usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

AmpFuzz: Fuzzing for Amplification DDoS Vulnerabilities

Johannes Krupp, *CISPA Helmholtz Center for Information Security;*
Ilya Grishchenko, *University of California, Santa Barbara;* Christian Rossow,
CISPA Helmholtz Center for Information Security

https://www.usenix.org/conference/usenixsecurity22/presentation/krupp

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Artifact Appendices to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

A Artifact Appendix

A.1 Abstract

This artifact contains the source code of the AmpFuzz fuzzer as well as a number of scripts that were used to evaluate it on a number of programs from the Debian repositories. As the evaluation pipeline is configured to run in multiple docker containers, a linux-based host-system running the docker daemon is required. To confirm that the artifact is functional and to reproduce individual results, a commodity laptop or computer does suffice. E.g., development took place on a core i5 with 8GB of RAM, on which re-discovery of the `bosserver` amplification vulnerability takes less than 5 minutes. To rerun the entire pipeline, which fuzzes all targets five times for 24 hours, a system with a larger number of cores and more RAM is recommended (our experiments ran on a server with two Intel Xeon Gold 6230N and 512GB of RAM, on which the pipeline finished in about 4 days).

Lastly, this artifact also contains code to synthesize python code from identified amplification vulnerabilities, which can be used to develop amplification honeypots. This step only requires a working Python3 installation on the host system and can also be run on a commodity system.

Overall, this artifact should show that

- The AmpFuzz fuzzer is functional and *can* discover amplification vulnerabilities.
- The honeypot synthesis step is functional and produces python code.
- A full 5x 24h evaluation reproduces Table 2 (within the confidence intervals provided), similar maximum amplification factors to those shown in Figure 5, and similar results to those shown in Figure 4 for UDP-aware fuzzing.

A.2 Artifact check-list (meta-information)

- **Algorithm:** No new algorithm is presented.
- **Program:** No standardized benchmark is available. Instead, AmpFuzz is evaluated on 20 services from the Debian repositories.
- **Compilation:** AmpFuzz leverages LLVM11.0.1 and builds on the compile-time instrumentation from ParmeSan and Angora. All sources are included with the artifact and automatically built.
- **Transformations:** AmpFuzz requires no external program transformation tools.
- **Binary:** No binaries are required/included.
- **Model:** No model is used.
- **Data set:** The "evaluation data set" consists of 20 debian programs. A script to reproduce the dataset from public debian repositories is included.
- **Run-time environment:** AmpFuzz was tested on Linux, requires access to a running docker daemon and relies on bash, GNU make, and xargs.
- **Hardware:** No special hardware is requried (a x86_64 processer is assumed).
- **Run-time state:** The artifact is non-sensitive to run-time state.
- **Execution:** Other heavy loads on the system could impact results.
- **Security, privacy, and ethical concerns:** AmpFuzz only performs local testing of publicly available programs. Where possible, care has been taken to prevent fuzzed services from opening external network connections.
- **Metrics:** Included scripts and the AmpFuzz report on
  - Number of unique execution paths
  - Number of unique network requests (as defined by unique basicblock coverage)
  - Number of amplification inducing network requests
  - Maximum amplification factor on layer 2 (including Ethernet frames)
  - Maximum amplification factor on layer 7 (UDP payload only)
  - Elapsed seconds until first response
- **Output:** Each fuzz run produces
  - A human-readable console log with statistics (`fuzz.log`)
  - A csv file with fine-grained statistics (`angora.log`)
  - a folder with tested requests as individual files (`queue/id:<numeric_id>`)
  - a folder with amplification inducing requests as individual files (`amps/amp_<amp_factor_l2>_<path_hash>_<input_hash>`)

  Scripts are provided to generate the tables and figures included in the paper from these raw-results:
  - `02_print_table.py` produces LaTeX code on which Table 2 is based
  - `03_plot_grid.py` procudes Figures 4 and 5
- **Experiments:** Mostly automated, see detailed description below.
- **How much disk space required (approximately)?:** For verifying functionality on individual targets, 20GB should suffice (the source-code repository requires approximately 3GB, most of this from the LLVM git repository, the basic docker containers take around 13GB). Docker containers and output from the full evaluation fit on a 1TB drive.
- **How much time is needed to prepare workflow (approximately)?:** Building the initial docker containers (step 1 above) should take less than 15 minutes.
- **How much time is needed to complete experiments (approximately)?:** Running the full evalution with no changed parameters (5 repetitions, 24 hour timeouts plus several 1 hour runs with different configurations) took about 4 days on a system with 80 threads.

