Security Focused Continuous Integration (SFCI) Documentation

Release 1.0

Getting Started Guide

Michael Lescisin, Qusay H. Mahmoud

January 2017

CONTENTS

CHAPTER

ONE

GETTING STARTED

1.1 Audience

The intended audience for this tool is software project managers or lead developers who wish to move secure software development closer to the developers by providing them with constant feedback of potential vulnerabilities which they might be introducing into a project being developed.

1.2 Platforms Supported

Our Continuous Integration tool relies on the host OS being able to run the QEMU virtual machine hypervisor and communicate with the host OS via TAP interface. Controlling the virtual machine is managed by BASH shell scripts. Web interface generation is done with Python/Jinja2.

Any platform that meets the above three requirements should be capable of running our tool. For best results, we recommend a popular Linux distribution such as Debian or Ubuntu.

1.3 Preparing the Environment (Tools Required)

Ensure that the following packages are installed on your system. They are available in the package repositories of most popular Linux distributions. In Ubuntu, they can be installed via *sudo apt-get install PACKAGE-NAME*.

- qemu
- python
- python-jinja2
- git
- tar
- genisoimage
- uml-utilities
- screen

After this, a TAP interface *(tap0)* needs to be created for communication between the host and the VM. This interface needs to be assigned the IP address *169.254.254.1*. The following commands will set up this interface.

```
sudo tunctl -u $(whoami)
sudo ifconfig tap0 169.254.254.1
```

The VM image for executing the test cases needs to be created. We have provided an Ansible playbook for configuring a standard Debian Jessie install with all the tools required for being the test environment for our framework.

When creating the base Debian image, the default user should be named *user*. The virtual disk file should be named *graphical_debian_testdrive.qcow*. To build the VM image, start the base image in QEMU with the command

```
qemu-system-i386 -hda graphical_debian_testdrive.qcow -net nic,\
        model=virtio -net tap,ifname=tap0,script=no,downscript=no
```

and configure all network parameters (name server, default gateway, etc.) required for accessing the Internet from within the VM. You will also need to instruct the host to allow traffic from the VM to be directed to the Internet. The following command will do so

```
sudo iptables -t nat -A POSTROUTING -s 169.254.254.2 -o wlan0 -j MASQUERADE
```

assuming *wlan0* is your network adapter connected to the Internet.

Next, the user, *user*, needs to be added to the *sudo* group. To do this, login to the VM as root and run the command

```
adduser user sudo
```

Your host machine now needs public key authentication based SSH access to the virtual machine. To enable this, login to the virtual machine as *user* and run the command

```
mkdir .ssh
```

After this, on the host machine enter the directory *~/.ssh* and run the command

```
scp id_rsa.pub user@169.254.254.2:.ssh/authorized_keys2
```

For more information about enabling SSH public key authentication please see http://www.cyberciti.biz/tips/ssh-public-key-based-authentication-how-to.html .

You can now configure the image with Ansible. To do so, enter the directory *Ansible_ImageBuilder* and run the command

```
ansible-playbook --user=user --ask-sudo-pass build_image_playbook.yml
```

Once the image is built it should be placed (if not there already) in the root directory of our tool (ie. in the same folder as *find_bugs.json*).

1.4 Installation

One copy of our tool is required for each software project which you wish to perform continuous integration security testing on.

1. Within the root directory of our tool, clone the git repository of the project which you wish to test to *LogParsing/ProjectGitRepo*.
2. Place your test case shell scripts under *LogParsing/ProjectTests*.
3. Modify the file *LogParsing/do_all_tests.sh* so that all tests under *LogParsing/ProjectTests* are called and their results are returned from the VM back to the host.
4. Modify the file *rx_results.sh* so that all results are retrieved from the VM in proper order.

