AUTOMATED MAC OS X SYSCALL FUZZER GENERATOR GRAPEVINE USER GUIDE 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

This user guide details the hardware and software requirements in order to run Grapevine. It also describes how to obtain the source, unpack it, and deploy it to the virtual machine host and the virtual machines themselves. A walkthrough on how to operate gfuzzd and ghost is provided to further enhance the user's understanding of the system. Finally, a section for developers is included. This section details the exposed public member functions of the VirtualBox control and information packages as well as how to write custom generators, calling mechanisms, and callbacks for gfuzzd and ghost.

Contents

1 Introduction

This user guide explains the use of the grapevine fuzzing framework. The framework consists of two main components: *a*) ghost.py, the host controller; and *b*) and gfuzzd.py, the fuzzer daemon. gfuzzd runs in the virtual machine to be fuzzed while the host controller, ghost, runs in a physical machine. The host controller controls the fuzzing daemon via a custom protocol through UDP.

The program is written in python and designed to work with Virtualbox virtual machines. The host controller can run on Windows (with a few tweaks to the folders), Linux (verified), BSD or OSX, depending on the availability of Virtualbox extensions and vboxapi on the particular operating system platform.

2 Operating Environment

The operating environment requirements are defined piecewise in two parts: the hardware, and the software. In addition, the procedural steps for getting the application running are also listed in this section.

2.1 Hardware

The following hardware is recommended:

1. at least one Intel Machine running a Linux distribution;
2. entry-level routers with DHCP Server capability;
3. network switch (optional); and
4. ethernet cables.

2.2 Software

The following software is required as a dependency:

1. Python 2.6/2.7 (On Physical and Virtual Machines);
2. python-twisted package;
3. VirtualBox;
4. VirtualBox Extensions;
5. VirtualBox vboxapi; and
6. Mac OSX Virtualbox Virtual Machines.

2.3 Environment Setup Guide

The following is the procedure for setting up the software requirements, configuring the servers and application programs, and dependencies required for Grapevine to run.

2.3.1 Installing Python

On a up-to-date Linux computer, Python version 2.6 or 2.7 should be installed by default. If not, install it via the package management system of the distribution.

On a debian based operating system, run the command,

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

2.3.2 Installing python-twisted package

The python-twisted package is available via Ubuntu's package management system.

On a debian based operating system, run the command,

```
sudo apt-get install python-twisted
```

2.3.3 Virtualbox Installation Instructions

Virtualbox, Virtualbox extensions and the vboxapi is available via the Virtualbox website at https://www.virtualbox.org/. Install the listed in the following order: Virtualbox, Virtualbox extensions, vboxapi. Refer to the official Virtuabox installation documentation for more details on setting up vboxapi.

2.3.4 Setting up the OSX Virtual Machine

Obtain a copy of OS X of any version you choose. Install it into a Virtualbox virtual machine (VM). Follow the OS X installation helper to install OS X on the virtual machine. Once installation is complete, log in to the OS X VM and turn off the firewall. Steps to do this varies across different OS X versions. In OS X Lion, you may do this by clicking through Settings ->Security Privacy ->Firewall. It is also possible to find the firewall settings page by searching for "firewall" in Spotlight, the first search result is usually where the firewall settings is located at.

2.3.5 Additional settings for the Ubuntu Linux Physical Machine

Turn off the firewall in Ubuntu if it is not off by default. Set up the Fuzzing Framework program. Copy the folders, gfuzzd and common, including all subdirectories and files

into a folder in the OS X VM. Copy the folders, ghost and common, including all subdirectories and files into a directory in the home folder of the admin user in the Ubuntu Linux physical machine.

3 Installing Grapevine

This section details the process by which Grapevine is deployed.

3.1 Getting Grapevine

To obtain a copy of the latest current version of Grapevine, download the latest distribution from https://github.com/jergorn93/grapevine/tree/master/grapevine/dist. Next extract the files from the downloaded compressed archive.

```
tar xvfz <dist>.tar.gz
```

The resulting two folders, ghost and gfuzzd, are the grapevine applications.

3.2 Installing gfuzzd

gfuzzd is required to be installed on all the virtual machines. To install, simply copy the folder into a location of your choice within the host.

3.3 Installing ghost

gfuzzd is required to be installed on all the host of the virtual machines. To install, simply copy the folder into a location of your choice in the machine.

4 Operating Grapevine

This section describes how to operate both ghost.py and gfuzzd.py in a typical setup. As a rule, gfuzzd instances should be deployed before the host controller is.

