COMS E6998-9: FruityFuzz

Due on May 11, 2011

Thompson - M 2:10P-04:00P

Jordan Schau

Introduction

Fuzz testing has been a technique used to find bugs in programs and its success has been growing. Initially, fuzz testing was used to find small bugs in command line programs, but has since evolved in to a technique that is commonly used to find exploitable bugs in distributed software. Because there are so many applications for fuzzers, there are an enormous amount of fuzzers to handle different programs and situations. In addition, there are more generic fuzzing frameworks that allow the user to specify certain formats and then fuzzed input is generated automatically, on the fly. Often times, these frameworks have a steep learning curve and are not easily used by the average developer doing routine bug finding. Likewise, the more specific file fuzzers are not generally portable to be used on many different applications without significant modification. Furthermore, like most software and bug finding applications, most of the development of fuzzers has been geared towards Windows OS and not the unix based platforms[2].

Mac Fuzzers

Fuzzers and Fuzzing frameworks that function on Mac OS X 10.6 are extremely limited[1]. Currently, PeachFuzz[3] and Sully[4] are two popular frameworks that work on Mac. Unfortunately, they are complex and geared toward security researchers and therefore not very useful to the average developer. Not only are the frameworks complex, they are poorly documented for the Mac and most of the testing and examples are made to run on Windows.

A bit about Mac OS X

The current state of most Apple computers running OS X is an important thing to consider when designing a fuzzer aimed at Mac Apps. Firstly, because Apple has strict control over the hardware used as well as the operating system, many assumptions can be made. The first being, all processors produced in current models are 64 bit architecture. Therefore, the fuzzers relying on an 8 register system on Windows cannot easily be ported to the Mac. The current process structure on Macs allows some programs to run in 32 bit mode while others run at 64 bit mode, so this must be taken in to consideration. Another thing to note is that Mac has not fully implemented address space layout randomization (ASLR). More specifically, certain libraries have their address space randomized, but in most cases it is not.

FruityFuzz

FruityFuzz[1] is a simple mutation-based file fuzzer. It takes as arguments, an executable, an example file, and a few optional variables that give the user a bit more control over the execution.

Tools Used

Because of its good high-level file I/O, I used Python 2.7 to implement the fuzzer. Python, unfortunately, has poor multithreading capabilities so as of now, the fuzzer runs serially and

[1]The name comes from the simple fact that Apples are Fruits!

waits for each test to complete before it starts the next.

The second major tool used is CrashReporter. Because all distributions of Snow Leopard come with CrashReporter pre-installed, it was an obvious choice to use instead of the typical debugger. CrashReporter generates a report for each crash which outlines the reason for the crash. In particular, the crash report details the information in each thread of the application as well as the values in all of the registers at the time of the crash. This information is valuable to the user for future debugging. There is a small drawback to using CrashReported, after 20 crashes, reports get deleted. The small fix of copying reports into a local folder was implemented to solve the problem.

Algorithm

1. Select a valid input file (selected by the user)
2. Select a random byte offset within the file
3. Insert fuzz at random point
4. Run fuzzy file
5. Check to see if a crash occurred
    - If crash report exists: move report into results directory
    - If crash did not happen: delete file
6. Return to step 2

Of the fuzzed inputs, there are some interesting strings to use. Because most native applications on the Mac are built in Objective-C, that is a good place to start in finding appropriate strings. The fuzz strings are chosen at random from a predetermined dictionary of *interesting* strings. The first set are the typical bounds checking, buffer overflow finding strings. They are:

```
"J" * 255, "J" * 256, "J" * 257, "J" * 420, "J" * 511, "J" * 512, "J" * 1023,
"J" * 1024, "J" * 2047, "J" * 2048, "J" * 4096, "J" * 4097, "J" * 5000,
"J" * 10000, "J" * 20000, "J" * 32762, "J" * 32763, "J" * 32764,
"J" * 32765, "J" * 32766, "J" * 32767, "J" * 32768, "J" * 65534,
"J" * 65535, "J" * 65536, "%x" * 1024, "%n" * 1025 , "%s" * 2048,
"%s%n%x%d" * 5000, "%s" * 30000, "%s" * 40000
```

Another interesting set of strings to use is special characters. Objective-C is a superset of C so all the C special characters apply as well as a few more. For these, the fuzzer uses:

```
"%.1024d", "%.2048d", "%.4096d", "%.8200d", "%99999999999s",
"%99999999999d", "%99999999999x", "%99999999999n",
"%99999999999s" * 1000, "%99999999999d" * 1000,
"%99999999999x" * 1000, "%99999999999n" * 1000, "%08x" * 100,
"%%20s" * 1000,"%%20x" * 1000,"%%20n" * 1000,"%%20d" * 1000,
"%#0123456x%08x%x%s%p%n%d%o%u%c%h%l%q%j%z%Z%t%i%e%g%f%a%C%S%08x%%#0123456x
%%x%%s%%p%%n%%d%%o%%u%%c%%h%%l%%q%%j%%z%%Z%%t%%i%%e%%g%%f%%a%%C%%S%%08x",
```

```
"%@" * 1000, "%@%%%d%D%i%u%U%hi%hu%qi%qu%x%X%qx%qX%o%O%f%e%E%g%G%c%C%s%S%p%L
%a%A%F%z%t%j%ld%lx%lu%lx%f%g%lp%lx%p%lld%llx%llu%llx"
```

In the set above, special characters are used in combination with very long strings. This helps to over flow a pointer or a buffer and then return valuable information.