- **Publicly available (explicitly provide evolving version reference)?:** AmpFuzz is publicly available at https://github.com/cispa/ampfuzz
- **Code licenses (if publicly available)?:** AmpFuzz is licensed under Apache License 2.0
- **Data licenses (if publicly available)?:** n/a
- **Workflow frameworks used?:** No workflow frameworks were used (evaluation pipeline only requires GNU make and xargs)
- **Archived (explicitly provide DOI or stable reference)?:** https://github.com/cispa/ampfuzz/releases/tag/usenix22_ae

A.3 Description

A.3.1 How to access

AmpFuz can be retrieved from GitHub via

```
git clone --recursive
 ↪  https://github.com/cispa/ampfuzz
```

A.3.2 Hardware dependencies

N/A

A.3.3 Software dependencies

- Linux host OS
- Docker
- bash
- GNU make
- GNU xargs
- Python3 with packages `pandas`, `numpy`, `seaborn`

A.3.4 Data sets

N/A

A.3.5 Models

N/A

A.3.6 Security, privacy, and ethical concerns

AmpFuzz only performs local testing of publicly available programs. Where possible, care has been taken to prevent fuzzed services from opening external network connections.

A.4 Installation

A.4.1 Build docker base images

from the project directory, run

```
make
```

This will take some time and build four docker images:

- `ampfuzz:base`: serves as the base-image for the other three stages, basically a Ubuntu 20.10 image with some packages installed and including a copy of the llvm source.
- `ampfuzz:wllvm_wrapper`: used to build ubuntu packages with `wllvm`, a the whole-program LLVM wrapper. Our later stages use `wllvm` to extract LLVM bitcode from installed packages.
- `ampfuzz:fuzzer`: includes the fuzzer and required instrumentation tools.
- `ampfuzz:symbolic_execution`: includes the symcc symbolic execution engine, and is used to instrument targets and replay the amplification inputs to collect path constraints.

A.5 Evaluation and expected results

We claim that

1. AmpFuzz fuzzer is functional and can discover amplification vulnerabilities (Table 2 in the paper)
2. UDP-aware fuzzing allows AmpFuzz to find amplification inducing responses *faster* than static timeouts (Figure 4 in the paper)
3. The amplification maximization routines of AmpFuzz can lead to higher maximum amplification factors than coverage-guided fuzzing alone (Figure 5 in the paper)

A full evaluation run should produce results from which Table 2, Figure 4 and Figure 5 could be reproduced (within the confidence intervals provided in the paper).

A.5.1 Prepare Evaluation Directory

from the `eval` subdirectory, run

```
make
```

This will generate a fresh evaluation directory in `eval/04_create_eval_dir/eval`. This directory contains

- **args**: A textfile containing the different fuzzer configurations and timeouts
- **build_scripts**: helper scripts to build containers used for fuzzing
- **eval_scripts**: helper scripts to analyze results (see below)
- **fuzz_all.sh**: bash script to run entire fuzzing pipeline
- **fuzz_scripts**: helper scripts used during fuzzing
- **hpsynth_scripts**: helper scripts used during honeypot code synthesis
- **Makefile**: a GNU make script with rules to build containers used for fuzzing (makes use of `build_scripts`)
- **targets**: configuration info for fuzz targets.
  - **<debian_package>/<path_to_binary_escape>/<port>**: configuration directory for a single fuzz target. Will also be used to store log-files and container information.
    - **args**: commandline arguments to be passed to the fuzz target

    - ∗ **config.sh**: bash script that configures the fuzz target for fuzzing

- **targets.json**: json file containing tuples of
  1. debian package
  2. path to binary
  3. port number

  for all fuzz targets.

This newly built `eval` directory can be moved around freely. Everything from here onwards will happen within this directory!