1.5 Developing Test Cases

1.5.1 Architecture

All tests conducted against a project are executed in a snapshot VM. The snapshot functionality means that the VM filesystem is the exact same every time the VM is powered on. The default VM image is Debian Jessie i386. Everything inside *LogParsing* and its sub-folders will be copied to */home/user/CodeTest_Environment* when the VM is started. The file *do_all_tests.sh* will be automatically executed when the VM starts. The VM has an Ethernet interface *eth0* assigned the IP address *169.254.254.2*. This is connected to the *tap0* interface on the host. *tap0* on the host is assigned the IP address *169.254.254.1*. This communication channel is used primarily for sending test results from the VM to the host.

There are several configuration files which may need to be modified for creating test cases. The following explains the purpose of each file. All listed files are located relative to the root directory of our tool. For example, if the tool has been downloaded to *~/CI_TOOL* then *find_bugs.json* will be located at *~/CI_TOOL/find_bugs.json* and *LogParsing/do_all_tests.sh* will be located at *~/CI_TOOL/LogParsing/do_all_tests.sh*.

1.5.2 find_bugs.json

This file contains a JSON encoded list of types of bugs to search for. The available types of bugs are:

- BufferOverflow
- MemoryLeaks
- DataRaceConditions
- DoubleFree
- UseAfterFree
- SqlInjection
- CommandInjection
- PathTraversal
- XSS

1.5.3 LogParsing/do_all_tests.sh

This file is automatically called by the VM Guest OS and is responsible for calling all other test cases, sending the results of the tests back to the host system, and optionally configuring the environment needed by all tests (eg. starting servers).

1.5.4 rx_results.sh

This file defines how the test results will be retrieved from the VM. A very simple method of doing this is to start a simple, single-threaded, netcat server for each file being received and have *do_all_tests.sh* connect to this server and send the files in the same order which they are received. This file needs to be modified so that only the files which are relevant to the tests are sent. For example, if an application is not tested for use-after-free bugs, the file *use_after_free.json* does not need to be sent.

1.5.5 LogParsing/ProjectTests/<TEST-NAME>

Each test case should be designed to look for a specific vulnerability (eg. command injection) within the project being tested. After the test is executed the results should be added, not overwritten, to a JSON file containing information about the type of vulnerability. For example, if a test case searches for path traversal vulnerabilities, a vulnerable URL should be added to the list of vulnerable URLs without removing any previously existing ones. Templates are provided for using popular dynamic analysis and penetration testing tools within the test case. The supported tools are:

- Valgrind Memcheck
- Valgrind Helgrind
- AddressSanitizer
- sqlmap
- Commix
- XSS Me
- DotDotPwn

1.6 Sample Run

The following is an example configuration scenario for testing a project for a certain type of vulnerability.

1.6.1 Writing a simple test case

The following is a simple program written in C which calculates a sequence of 64 numbers based on any seed number greater than 200. It has no practical purpose but it does demonstrate a potential use-after-free vulnerability. Its code should be placed under *LogParsing/ProjectGitRepo/math_program.c*

```c
#include <stdio.h>

int main(int argc, char **argv)
{
        int x;
        int y;
        printf("Value of first argument was %s\n",argv[1]);
        x = atoi(argv[1]);

        int *my_buffer = malloc(64 * sizeof(int));

        if (x > 400)
        {
                int i;
                for (i = 0; i < 64; i++)
                {
                        my_buffer[i] = x + i;
                }

                for (i = 0; i < 16; i++)
                {
                        my_buffer[i] =  my_buffer[i] + my_buffer[63 - i] * 3;
                }

                //Output
                for (i = 0; i < 64; i++)
                {
                        printf("-%d-",my_buffer[i]);
                }
                printf("\n");

                free(my_buffer);
        }

        //Should have been 'else if'
        if (x > 200)
        {
                int i;
                for (i = 0; i < 64; i++)
                {
                        my_buffer[i] = x - i;
                }

                for (i = 0; i < 16; i++)
                {
                        my_buffer[i] =  3*my_buffer[i] + my_buffer[63 - i];
                }

                //Output
                for (i = 0; i < 64; i++)
                {
                        printf("-%d-",my_buffer[i]);
                }
                printf("\n");

                free(my_buffer);
        }

        return 0;
}
```