Targets and Result

In my testing I targeted two core pieces of software. The first being VLC (Version 1.1.9). VLC is an obvious target for a few reasons. For one, it is open source and finding a bug could led to a quick fix with its strong developer community. Another reason for fuzzing VLC is that it is widely distributed and supports a tremendous amount of file types. One of these file types is .FLV — flash video.

The next target was Preview (Version 5.0.3 (504.1)). Preview comes preinstalled on all Macs and is a major application used for viewing PDFs as well as other documents. Like VLC, it supports many different filetypes, some more esoteric than others. However, the main reason to try and find bugs in Preview is the fact that it serves as the backbone to many other Mac applications. For instance, PDFs opened in Safari are rendered with Preview. A bug in Preview can lead to bugs in many more core Mac Applications.

In VLC, fuzzing with flash video files, I found many bugs. Most of them were buffer overflows. Unfortunately, I did not find any bugs in Preview with my mutation-based fuzzer. This may be accounted to the fact that the majority of my testing was done using VLC and

that Preview is produced by Apple and has gone through a more extensive testing process before release.

Drawbacks

For the purposes of this project, the necessary functionality was implemented, but during testing, I found things I would like to add in the future. The major drawback to FruityFuzz is that it has no understanding of code coverage. That is, with each iteration, the fuzzer is unaware if it hits different parts of the code. In addition, unlike the frameworks, FruityFuzz does not have the capability to handle specific protocols or file formats. For instance, suppose a file has its first 20 bytes reserved as a hash that is checked at initial load-time. If the fuzzer overwrites those bytes, it will surely not find anything. Future work would allow the user to specify certain constraints in terms of file format or protocols. Another small drawback is that the fuzzer is dependent on the current distribution of OS X. If Apple decides to change the way CrashReporter handles crashes or some other unforeseen change, the fuzzer most likely will not be compatible. Lastly, a drawback comes directly from Python. In particular, Python does not handle multitasking and multithreading well. This led me to develop a serial flow that is slow when compared to something that runs in parallel.

Future Work

For reasons mentioned above, my future plans are to implement many of the features I feel this version lacks. Adding certain features will both increase the likelihood of finding bugs

as well as the ability to recognize exploitable vulnerabilities. Adding multithreading support will allow for a parallel flow and essentially allow multiple tests to run at once. Another nice feature would be robust reporting. Rather than using CrashReporter, porting PyDbg to the current OS X might be a successful means to access system level memory and register information. Lastly, the fuzzer could benefit from having protocol aware trials. Without creating a framework like PeachFuzz or Sulley, implementing simple protocol detection and awareness might allow for a higher number of bugs to be found.

