BFF Linux Readme

CERT Basic Fuzzing Framework (BFF) Readme

Change Log

See BFF Release Notes

Requirements

- UbuFuzz requires VMWare Workstation 7 or later, or compatible virtualization software.
- The OS X installer requires Snow Leopard or later.
    - Xcode Tools is recommended, which provides libgmalloc (Guard Malloc).

Demo Quick Start

UbuFuzz

1. Unzip `BFF-2.7.zip` to `c:\fuzz` or another folder to be shared as "`fuzz`"
2. Unzip `UbuFuzz-2.7.zip`
3. Open `UbuFuzz.vmx`
4. Create a snapshot in VMWare
5. Power on the VM

If you do not wish to use a shared folder, simply remove `~/bff`
and unzip `BFF-2.7.zip` to the `~/bff` directory.

OS X

1. Install BFF
2. Run BFF
3. Run `./batch.sh`

To check the progress of the fuzzing campaign, run `~/bff/quickstats.sh`

Look in `~/results` to investigate the fuzzing results.

To use BFF without UbuFuzz

See BFF Installation Notes

Additional information

The configuration for VMWare may prevent virtual machines from utilizing shared folders by default. You may need to manually enable shared folders for the VM after opening the VMX. If you chose to unzip scripts.zip to a folder other than `c:\fuzz`, then you will need to modify the properties of the shared folder in VMWare to point to the new location of the files. Alternatively, if you may unzip the BFF scripts into `~/bff` if you do not wish to use a shared folder.

The fuzzing virtual machine is preconfigured to automatically begin a fuzzing run on several image format decoders provided by ImageMagick's "`convert`" program. An old (5.2.0) version of ImageMagick is preloaded onto the VM. ImageMagick was built with debug symbols as well as non-optimized. This makes gdb provide more useful crash reports. ImageMagick was configured using the following command:

```
CFLAGS="-g -O0" ./configure --without-x
```

Analyzing results

When the fuzzing run encounters a crash, BFF will analyze the details of the crash. This involves capturing stderr, gdb, valgrind, and callgrind output. The gdb output contains several pieces of information, including the memory map, signal information, backtrace, registers, disassembly, as well as output from the CERT Triage Tools, which indicates possible exploitability of the crash. On the OS X platform, CrashWrangler is used instead of gdb. By looking at the backtrace, BFF will keep track of which test cases cause unique crashes. Each unique crash will be placed in the configured output directory.

```
results/
|-- bff.cfg
|-- bff.log
|-- crashers
| |-- <crash_id>
| | |-- <crash_id>.log
| | |-- <analysis_tools>.stderr
| | |-- minimizer_log.txt
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>-minimized.<seedfile_ext>
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>-minimized.<seedfile_ext>.callgrind
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>-minimized.<seedfile_ext>.callgrind.annotated
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>-minimized.<seedfile_ext>.callgrind.calltree
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>-minimized.<seedfile_ext>.gdb
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>-minimized.<seedfile_ext>.stderr
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>-minimized.<seedfile_ext>.valgrind
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>.<seedfile_ext>
| | |-- sf_<seedfile_md5>-<zzuf_seed_number>.<seedfile_ext>.gdb
| | `-- sf_<seedfile_md5>.<seedfile_ext>
|-- seeds
| |-- <seedfile1_md5>
| | `-- zzuf_log.txt
| |-- <seedfile2_md5>
| | `-- zzuf_log.txt
| |-- seedfile_set.log
| |-- sf_<seedfile1_md5>.<seedfile1_ext>
| |-- sf_<seedfile2_md5>.<seedfile2_ext>
`-- uniquelog.txt
```

The UbuFuzz startup script, `batch.sh`, looks for the BFF code in `/home/fuzz/bff`, which by default is a soft link to the VMWare shared folder `/mnt/hgfs/fuzz`. The default results location is `/home/fuzz/results`, which in the UbuFuzz vm is a soft link to `/home/fuzz/bff/results` (thus `/mnt/hgfs/fuzz/results`). Either of these soft links can be changed if desired, or you can edit conf.d/bff.cfg to point BFF at a different destination.

BFF will copy its configuration to results/bff.cfg, and will log messages of level INFO or higher into results/bff.log. The config file copied here is just for recording purposes, another copy is made in /home/fuzz and it is this copy of the file that actually gets used by BFF.