This directory should be a git repository and *math_program.c* should be included in that repository. To do that, enter the *LogParsing/ProjectGitRepo* directory and execute the following commands:

```
git init #Create the git repository
git add math_program.c #Add math_program.c to the staging area
git commit -m "Initial commit" #Commit math_program.c with a message
```

If you have never used git before you will need to set a username and email. Instructions to do that can be found at https://help.github.com/articles/set-up-git/

Let us first write the test case files to use AddressSanitizer to search for use-after-free vulnerabilities. Let us start with the *AddressSanitizer_UseAfterFree.sh* template. We first need to change the line

```
clang -fsanitize=address -g example.c
```

to

```
clang -fsanitize=address -g math_program.c
```

We also need to change the variables

```
LOGFILE="asan_log.log"
CFILE="ProjectGitRepo/example.c"
```

```
TESTFILE="ProjectTests/uaf_testcase_001.sh"
JSONFILE="json/use_after_free.json"
```

to

```
LOGFILE="asan_log.log"
CFILE="ProjectGitRepo/math_program.c"
TESTFILE="ProjectTests/test-001-use_after_free.sh"
JSONFILE="json/use_after_free.json"
```

We will create three copies of this file and name them, *test-001-use_after_free.sh*, *test-002-use_after_free.sh*, and *test-003-use_after_free.sh*. For each file, the line

```
TESTFILE="ProjectTests/test-001-use_after_free.sh"
```

will be changed to

```
TESTFILE="ProjectTests/TESTCASE_NAME"
```

replacing TESTCASE_NAME with the file's name.

These files will be placed under *LogParsing/ProjectTests*. We will test to see how the program reacts to a seed value of 30, a seed value of 300, and a seed value of 3000. We will do this by changing the line

```
./ProjectGitRepo/a.out 2>asan_log.log
```

to

```
./ProjectGitRepo/a.out 30 2>asan_log.log
```

and saving the file as *test-001-use_after_free.sh*.

In *test-002-use_after_free.sh* and *test-003-use_after_free.sh* this line will be changed to

```
./ProjectGitRepo/a.out 300 2>asan_log.log
```

and

```
./ProjectGitRepo/a.out 3000 2>asan_log.log
```

respectively. *test-001-use_after_free.sh* should look as follows.

```
#         This program should be executed as ./ProjectTests/test-001-use_after_free.sh
#         not as ./test-001-use_after_free.sh


#Display line numbers for debugging output
export ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
export ASAN_OPTIONS=symbolize=1

#Get current directory
ROOT_DIR=$(pwd)

#Switch to program source code directory
cd ProjectGitRepo

#Compile the program with Clang (AddressSanitizer)
clang -fsanitize=address -g math_program.c

#Switch back to original directory
cd $ROOT_DIR
```

```
#Test for use-after-free
./ProjectGitRepo/a.out 30 2>asan_log.log

#See if there were any errors
if [ -s asan_log.log ]
then
        #Parse AddressSanitizer log for use-after-free errors and JSONify
        LOGFILE="asan_log.log"
        CFILE="ProjectGitRepo/math_program.c"
        TESTFILE="ProjectTests/test-001-use_after_free.sh"
        JSONFILE="json/use_after_free.json"
        python additive_log_parse.py $LOGFILE $CFILE $TESTFILE $JSONFILE
else
        echo "ERROR_FREE" > asan_log.log
fi
```

Now, we must modify the script *LogParsing/do_all_tests.sh* so that our test case is executed and the results are served back to the host. The following is an example of a test case which will do such.

```
#Run the test cases
./ProjectTests/test-001-use_after_free.sh
./ProjectTests/test-002-use_after_free.sh
./ProjectTests/test-003-use_after_free.sh

#After the tests are done, serve the JSON file
#(listing the vulnerabilities) with netcat

nc 169.254.254.1 9000 -q 0 < json/use_after_free.json > /dev/null
```