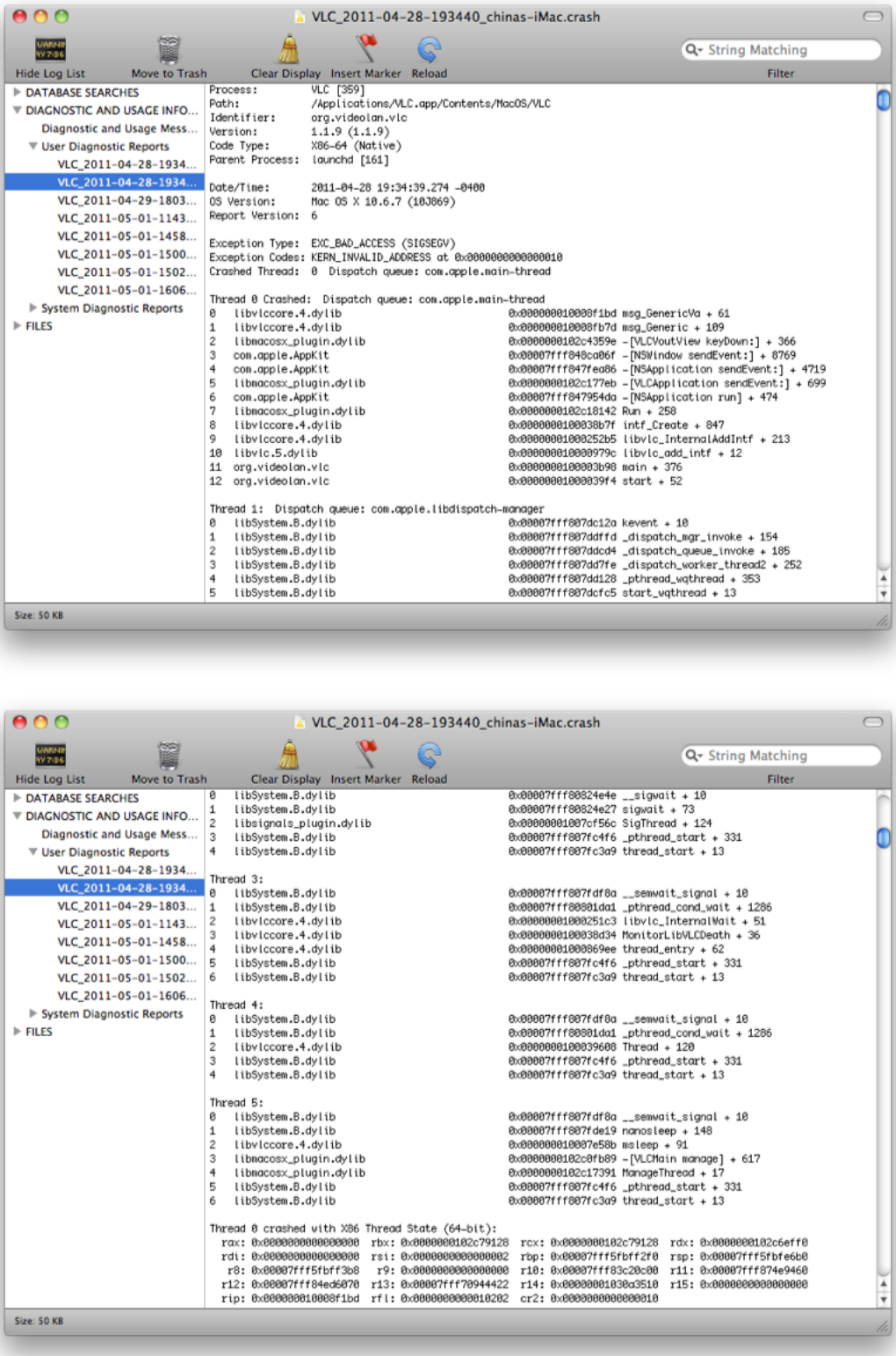

Figure 1: Typical Reports Generated by CrashReporter

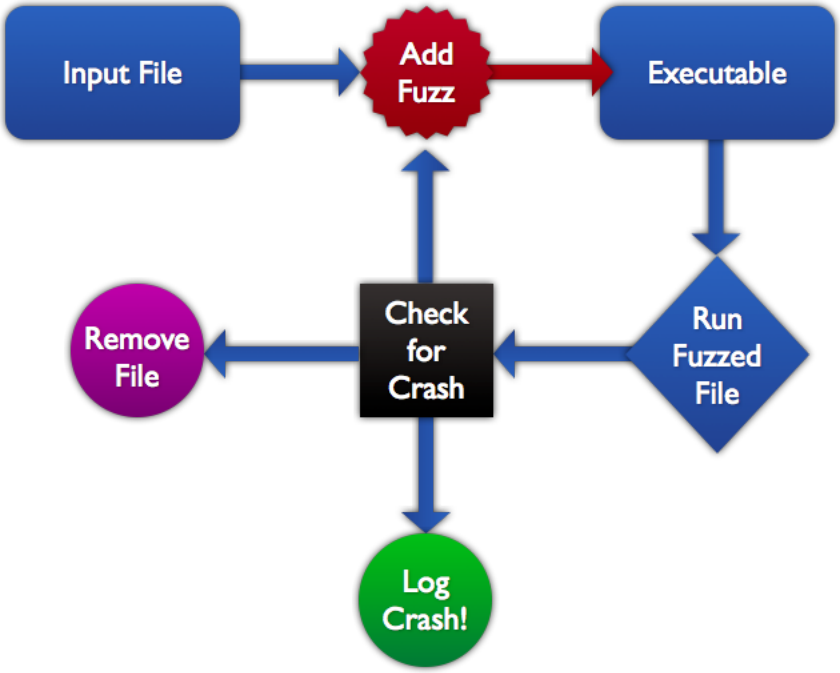

Figure 2: General Overview of Fuzzer Flow

Usage

```
python FruityFuzz.py -a ./path/to/executable -f ./path/to/file
        -t ./path/to/directory/for/test/files [options]
    [required]
        -a: Path to Application Executable
        -f: File to seed the Fuzzness
    [options]
        -t: Directory to put the test cases (defaults to ./test/)
        -c: Number of test cases to run (defaults to 75)
        -T: wait time between trials (default is 5 seconds)
        -v: Verbose
        -h: Help page


Ex: python FruityFuzz.py -a '/Applications/Preview.app/Contents/MacOS/Preview'
        -f '/Users/username/Desktop/fuzzypsd.psd' -t
        '/Users/username/Desktop/fuzz2 -c 50 -T 3 -v
```

References

[1] Charles Miller and Dino Dai Zovi. *The Mac Hacker's Handbook*. Wiley, 2009.

[2] Charlie Miller. Fuzz by number - more data about fuzzing than you ever wanted to know. `http://cansecwest.com/csw08/csw08-miller.pdf`.

[3] Peach Fuzzing Platform. `http://peachfuzzer.com/`.

[4] Sulley. `http://code.google.com/p/sulley/`.