FOE 2.1 Windows Readme

Failure Observation Engine (FOE) 2.1 README

License

See LICENSE.txt

Change Log

See NEWS.txt

Quick Start

Because fuzzing can fill temporary directories, put the target application in an unusable state, or trigger other operating-system-level bugs, we recommend that FOE be used in a virtual machine.

Run `FOE-2.1-setup.exe` in a virtual machine to install FOE 2.1.

The installer should detect and attempt to download prerequisites and configure your environment appropriately.

Running FOE

1. Click the FOE2 item in the Windows Start menu.
2. Run `foe2.py`
3. Run `tools\quickstats.py` to check fuzzing progress when you wish.

How it works

When a campaign starts, FOE will gather available seed files and create scorable sets:

1. The seed files themselves
2. The fuzz percent ranges for each seed file

Each interval of a campaign will choose a seed file, and then for that file, it will choose an percent range to mangle the file. After mangling the file, FOE will launch the target application, using the configured command line to have it parse the fuzzed file. If the "winrun" runner is compatible with the current platform, this is accomplished by preloading a crash-intercepting hook into the target application's process space. This allows crash detection without relying on a debugger. The "nullrun" runner simply runs each invocation through the debugger (cdb).

When a crash is detected, it is then verified using a combination of cdb and the Microsoft !exploitable debugger extension. If the crash is determined to be unique (by the chain of !exploitable crash hashes), then some additional analysis steps are taken:

1. A !exploitable report is created for each continuable exception.
2. If configured to do so, FOE will create a minimized test case.
3. The seed file and percent range that were used to fuzz are scored

Seed files that produce more crashes are given a preference over less-productive files, and for each seed file, the mangling percent ranges that are more productive are also given preference. These scoring features together minimize the amount of knowledge required to perform an effective fuzzing campaign.

Analyzing results

```
.\results\<campaignid>\
  +- <configname>.yaml
  +- version.txt
  +- <SEVERITY>/
     +- <hash_1>/
        +- minimizer_log.txt
        +- sf_<seedfile_hash>.<ext>
        +- sf_<seedfile_hash>-<iteration>-<EFA>.<ext>
        +- sf_<seedfile_hash>-<iteration>-<EFA>-<SEVERITY>.<ext>.msec
        +- sf_<seedfile_hash>-<iteration>-<EFA>-minimized.<ext>
        +- sf_<seedfile_hash>-<iteration>.<ext>.e<n>.msec
     +- <hash_2>/
     +- ...
     +- <hash_n>/
```

`<configname>.yaml`
This is a copy of the config file used for this run. It is stored for historical purposes ("Which options did I use for that run?").

`version.txt`
This file stores the version of FOE that was used for fuzzing.

`<SEVERITY>`
This is the "Exploitability Classification" assigned to the crash by !exploitable. Values can be EXPLOITABLE, PROBABLY_EXPLOITABLE, UNKNOWN, or PROBABLY_NOT_EXPLOITABLE. For crashes that include multiple exceptions, the highest exploitability of any of the exceptions is used for this directory. Be aware that !exploitable has limitations and only provides a rough (possibly false-positive) assesment of a crash.
More information on !exploitable can be found here:
http://msecdbg.codeplex.com/
http://blogs.technet.com/b/srd/archive/2009/04/08/the-history-of-the-exploitable-crash-analyzer.aspx

`<hash_n>`
This is the hash in Major.Minor form provided by !exploitable.

`minimizer_log.txt`
This is the log file that was produced during crash minimization.

`sf_<seedfile_hash>.<ext>`
This is the original file (pre-fuzz). This is provided as a convenient "diff" source.

`sf_<seedfile_hash>-<iteration>-<EFA>.<ext>`
This is the fuzzed file that caused the crash. <EFA> is the exception faulting address, as reported by !exploitable.

`sf_<seedfile_hash>-<iteration>-<EFA>-<SEVERITY>.<ext>.msec`
This is the cdb text output from the crash, which includes output from the !exploitable tool.

`sf_<seedfile_hash>-<iteration>-<EFA>-minimized.<ext>`
This is the minimized version of the crashing test case. It is the "least different" version of the original fuzzed file that caused a specific crash (hash).

`sf_<seedfile_hash>-<iteration>.<ext>.e<n>.msec`
This is the cdb output for an exception that is continued <n> number of times. One file is provided for each continued exception until an uncontinuable exception is encountered, or the handled exception limit has been reached, or the target application proceeds without encountering another exception.

Fuzzing on your own

Once you are comfortable with FOE's default ImageMagick fuzz run, you can try fuzzing an application of your choice. The first step is to place seed files into the FOE seedfiles directory. These are the files that will be mangled and opened by the target application. Next modify the foe.yaml file to suit your needs. The foe.yaml file is documented to describe what each of the features mean. The important parts to modify are:

`campaign: id:`
This field is used in determining the fuzzing campaign, and subsequently, where the results should be stored. This should probably be the target application name and version.