Now, we must modify the script *rx_results.sh* to tell the host process how to receive test results from the VM. The method used in this example is the host listens for TCP connections on *169.254.254.1:9000* and reads test results over that connection. The following file implements this.

```
#!/bin/bash

#Switch to the test results directory
cd TestResults

#Receive the results in proper order
nc -l -p 9000 -s 169.254.254.1 -q 0 > use_after_free.json
```

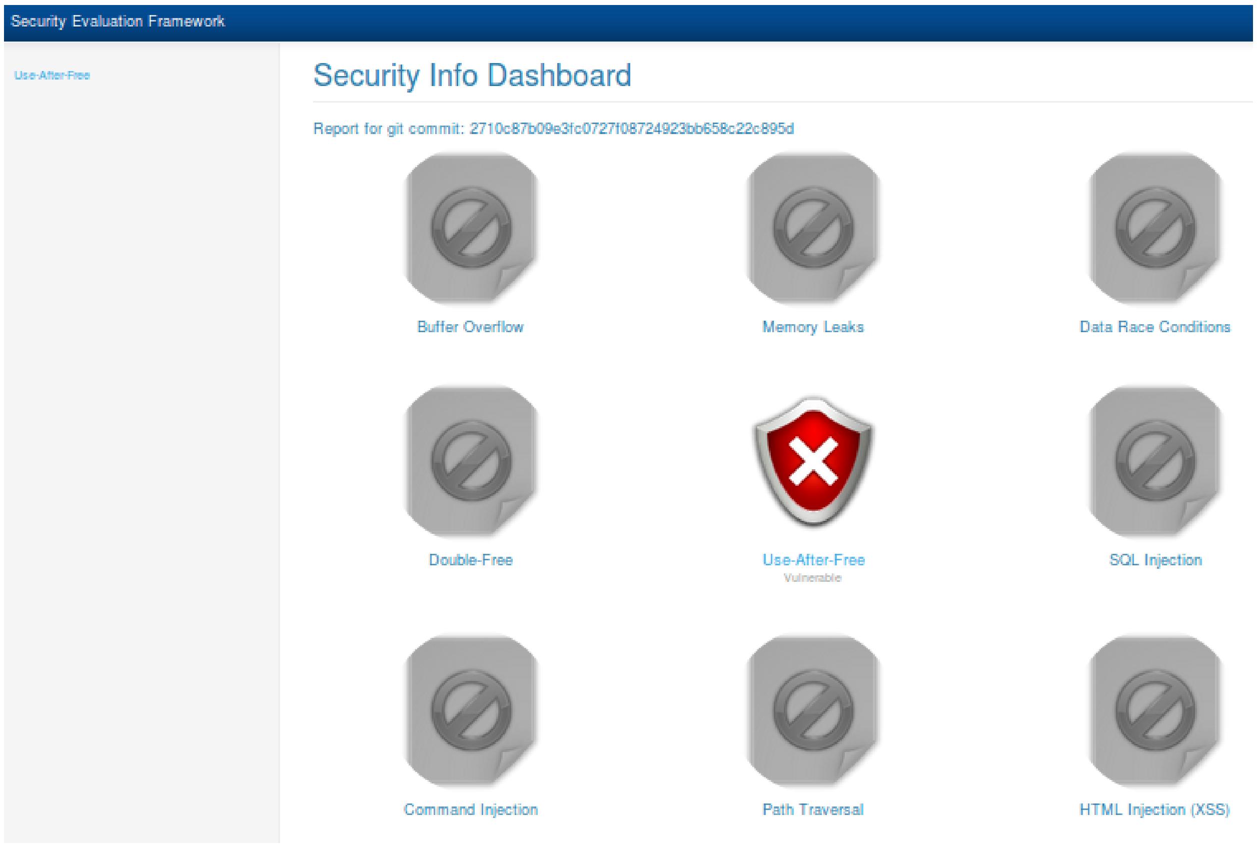
Now, we must tell our tool that we are only looking for Use-After-Free bugs. Edit the file *find_bugs.json* and set its contents to

```
["UseAfterFree"]
```