4.1 Fuzzing Daemon, gfuzzd

To deploy the gfuzzd instance, ensure that port 10001 is unbound or replace the port number to listen on in gfuzzd.py to the desired listening port. Next, start the instance via:

```
./gfuzzd.py
or
python gfuzzd.py
```

An alternative to starting the gfuzzd instance manually would be to set it to act as an automatically started daemon. Please consult your operating system instructions on how to do this.

4.2 Host Controller, ghost

To start up the ghost application and receive an interactive command line shell, simply execute the ghost.py file.

```
./ghost.py
or
python ghost.py
```

To obtain a list of possible commands, input 'help' in the prompt.

```
./ghost.py
ghost> help
Grapevine Host Control alpha
Commands:
currenthost: displays IP and PORT of currently connected HOST
connect: prompts for new connection details
fuzz: start fuzzing in the connected host.
exit: exits the program.
stopfuzz: Stops fuzzing.
dumpstate: Dumps some gfuzzd internal variables.
loadgen: Loads a new generator.
help: Prints this help message.
ghost>
```

4.2.1 Connecting to a host

Before most of the commands may be used, ghost must be connected to at least one gfuzzd instance and have it selected as its current host.

```
./ghost.py
ghost> connect 127.0.0.1 10001
Connecting to 127.0.0.1:10001.
ghost>
Successfully connected to 127.0.0.1:10001.
ghost>
```

4.2.2 Obtaining the address of the current host

We may also check which host we are currently connected to with currenthost. Requires a currently selected host.

```
ghost> currenthost
We are currently connected to 127.0.0.1:10001
ghost>
```

4.2.3 Listing the statuses of hosts

We may check the statuses of the hosts that ghost is tracking through the use of hoststatus. It displays the address and the status of each host we are monitoring for events such as system crashes or disconnections.

```
ghost> connect 127.0.0.1 10002
Connecting to 127.0.0.1:10002.
ghost>
Successfully connected to 127.0.0.1:10002.
ghost> hoststatus
127.0.0.1:10001 - CONNECTED
127.0.0.1:10002 - CONNECTED
ghost>
```

4.2.4 Beginning the fuzz process

The fuzz command begins the fuzzing on the selected host. Requires a currently selected host.

```
ghost> fuzz
Fuzzing 127.0.0.1:10001.
ghost>
Notice: We have received data from 127.0.0.1:10001.
Fuzzing is turned on.
ghost>
```

4.2.5 Stopping the fuzz process

The stopfuzz command stops the fuzzing on the selected host. Requires a currently selected host.

```
ghost> stopfuzz
We stopped fuzzing 127.0.0.1:10001.
ghost>
Notice: We have received data from 127.0.0.1:10001.
Fuzzing is turned off.
ghost>
```

4.2.6 Debugging the gfuzzd

The dumpstate command dumps some internal variables of the fuzzd instance on the selected host. Requires a currently selected host.

```
ghost> dumpstate
ghost>
Notice: We have received data from 127.0.0.1:10001.
Dump of current state:

listen: <bound method FuzzD.listen of <fuzzd.gvfuzz.FuzzD instance
    at 0x14cdf38>>
__module__: fuzzd.gvfuzz
call_mech: <fuzzd.gvcallingmechanisms.TestCallingMechanism instance
    at 0x14cdea8>
generator: <common.fuzzgenerator.gvgenerator.DefaultGenerator
    instance at 0x14cdf80>
udp_ip: 0.0.0.0
_FuzzD__interrupt_handler: <bound method FuzzD.__interrupt_handler
    of <fuzzd.gvfuzz.FuzzD instance at 0x14cdf38>>
sock: <socket._socketobject object at 0x14cbbb0>
_FuzzD__fuzz: <bound method FuzzD.__fuzz of <fuzzd.gvfuzz.FuzzD
    instance at 0x14cdf38>>
udp_port: 10001
syscalls_profile: None
_FuzzD__sendback: <bound method FuzzD.__sendback of <fuzzd.gvfuzz.
    FuzzD instance at 0x14cdf38>>
_FuzzD__handle: <bound method FuzzD.__handle of <fuzzd.gvfuzz.FuzzD
    instance at 0x14cdf38>>
generator_namespace: Removed for practical reasons.
```

```
fuzzing: False
_FuzzD__safe_exit: <bound method FuzzD.__safe_exit of <fuzzd.gvfuzz.
    FuzzD instance at 0x14cdf38>>
logger: <fuzzlogger.gvlogger.LoggerClient instance at 0x14cdcb0>
_FuzzD__sig_handler: <bound method FuzzD.__sig_handler of <fuzzd.
    gvfuzz.FuzzD instance at 0x14cdf38>>
__doc__: None
__init__: <bound method FuzzD.__init__ of <fuzzd.gvfuzz.FuzzD
    instance at 0x14cdf38>>
fuzz_thread: <Thread(fuzz, stopped 140357644777216)>

ghost>
```