A.5.2 Fuzz

Running `fuzz_all.sh` within this newly created eval directory will now

1. use the generated `Makefile` to prepare all targets for fuzzing (i.e., building and instrumenting the target into individual docker images)
2. fuzz each target with each configuration and collect all results into a new `results` directory
3. run the paths-to-message deduplication script. This script collects all unique "paths" found during fuzzing and executes them against the dataflow-instrumented target binary, collecting only request-dependent CFG edges.

For each target and run, a new subfolder will be created of the form `results/<pkg>/<binary>_<port>/<run>`.

A.5.3 Analyze results

Once fuzzing and path-deduplication has completed, the new `results` directory can be analyzed:

1. `eval_scripts/01_compute_amp_stats.py` will extract final stats for each run into a file `results/results.json`
2. `eval_scripts/02_print_table.py` will generate latex code for the overview table shown in the paper
3. `eval_scripts/03_plot_grid.py` will generate the plots to show the results of different timeouts and amplification maximization runs

A.5.4 (optional) generate honeypot code

Prepare a target for symbolic exeuction, run constraint-collection for a run folder (`results/<pkg>/<binary>_<port>/<config>/<run>`) and convert the collected constraints to python code:

1. Build a docker container for symbolic execution of the target:

   ```
   make targets/<debian_package>/.sym_config_<path⌋
    ↪  _to_binary_escaped>_<port>.iid
   ```

2. `bash hpsynth_scripts/synth_one.sh <run_folder>` will create a constraints file named `hpsynth/sym.result` in the run folder.
3. `python hpsynth_scripts/main.py <sym.result>` will output python code for a number of `check` and `output` functions, along with a combined `gen_reply` function.

(Honeypot-skeleton for listening on ports and providing rate-limiting is not provided with this project)

A.6 Experiment customization

A.6.1 Full pipeline customization

Evaluation is controlled by two files, `args` and `fuzz_all.sh`. `args` contains the different fuzzing configurations, one per line, in the following format

```
<output_directory> <timeout> [extra_args ...]
```

E.g., the two lines

```
24h 24h
1h_100ms 1h -a=--disable_listen_ready
 ↪  -a=--early_termination=none
 ↪  -a=--startup_time_limit=100000
 ↪  -a=--response_time_limit=100000
```

will run

1. a default configuration for 24 hours and store the results into directory `24h`
2. a configuration with a static timeout of 100ms and store the results into directory `1h_100ms`

The `fuzz_all.sh` script further specifies how often each experiment should be repeated. This is controlled with the `N_RUNS` variable (defaults to `5`).

A.6.2 Individual results

Individual targets (e.g., for confirming functionality) can be tested as follows: First, to build a docker container for a specific target, run

```
make targets/<debian_package>/.fuzz_config_<path_to⌋
 ↪  _binary_escaped>_<port>.iid
```

to build one of the configred targets. For example, to prepare a container to fuzz `/usr/sbin/bosserver` from the package `openafs-fileserver` on port 7007, use

```
make targets/openafs-fileserver/.fuzz_config__usr_s⌋
 ↪  bin_bosserver_7007.iid
```

This will

- download the package sources for the target from the debian repositories
- compile it using wllvm
- install the package
- instrument the target binary for fuzzing
- apply additional configurations (from files `args` and `config.sh` found under `targets/<debian_package>/<binary>/<port>`)

A list of all valid targets can be retrieved using

```
make -qp|grep -oE
 ↪  'targets/[^/]*/.fuzz_config[^/]*iid'|sort -u
```

To fuzz the target in its newly built container, run

```
        bash fuzz_scripts/fuzz_one -r <runid>
         ↪  <debian_package> <path_to_binary>
         ↪  <port> <result_directory> <timeout>
```

where `<runid>` is just some number to identify the run (used as part of the output path).

Sticking with the example, to fuzz `/usr/sbin/bosserver` for 5 minutes and storing the results under `results/openafs-fileserver/_usr_sbin_bosserver_7007/5m/01/`, use

```
bash fuzz_scripts/fuzz_one.sh -r 1
 ↪  openafs-fileserver /usr/sbin/bosserver 7007
 ↪  results/openafs-fileserver/_usr_sbin_bosserver_⌋
 ↪  7007/5m
 ↪  5m
```

A.7 Version

Based on the LaTeX template for Artifact Evaluation V20220119.