Additionally the following subdirectories are created in the results dir:

- `crashers`: Contains a subdir for each uniquely-crashing test case and its analyzed results
- `seeds`: Contains the original seedfiles as well as logs specific to that seedfile

Other files of note:

- `results/uniquelog.txt` – a log file that tracks the unique crashers found during the run

The `results/crashers` directory will contain the uniquely-crashing test cases. The variants that have crashed the target application will be stored here with the zzuf seed number appended to the seed file name. For each uniquely-crashing case, there will also be a `.stderr`, `.gdb`, `.callgrind` and `.valgrind` file that contains the stderr, gdb, callgrind and valgrind output for that case, respectively.

The CERT BFF minimizes crashers. In other words, each crashing test case will have a number of bytes that have been modified from the seed file. When a crasher is minimized, a test case is generated with a minimal number of bytes that have changed from the seed file. The minimized test case will have "-minimized" inserted into the filename before the extension. Also, a minimizer_log.txt file is created containing a log of the minimization process. This is used by `tools/minimizer_plot.py` to produce a chart showing the minimizer progress.

Aside from minimizing to the seed file, BFF also includes the ability to minimize a crashing testcase to a string pattern. This will show which bytes of a file may be altered without affecting the crash. Due to performance issues, the minimize-to-string option is disabled by default. However, the minimizer can be run in a standalone mode.

Say you have a crashing test case, but you really need to get it to a proof-of-concept exploit. The problem is when you load the crash into your debugger you can't easily tell which registers, stack values, or memory locations are under your control. But what if you could change the crashing test case so that it had only the bytes required to cause that crash, and the rest were all masked out with a fixed value, say "x" (0x78)? Then you'd know that if you saw EIP=0x78787878, you may already be a winner. The minimize-to-string option does just that.

To get command-line usage of the minimizer, run:

```
tools\minimize.py --help
```

To minimize a crashing testcase to the Metasploit string pattern, run:

```
tools\minimize.py --stringmode <crashing_testcase>
```

When minimizing to the Metasploit pattern, FOE will use the resulting byte map to create an additional minimized file that uses a string of 'x' characters.

Note that this file is not guaranteed to produce the same crash as the original string minimization.

Metasploit pattern enumeration

Especially with larger files, you may notice that the Metasploit pattern repeats several times over the length of a Metasploit-minimized crasher. Given any particular dword, it may not be obvious which instance is the one that you are dealing with. This is where the tools\mtsp_enum.py script comes in handy. For example, let's say that you have a crasher.doc were EIP = "Aa0A" If you run:

```
tools\mtsp_enum.py Aa0A crasher.doc
```

You will end up with a file called `crasher-enum.doc`. With this file, every instance of the byte pattern "Aa0A" will be replaced with a unique, incrementing replacement. For example, "0a0A", "1a0A", "2a0A", etc. Now when you open crasher-enum.doc, you could for example get EIP = "5a0A". If you search for that pattern in the file, there should be only once instance of it. Note that you can use a search pattern of any length and you can also search for hex values. For example: "\x01\x02\x03\x04"

Analysis tools

The analysis directory contains a few tools for analyzing the results of a fuzz run.

Try `python <script> --help` for detailed usage options.

- `tools/bff_stats.py` – generates a concise summary of the fuzz run results so far, including how many times each unique crash was seen, the first seed number it was seen at, the most recent seed number it was seen, and the bitwise and bytewise Hamming Distance from the original seedfile for the minimized testcase.
- `tools/callsim.py` – can display crashes clustered by the similarity of their called functions. This analysis is based on the idea that crashes with similar call history are likely to be related even if they result in unique crash hashes. The resulting clusters of crashes can be useful in deciding which crashes to investigate first given a large number of crashes. The source data for this analysis is callgrind output generated for each crash.
- `tools/create_crasher_script.py` – will generate a shell script that in turn can be used to regenerate all the test cases for a given crash id. The use of the '--destination' option is highly recommended.
- `tools/minimizer_plot.py` – plots the minimizer data for a given crash, showing how the minimizer tunes its parameters as it progresses in order to find the optimal minimized test case.
- `tools/drillresults.py` – will search for crashing test cases that are more easily exploitable than the others. It searches based on the type of crash as well as whether the faulting address matches patterns in the fuzzed file.
- `tools/repro.py` – will launch the specified application using the same command-line parameters as configure for the fuzzing campaign. This can be used to test crashing testcases interactively.