4.2.7 Loading a new generator

The loadgen command loads a new generator from the ghost to the selected host. Requires a currently selected host.

```
ghost> loadgen
Generator Name: RandomFI
Seed: 0
Generator File: generators/random_fi.py
ghost>
Notice: We have received data from 127.0.0.1:10001.
generator RandomFI loaded
ghost>
```

5 Extending Grapevine

Grapevine is an application geared towards security researchers which we assume have the required programming knowledge to modify simple source code to their needs. This section lists the relevant application programming interfaces (APIs) and describes how to write code to extend Grapevine's functionality.

5.1 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. The following are public member functions found in VBControl.py, a utility module.

5.1.1 Activating Machines

void Controller::activateMachine(string MachineName, boolean guiMode)

Spawns a new process that will activate a virtual machine by name and in either a VRDP (VirtualBox Remote Desktop Protocol) or GUI (Graphical User Interface) . It will also obtain a shared lock on the machine for the calling session.

5.1.2 Shutting Down Machines

void Controller::shutdownMachine(string machineID)

Shuts down a virtual machine by machineID and its progress is being tracked as well. After operation is completed, the machine will go into a PoweredOff state.

5.1.3 Command Machines

boolean Controller::commandMachine(string machineID, string command, string arguments)

Takes in a command string that translates into several action procedures. For example, a 'takeSnapshot' command will initiate a procedure to capture the snapshot of the virtual machine, the progress of execution will also be tracked.

5.1.4 Dumping Guest Core

void Controller::dumpGuestCore(string machineID, string filepath)

Takes a core dump of the target machine, which is identified by its unique machine ID. Contents of the core dump will be written to the filepath specified by the user.

5.2 VirtualBox Info

The VirtualBox Info module contains methods that allow for information to be programmatically retrieved.

5.2.1 Checking if a machine is alive

boolean Information::checkIfAlive(string IPAddress)

Checks if a particular machine is alive via pending for a ping reply. Returns true if alive and false if otherwise.

5.2.2 Obtaining information

string Information::jsonifyMachinesData()

Enumerates all virtual machines on the host machine, gathering key informations such as machine states and session states, and the data is returned in a JSON format. This data is used primarily to update the programs web-based user interface.

5.2.3 Getting crashed machines

string[] Information::getCrashedMachines()

Compiles an array of machines that have ceased operation, either in ’Paused’ or ’Aborted’, it returns an array of machine names that crashed, or “0” if there are no crashed machines. This data is used primarily to update the programs web-based user interface.

5.2.4 Getting all live machines

string[] Information::getAllLiveMachinesID()

Compiles an array of machines that are still in operation. This data is used primarily to update the programs web-based user interface.

5.3 Calling Mechanisms

Calling mechanisms are a logical layer in the gfuzzd system. It is the point where the generated data is placed in the input, i.e. executed, requested, etc, and where the output is received. In order to adapt Grapevine for a particular system, the calling mechanism needs to be developed for that system (and then the generator to provide data to the calling mechanism).

To begin, we must look at the base class for a CallingMechanism in gfuzzd/fuzzd/gvcallingmechanism.py.

```
class CallingMechanism:

    def call(self, syscall_number, *args):
        raise Exception(''Unimplemented call() in abstract
            CallingMechanism class.'')
```

There is only one method to override when implementing the CallingMechanism. The key caveat to remember is that the return value should always be returned in this function. This is so information may be utilised by the generator.

As an example, we provide the XNUCallingMechanism. It executes syscalls in a Mac OS X environment and passes the return values back to its generator.

```
from ctypes import *
libc = cdll.LoadLibrary(''libc.dylib'')
class XNUCallingMechanism(CallingMechanism):

    def call(self, syscall_number, *args):
        "''Takes in a syscall number and a list or tuple of
            arguments"''
        returnVal = libc.syscall(syscall_number, *args)
        return returnVal
```

5.4 Generators

Calling Mechanisms are nothing without data input. Generators provide this data to the Calling Mechanism, which in turn executes its call() function and returns the results back to the generator. Thus, it is logical to implement the generator as a state machine that keeps a memory of its state and acts and reacts accordingly. This allows for very complex rulesets to be introduced making it possible to attach semantics to the input.

