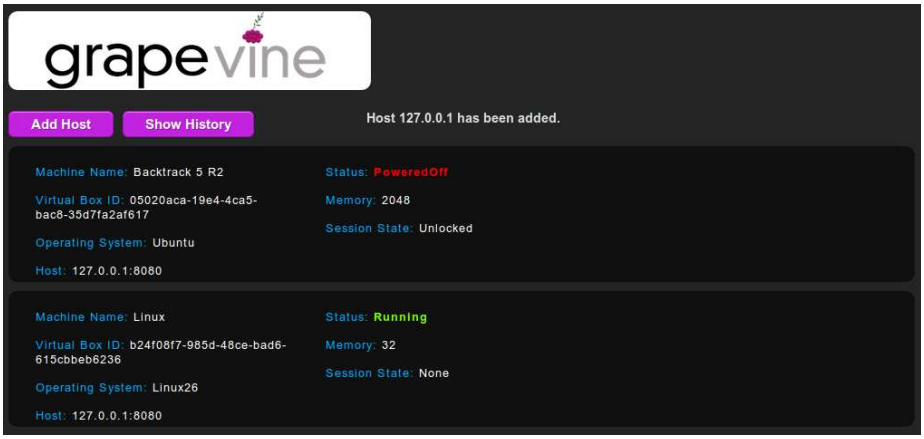

Figure 3: Web-Based Monitoring GUI

7.3 Log Transport Design

When a syscall is called, the syscall number and the generated arguments are pickled (serialised) individually and then packed into a JSON string.

The format is as follows:

```
{''syscallnr'': syscallnr,
 ''arg1'': arg[0],
 ''arg1+N'': arg[0+N]
}
```

Return values, commands, and events are packed into JSON strings. JSON string format of return value,

```
{''return_value'': str(retVal)}
```

JSON string format of events,

```
{''event'': event, ''urgency'': urgency}
```

JSON string format of command data,

```
{''data'': data,
 ''addr'':addr,
  ''param0'': param0,
 ''param1+N'': param1+N
}
```

---

8 Source Code Listing

8.1 gfuzzd

The fuzzing daemon, gfuzzd, is dependent on the following modules: common, fuzzd, and fuzzlogger.

8.1.1 fuzzd

The fuzzd module is responsible for the core functionality of gfuzzd. It provides two python files: gvcallingmechanism.py and gvfuzz.py.

gvcallingmechanism.py This file contains the base, default, and XNU kernel syscall calling mechanisms.

gvfuzz.py This file provides the binding to an address and listening for messages and acts as the core engine of gfuzzd.

8.1.2 fuzzlogger

The fuzzlogger module is responsible for the logging functionality of gfuzzd.

gvlogger.py This file provides the logging interface as well as handles the client side of the logging framework.

8.2 ghost

The host controller, ghost, is dependent on the following modules: common, generators, host, and logger.

8.2.1 generators

The generators folder contains the user defined generators. We provide a random $_f i.pyasasample.$

random_fi.py Sample generator for loading into a gfuzzd instance.

8.2.2 host

The host module contains the core functionality of the host controller.

gvhost.py This file provides two important core components of ghost. It exposes the Host Controller, which handles multithreaded host monitoring and management. It also provides the Host, which makes use of the callbacks passed into the Host Controller through its constructor.

8.2.3 logger

The logger module contains the logger listener.

gvloglistener.py This file is responsible for listening for log entries and writing them to a file so that events such as crashes, faults, commands, inout data can be retraced when needed to reproduce them.

8.3 common

The common module provides shared resources such as syscall profiles or generators to both ghost and gfuzzd.

8.3.1 fuzzgenerator

The fuzzgenerator module contains the base generator definitions.

gvgenerator.py This file contains the base abstract generator, default generator, and random_fi.

8.3.2 fuzzprofile

The fuzzprofile module contains the definitions for both mach and bsd syscalls as well as the parser to derive the syscall tables.

gvparser.py This file provides the utility functions to convert kernel source files to logical, object forms of the syscalls with usable information such as data types in a tree structure (bsd), or a list (mach).

gvsyscalls.py This file provides the syscall profiles (both mach and bsd) as well as the collections to hold them. Also contains the profile, which is a collection of collections.

9 Future Work

Grapevine is a framework, and the use of its fuzzing capabilities go beyond the XNU kernel. In fact, it goes beyond kernels, web applications and their servers can be fuzzed with very simple user additions to the program. Simply implement a HTTP calling mechanism and a generator that generates appropriate parameters for HTTP requests.

Replacing the current command line interfaces with something more graphical is easy as well. Simply obtain a programmatic instance of the corresponding core for gfuzzd and ghost. With the modular nature of the cores, they may be embedded into other applications.

In the future, more configurability would call for allowing for more choices than VirtualBox. Likewise, a VM-less option might also open up the use of the system to people who want a more minimalistic set up.

Perhaps a repository for generators, calling mechanisms, callbacks, and the embedded instances of the core itself may be put up for free use so that security research may be carried out.

With the open souce nature of the project, it is hoped that the Grapevine would grow through the power of the community.

References

[1] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing system virtual machines. In *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, pages 171–182, New York, NY, USA, 2010. ACM.

[2] nemo. Abusing mach on mac os x. *Uninformed*, May 2006.

[3] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Syngress, 2010.

[4] K. Y. Sim, F. C. Kuo, and R. Merkel. Fuzzing the out-of-memory killer on embedded linux: an adaptive random approach. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 387–392, New York, NY, USA, 2011. ACM.

[5] Mary R. Thompson and Linda R. Walmer. A programmers guide to the mach system calls. *Carnegie-Mellon University*, February 1988.

[6] TIELEI WANG, TAO WEI, GUOFEI GU, and WEI ZOU. Checksum-aware fuzzing combined with dynamic taint analysis and symbolic execution. *ACM Transactions on Information and System Security*, September 2011.

[7] Derek Wright and Todd M. Bezenek. Using fuzz to test the reliability of unix kernels. *University of WisconsinMadison*, December 1996.