Fuzzing on your own

When the UbuFuzz VM is powered on, it will automatically execute the `batch.sh` script in the VMWare shared folder. In order to power on the virtual machine without it beginning a fuzzing campaign, you should rename `batch.sh`. This will allow you to power on the virtual machine to install the target software. Once BFF has started a fuzzing run, it will copy `bff.cfg` to the `/home/fuzz` directory in the virtual machine. This configuration file will be used for subsequent fuzzing runs, rather than the copy in the shared folder. This is why it is important to make a snapshot of the VM in its clean state. If you wish to reset a fuzzing machine to a clean state, e.g. to start a new fuzzing campaign or if you've change fuzzing parameters or seed files,you should run the `~/bff/reset_bff.sh` script.

The first step to beginning a fuzzing run is to obtain and install the target software. Usually this process will involve downloading the application source code and compiling a debug version of it. Rather than using Ubuntu's package repository, this will ensure that you will be fuzzing the latest version of the target application. It will also give you the ability to have debug symbols, as well as the ability to use a non-optimized build if you like.

Create a new snapshot of the VM after the target software has been installed.

The `bff.cfg` file contains all of the parameters for the fuzzing run. This file must be edited to suit the software that you will be fuzzing. The `bff.cfg` file is annotated and should be relatively self-explanatory.

The default `bff.cfg` file will start a fuzzing run that invokes:

```
convert $SEEDFILE /dev/null
```

This will use ImageMagick's "`convert`" program to process the seed file, outputting to `/dev/null`. Because BFF will mangle the seed file, this fuzzing run will exercise ImageMagick's decoding capabilities.

You should choose a command line that efficiently exercises the capabilities of the application that you wish to test. Ideally, this will involve a command-line application that terminates immediately upon completion. In such cases, the fuzzing run will be CPU-bound, as BFF will invoke the application as fast as it can. Applications that have a GUI can still be fuzzed using this framework. However, in most cases, each invocation of the target application will need to wait for the specified timeout to elapse before the process is terminated.

BFF analyzes gdb (or CrashWrangler on OS X) backtraces to determine uniquely-crashing testcases. The default setting is to hash the last five code locations in the backtrace to determine a hash for the crash. We have found this value to be effective for most applications, however you can adjust the value to fit your needs.

Another important `bff.cfg` option is the *copymode* setting. By default, zzuf will use LD_PRELOAD to hook into an application to perform input mangling and intercept signals. In some cases, the target application will not behave properly in this mode. The file may fail to be mangled, or the target application may fail to run properly at all. If this is the case, set *copymode* to 1 and zzuf will create a temporary file and then open that file with the target application. Most OS X applications require copy mode, so the default `bff.cfg` provided on OS X will default to this mode.

There are several options within `bff.cfg` related to application timeouts. *progtimeout* is the maximum time that BFF will allow a single invocation of the target application to run before terminating it. In the case of the `convert` component of ImageMagick, it is reasonable to expect the program to finish within a few seconds. Depending on the application you are fuzzing, this value will need to be adjusted. Especially with GUI applications, the goal is to allow the application to run long enough to process the input file, but not so long that the application is sitting idle for an amount of time for each invocation. The *killprocname* and *killproctimeout* options are used for the external process killer. When running some analyzers, the application that you are fuzzing may be left running after the analyzer is terminated. The external process killer (`killproc.sh`) is a script that polls running processes and makes sure that no single instance of the specified application is allowed to run for longer than the specified time.

Once you have configured the options in `bff.cfg`, power on the UbuFuzz VM. It should begin fuzzing automatically. If the target is a GUI application, you should see the application launch repeatedly as it is being fuzzed. If it is a command-line application, you should notice increased CPU usage in the top window. The first seed_interval that BFF executes will also display stderr to the terminal window to let you see if something is obviously wrong. If you need to tweak your settings, such as by using a smaller *progtimeout* value to improve throughput, you can edit the `bff.cfg` file and restore the VM to its previous snapshot. Alternatively, you can run `~/bff/reset_bff.sh` and restart X to cause the VM to re-read the shared `bff.cfg` file and start fuzzing again.