To begin with generators, we take a look at the Generator base class at common/fuzzgenerator/gvgenerator.py.

```
class Generator:
    profile = None
    state = None
    seed = None

    # Initialise the class with a seed value to kick start the
        generator.
    def __init__(self, profile, seed):
        self.profile
        self.seed = seed

    # Call getNext() to obtain the next set of inputs.
```

```
    # Should always return a list of values (Data should be expected
         by the syscall
    # calling mechanism).
    def getNext(self):
        raise Exception(''Unimplemented getNext() in template
            Generator class.'')

    # Call this function with a list of return values after fuzz
        input is run
    # to affect the state of the Generator in order to dynamically
        generate
    # input based on complex rules.
    # data is always a list of values.
    def affectState(self, data):
        raise Exception(''Unimplemented affectState() in template
            Generator class.'')
```

There are a couple of items of interest in that definition. The first is the constructor. It takes in a profile and a seed. Profiles, in the context of an XNU kernel fuzzer, are object representations of syscalls (mach and bsd).

However, essentially, the profile is simply a method of passing a structure describing acceptable input data. A seed is an initial starting value to place the state machine in its initial or starting state.

The second is the getNext() function. It yields input data to the Calling Mechanism.

The third is the affectState(data) function. The Calling Mechanism's return value will be passed back into the generator through this function in order to produce a stateful effect to the state machine of the generator.

The following is an implementation of a test generator that supplies contrived data in order to debug.

```
class DefaultGenerator(Generator):

    state = 0

    def getNext(self):
        return [0, 1, 2, 3, 4, 5, 6, 7, 8]

    def affectState(self, data):
        self.state = self.state + 1
        print ''Generator got %s back. (State %d)'' % (data, self.
            state)
```

5.5 Callbacks

The power of ghost comes from its callbacks. Behaviour of the host controller is completely controlled by callbacks that are user defined. Callbacks are attached to events in advance of its triggering. A list of events supported by the current version of Grapevine are as follows:

1. unable_to_connect
2. lost_connection
3. data_received
4. reconnected
5. connected

The internal implementation of the callbacks include the following:

```
...
            self.callback_set = {
                'unable_to_connect': self.
                    __default_unable_to_connect_callback,
                'lost_connection':  self.
                    __default_lost_connection_callback,
                'data_received':  self.
                    __default_data_received_callback,
                'reconnected':  self.__default_reconnected_callback,
                'connected': self.__default_connected_callback,
                }
...
    @staticmethod
    def __default_lost_connection_callback(addr):
        print ''Not implemented.''

    @staticmethod
    def __default_unable_to_connect_callback(addr):
        print ''Advice: Not implememnted.''

    @staticmethod
    def __default_reconnected_callback(addr):
        print ''Advice: Reconnected event is not handled.''

    @staticmethod
    def __default_connected_callback(addr):
        print ''Advice: Connected event is not handled.''

    @staticmethod
    def __default_data_received_callback(addr, data):
        print ''\nData is Received:\n'', data
```

Points to notice are: callbacks are stored in a dictionary in the Host object. The key is the corresponding event in a string form and the value is a reference to the function that handles the callback. Callbacks may take variable arguments because the internal mechanism for calling the callback supports it. The following shows the private callback caller as well as an example of a callback being called.

```
...
    def __call_callback(self, event, *args):
        self.callback_set[event](*args)
...
        self.__call_callback('data_received', (self.ip, self.port),
            data)
```

The following is an example of how to implement the callbacks and how to register them.

```
...
def __reconnected_callback(addr):
    sys.stdout.write(''\nWe regained our connection to %s:%d.\nghost
        > '' % addr)
    sys.stdout.flush()

def __connected_callback(addr):
    sys.stdout.write(''\nSuccessfully connected to %s:%d.\nghost> ''
         % addr)
    sys.stdout.flush()

def __data_received_callback(addr, data):
    no_print = ['pong']
    if not data.startswith(''hello'') and not data.startswith(''bye
        '') and not data in no_print:
        sys.stdout.write(''\nNotice: We have received data from %s:%
            d.\n'' % addr)
        sys.stdout.write(''%s\nghost> '' % data)
        sys.stdout.flush()

def __unable_to_connect_callback(addr):
    sys.stdout.write(''\nError: We were unable to connect to %s:%d.
        Removing host from tracked hosts.\nghost> '' % addr)
    ghost.remove_host(addr[0], addr[1])
    sys.stdout.flush()
...
    callbacks = {
        'unable_to_connect': __unable_to_connect_callback,
        'lost_connection': __lost_connection_callback,
        'reconnected': __reconnected_callback,
        'data_received': __data_received_callback,
        'connected': __connected_callback,
        }
    ghost = HostsController(callbacks=callbacks)
```