`campaign: use_buttonclicker:`
When fuzzing a GUI application, the FOE button clicker can increase throughput and code coverage. Note that the button clicker is not configurable, but rather it has a built-in heuristic for determining which buttons to click.

`target: program:`
This is the full path to the target application that you wish to fuzz.

`target: cmdline_template:`
This specifies the commandline syntax for invoking the target application.

`runner: runtimeout:`
This value specifies how long FOE should wait before terminating the application and moving on to the next iteration. Note that this setting only applies to the "winrun" runner (32-bit Windows XP and Server 2003 systems).

`debugger: runtimeout:`
This value specifies how long FOE should allow the target application to run when it is invoked from the debugger. On platforms that use the "null" runner (64-bit Windows or Windows Vista or newer), this is the only timeout value that is used.

FOE periodically saves state of a fuzzing campaign, so it will by default continue a cached campaign if foe.yaml has not been modified.
To clear the FOE cached state, run:
`tools\clean_foe.py`
For additional options, run:
`tools\clean_foe.py --help`

Digging deeper into results

When FOE has produced results, you may wish to perform some additional steps.

Finding interesting crashes:

With some target applications, FOE may produce too many uniquely-crashing test cases to investigate manually in a reasonable amount of time. We have provided a script called drillresults.py to pick out crashes that are most likely to be exploitable and list those cases in a ranked order (most exploitable first).

To run this script, run:
`tools\drillresults.py`
For command-line usage, run:
`tools\drillresults.py --help`

Reproducing crashes:

The `tools\repro.py` script can be used to reproduce a crash by running it in the same manner that FOE did.
For command-line usage, run:
`tools\repro.py --help`

Comparing zip-based files:

The `tools\zipdiff.py` script can be used to compare zip-based files.
For command-line usage, run:
`tools\zipdiff.py --help`

Minimization to string:

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`"

Included Fuzzing Strategies

bytemut: replace bytes with random values
swap: swap adjacent bytes
wave: cycle through every possible single-byte value, sequentially
drop: removes one byte from the file for each position in the file
insert: inserts a random byte for each position in the file
truncate: truncates bytes from the end of the file
crmut: replace carriage return bytes with random values
crlfmut: replace carriage return and linefeed bytes with random values
nullmut: replace null bytes with random values
verify: do not mutate file. Used for verifying crashing testcases
range_list: byte ranges to be fuzzed. One range per line, hex or decimal

Verifying crashing results

FOE can be used to verify crashing test cases. This can be useful for when a new version of an application is released or if you are the developer and you want to see how many uniquely-crashing test cases disappear when you fix a bug. To perform a verfification campaign:

1. Run tools\copycrashers.py to collect all of the crashing cases from a campaign. By default it will copy all of the uniquely-crashing test cases to the "seedfiles" directory, which should be empty.
2. Modify configs\foe.yaml to use the "verify" fuzzer and also specify a new campaign ID.

When you run FOE, it will run each case with the target application, and cases that still crash will be placed in the results directory for the new campaign.

Manually Installing FOE

If you have installed FOE using the installer, you can skip this section. To install FOE manually, you will need the following prerequisites:

- Windows XP or Server 2003 32-bit is recommended to allow exception hooking (winrun) Other Windows versions will use debugger mode (nullrun)
- Python 2.7
  http://www.python.org/download/releases/2.7.5/
- SciPy
  http://sourceforge.net/projects/scipy/files/scipy/0.10.1/scipy-0.10.1-win32-superpack-python2.7.exe/download
- NumPy
  http://sourceforge.net/projects/numpy/files/NumPy/1.6.1/numpy-1.6.1-win32-superpack-python2.7.exe/download
- PyYAML
  http://pyyaml.org/download/pyyaml/PyYAML-3.10.win32-py2.7.exe
- pywin32
  http://sourceforge.net/projects/pywin32/files/pywin32/Build%20218/pywin32-218.win32-py2.7.exe/download
- Python WMI
  https://pypi.python.org/packages/any/W/WMI/WMI-1.4.9.win32.exe
- Debugging Tools for Windows
  http://www.microsoft.com/whdc/devtools/debugging/default.mspx
  Set up symbols, if so desired.
- Microsoft !exploitable
  Copy the !exploitable dll (`msec.dll`) to winext directory.
  (probably `C:\Program Files\Debugging Tools for Windows (x86)\winext`)
  http://msecdbg.codeplex.com/
- Add debugging tools (specifically `cdb.exe`) to your PATH.
  (probably C:\Program Files\Debugging Tools for Windows (x86)\)
- Copy the `foe.yaml` config file from `configs\examples\` to the `configs` directory and modify as necessary.
- Copy seed files to the `seedfiles` directory