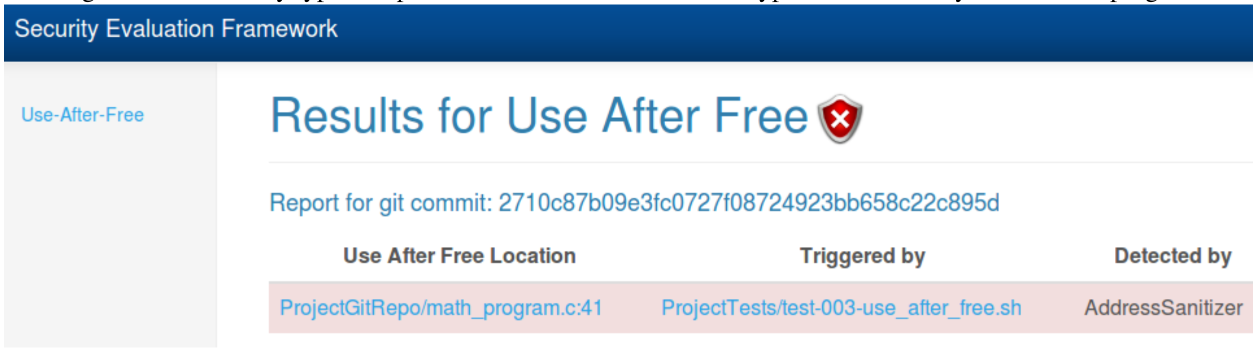
It is now time to run the test cases. Enter the root directory for the security tool and execute the command *./run_tests.sh*. Once the command is finished executing, open the file *rendered/main.html* in your browser to see the test results.

1.7 Interpreting the Results

After all tests are complete, the dashboard interface can be accessed at *rendered/main.html*. It will provide a general overview of the state of each type of vulnerability searched for in the program.



Clicking on a vulnerability type will provide further details about that type of vulnerability found in the program.

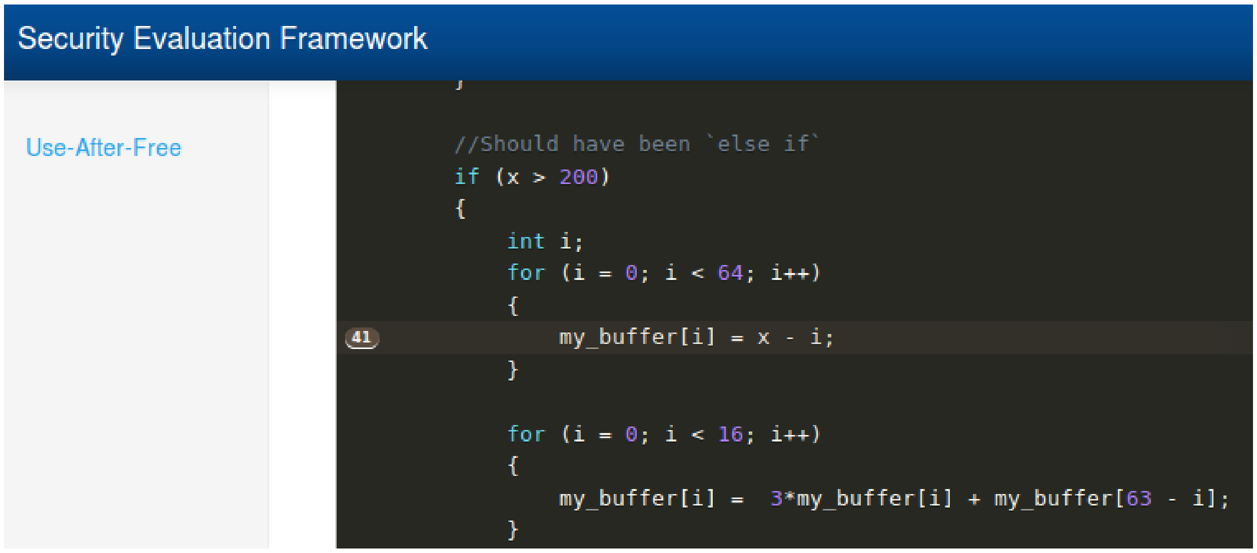


Results for Use After Free

Report for git commit: 2710c87b09e3fc0727f08724923bb658c22c895d

Use After Free Location	Triggered by	Detected by
ProjectGitRepo/math_program.c:41	ProjectTests/test-003-use_after_free.sh	AddressSanitizer

This page will then link to the test case file which triggered the vulnerability and possibly the line of source code where the vulnerability occurred.



```
    //Should have been `else if`
    if (x > 200)
    {
        int i;
        for (i = 0; i < 64; i++)
        {
            my_buffer[i] = x - i;
        }

        for (i = 0; i < 16; i++)
        {
            my_buffer[i] =  3*my_buffer[i] + my_buffer[63 - i];
        }
```

1.8 Adding Support for Another Tool

This section will describe the procedure for adding support for another tool to be used during the test cases of your application.

1.8.1 American Fuzzy Lop

This example will describe how to add support for American Fuzzy Lop - a genetic algorithm based fuzzing tool.

First, it is necessary to install the American Fuzzy Lop (afl) tool in the VM. The source code for afl can be downloaded from http://lcamtuf.coredump.cx/afl/releases/afl-latest.tgz. Compiling the source code is simple. Extract the archive with the command *tar -xvf afl-latest.tgz*, enter the source code directory and execute the *make* command. We will implement this by placing *afl-latest.tgz* under *LogParsing/AmericanFuzzyLop*. We will now modify *do_all_tests.sh* so that it installs afl every time the VM is started. To do this we will add the following lines to the beginning of *do_all_tests.sh*.

```
#Install American Fuzzy Lop (afl)
mkdir ~/AFL
cp AmericanFuzzyLop/afl-latest.tgz ~/AFL
ROOT_DIR=$(pwd)
cd ~/AFL
tar -xvf afl-latest.tgz
cd afl-2.19b
make
cd $ROOT_DIR
```

Now we will create *test-afl.sh*, the actual test case which will test a program for use-after-free bugs. We will use *TEMPLATES/AddressSanitizer_UseAfterFree.sh* as our starting file. Before adding anything to this file remove the lines

```
#Display line numbers for debugging output
export ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
export ASAN_OPTIONS=symbolize=1
```

from the beginning of the file as they cause *afl-fuzz* to malfunction.

The first step to fuzz testing a program with afl is to compile it with an afl compiler so to produce an instrumented binary. To do this we will add the line

```
#Compile the program with afl-gcc
~/AFL/afl-2.19b/afl-gcc math_program.c -o math_program
```

below the line

```
cd ProjectGitRepo
```

The second step to fuzz testing a program with afl is to create several "seed" test cases. These test cases are simple valid inputs for the program being tested. The following commands (to be placed in *test-afl.sh* under the *afl-gcc* line) will implement this.

```
#Create the testcase_dir directory
mkdir testcase_dir

#Seed afl with input testcases which are known to work
echo 301 > testcase_dir/testcase_001
echo 302 > testcase_dir/testcase_002
echo 303 > testcase_dir/testcase_003
echo 304 > testcase_dir/testcase_004
echo 305 > testcase_dir/testcase_005
```

The third step is to create the output directory for afl. This directory will contain the inputs which crash the program being tested as well as information about tests which have been executed. Adding the following line implements this.

```
#Create the findings directory
mkdir findings_dir
```

It is now time to begin the actual fuzzing. We must call afl-fuzz to run the fuzz tests and set the environment variable **AFL_NO_AFFINITY** because our VM only has 1 emulated CPU. Adding the following lines implements this.

```
#Begin afl fuzzing
export AFL_NO_AFFINITY=1
~/AFL/afl-2.19b/afl-fuzz -i testcase_dir -o findings_dir -- ./math_program &
```

Because we want afl to run only for a limited amount of time we must terminate its process after a given number of cycles has elapsed. In this example we chose 5 cycles. Adding the following lines implements this.

```
#Wait for 5 cycles
AFL_PID=$!

while [ ! -e findings_dir/fuzzer_stats ]
do
        echo "Waiting for fuzzer_stats file"
        sleep 5
done

AFL_CYCLES=$(cat findings_dir/fuzzer_stats | grep cycles_done | cut -d ':' -f2 | cut -d ' ' -f2)

while [ $AFL_CYCLES != 5 ]
do
        AFL_CYCLES=$(cat findings_dir/fuzzer_stats | grep cycles_done \
                | cut -d ':' -f2 | cut -d ' ' -f2)
done

#Stop afl fuzzing
kill -SIGINT $AFL_PID
```

Now, we will compile the program being tested with *clang (AddressSanitizer)*, execute the program with inputs from afl, and check if use-after-free bugs occurred. Adding the following code implements this.

```
#Display line numbers for debugging output
export ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
export ASAN_OPTIONS=symbolize=1

#Compile the program with Clang (AddressSanitizer)
clang -fsanitize=address -g math_program.c


#Switch back to original directory
cd $ROOT_DIR

#Execute all test cases from afl which caused crashes and look for
#use-after-free errors

rm ProjectGitRepo/findings_dir/crashes/README.txt

for f in ProjectGitRepo/findings_dir/crashes/*
do
        cat $f | ./ProjectGitRepo/a.out 2>asan_log.log

        #See if there were any errors
        if [ -s asan_log.log ]
        then
                #Parse AddressSanitizer log for use-after-free errors and JSONify
                LOGFILE="asan_log.log"
                CFILE="ProjectGitRepo/math_program.c"
                TESTFILE="ProjectTests/test-afl.sh"
                JSONFILE="json/use_after_free.json"
                python additive_log_parse.py $LOGFILE $CFILE $TESTFILE $JSONFILE
        else
                echo "ERROR_FREE" > asan_log.log
        fi
done
```

The file *LogParsing/ProjectTests/test-afl.sh* should now look as follows.

```
#        This program should be executed as ./ProjectTests/test-afl.sh
#        not as ./test-afl.sh

#This test case uses American Fuzzy Lop (afl) to generate inputs which
#crash the program.


#Get current directory
ROOT_DIR=$(pwd)

#Switch to program source code directory
cd ProjectGitRepo

#Compile the program with afl-gcc
~/AFL/afl-2.19b/afl-gcc math_program.c -o math_program

#Create the testcase_dir directory
mkdir testcase_dir

#Seed afl with input testcases which are known to work
```

```
echo 301 > testcase_dir/testcase_001
echo 302 > testcase_dir/testcase_002
echo 303 > testcase_dir/testcase_003
echo 304 > testcase_dir/testcase_004
echo 305 > testcase_dir/testcase_005

#Create the findings directory
mkdir findings_dir

#Begin afl fuzzing
export AFL_NO_AFFINITY=1
~/AFL/afl-2.19b/afl-fuzz -i testcase_dir -o findings_dir -- ./math_program &

#Wait for 5 cycles
AFL_PID=$!

while [ ! -e findings_dir/fuzzer_stats ]
do
        echo "Waiting for fuzzer_stats file"
        sleep 5
done

AFL_CYCLES=$(cat findings_dir/fuzzer_stats | grep cycles_done | cut -d ':' -f2 | cut -d ' ' -f2)

while [ $AFL_CYCLES != 5 ]
do
        AFL_CYCLES=$(cat findings_dir/fuzzer_stats | grep cycles_done \
                | cut -d ':' -f2 | cut -d ' ' -f2)
done

#Stop afl fuzzing
kill -SIGINT $AFL_PID

#Display line numbers for debugging output
export ASAN_SYMBOLIZER_PATH=/usr/bin/llvm-symbolizer-3.5
export ASAN_OPTIONS=symbolize=1

#Compile the program with Clang (AddressSanitizer)
clang -fsanitize=address -g math_program.c


#Switch back to original directory
cd $ROOT_DIR

#Execute all test cases from afl which caused crashes and look for
#use-after-free errors

rm ProjectGitRepo/findings_dir/crashes/README.txt

for f in ProjectGitRepo/findings_dir/crashes/*
do
        cat $f | ./ProjectGitRepo/a.out 2>asan_log.log

        #See if there were any errors
        if [ -s asan_log.log ]
        then
                #Parse AddressSanitizer log for use-after-free errors and JSONify
                LOGFILE="asan_log.log"
                CFILE="ProjectGitRepo/math_program.c"
                TESTFILE="ProjectTests/test-afl.sh"
                JSONFILE="json/use_after_free.json"
                python additive_log_parse.py $LOGFILE $CFILE $TESTFILE $JSONFILE
        else
                echo "ERROR_FREE" > asan_log.log
        fi
done
```

Now, we must have *do_all_tests.sh* call *test-afl.sh* and return the results back to the host. Adding the following lines to *do_all_tests.sh* implements this.

```
./ProjectTests/test-afl.sh

#After the tests are done, serve the JSON file
#(listing the vulnerabilities) with netcat

nc 169.254.254.1 9000 -q 0 < json/use_after_free.json > /dev/null
```

rx_results.sh should look as follows.

```
#!/bin/bash

#Switch to the test results directory
cd TestResults

#Receive the results in proper order
nc -l -p 9000 -s 169.254.254.1 -q 0 > use_after_free.json
```

The program which will be tested in this example is the same one as the previous example except it has been modified to accept its input number over STDIN and not as a command line argument. This makes it easier to test will afl. Source code for this program, *math_program.c* can be found below.

```
#include <stdio.h>

int main(int argc, char **argv)
{
        int x;
        int y;
        //printf("Value of first argument was %s\n",argv[1]);
        //x = atoi(argv[1]);
        printf("Enter a number: ");
        scanf("%d",&x);
        printf("\n");

        int *my_buffer = malloc(64 * sizeof(int));

        if (x > 400)
        {
                int i;
                for (i = 0; i < 64; i++)
                {
                        my_buffer[i] = x + i;
                }

                for (i = 0; i < 16; i++)
                {
                        my_buffer[i] =  my_buffer[i] + my_buffer[63 - i] * 3;
                }
                //Output
                for (i = 0; i < 64; i++)
                {
                        printf("-%d-",my_buffer[i]);
                }
                printf("\n");

                free(my_buffer);
        }

        //Should have been 'else if'
        if (x > 200)
        {
                int i;
                for (i = 0; i < 64; i++)
                {
                        my_buffer[i] = x - i;
                }

                for (i = 0; i < 16; i++)
                {
                        my_buffer[i] =  3*my_buffer[i] + my_buffer[63 - i];
                }

                //Output
                for (i = 0; i < 64; i++)
                {
                        printf("-%d-",my_buffer[i]);
                }
                printf("\n");

                free(my_buffer);
        }

        return 0;
}
```

Running *./do_tests.sh* will now run 5 cycles of fuzz testing with American Fuzzy Lop, check for use-after-free bugs with Address Sanitizer and report any bugs found through the web interface.