USENIX Security '24 Artifact Appendix: **RESOLVERFUZZ: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing**

Qifan Zhang[†], Xuesong Bai[†], Xiang Li[*✉], Haixin Duan[*§¶], Qi Li[*], and Zhou Li[†✉]

[†]University of California, Irvine, [*]Tsinghua University
[§]Zhongguancun Laboratory, [¶]Quan Cheng Laboratory

A Artifact Appendix

A.1 Abstract

Domain Name System (DNS) is a critical component of the Internet. DNS resolvers, which act as the cache between DNS clients and DNS nameservers, are the central piece of the DNS infrastructure, essential to the scalability of DNS. However, finding the resolver vulnerabilities is non-trivial, and this problem is not well addressed by the existing tools. To list a few reasons, first, most of the known resolver vulnerabilities are non-crash bugs that cannot be directly detected by the existing oracles (or sanitizers). Second, there lacks rigorous specifications to be used as references to classify a test case as a resolver bug. Third, DNS resolvers are stateful, and stateful fuzzing is still challenging due to the large input space.

In this paper, we present a new fuzzing system termed RESOLVERFUZZ to address the aforementioned challenges related to DNS resolvers, with a suite of new techniques being developed. First, RESOLVERFUZZ performs constrained stateful fuzzing by focusing on the short query-response sequence, which has been demonstrated as the most effective way to find resolver bugs, based on our study of the published DNS CVEs. Second, to generate test cases that are more likely to trigger resolver bugs, we combine probabilistic context-free grammar (PCFG) based input generation with byte-level mutation for both queries and responses. Third, we leverage differential testing and clustering to identify non-crash bugs like cache poisoning bugs. We evaluated RESOLVERFUZZ against 6 mainstream DNS software under 4 resolver modes.

To facilitate functionalities of the core experiments presented in RESOLVERFUZZ, our artifacts consist of the following 4 parts: (1) Docker images ("images") of 6 evaluated DNS software, attacker client and attacker-controlled authoritative nameserver, (2) evaluation environments and dependencies, (3) testing infrastructure for test case generation and DNS software testing, and (4) differential analysis on results generated by the testing infrastructure. Together, these components could enable the functionality of our artifact and support the claims.

✉ Corresponding authors. Most of Xiang Li's work was done when visiting UCI as a project specialist.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

As discussed in our paper, fuzzing the resolver within the standard DNS infrastructure could affect the other remote nameservers. Hence, we localize the root and TLD servers in our lab network, which improves the efficiency of RESOLVERFUZZ. The local nameserver is implemented in Go and available on GitHub[1]. The installation and execution of this local nameserver is optional, and, to our best knowledge, will not affect performance or results of RESOLVERFUZZ.

A.2.2 How to access

The testing infrastructure and differential analysis of this artifact are available on GitHub[2]. The images are available on Docker Hub, where their links are listed in the README[3] of the GitHub repository. In addition, the images could be built from Dockerfiles provided in the GitHub repository[4].

To start evaluation, please first clone the GitHub repository on a computer running Linux, install necessary environments and dependencies. And then, with Docker engine installed, pull and tag the images from Docker Hub as instructed in README[5].

[1]https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/local_ns
[2]https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1
[3]https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1?tab=readme-ov-file#list-of-dns-software-tested
[4]https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/docker_images
[5]https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1?tab=readme-ov-file#02-pulling-docker-images-from-docker-hub

A.2.3 Hardware dependencies

The hardware specs of our workstation for RESOLVERFUZZ development and testing are:

- CPU: AMD Ryzen 5950X
- Memory: 128 GB
- 1TB SSD

SSD, rather than HDD, is recommended for storage to ensure the best performance.

RESOLVERFUZZ is configurable to fit workstations with different specs to boost the maximum performance by customizing the number of units deployed and tested during execution (flag `--unit_size`) and the number of payloads to be tested in each unit (flag `--payload_num`). Details are introduced in the README[6] of the testing infrastructure.

A.2.4 Software dependencies

RESOLVERFUZZ is developed and tested on Ubuntu 22.04 with Python 3.8 and Docker Engine. To set up the software dependencies, you first need to install Docker Engine[7] and Anaconda[8]. Installation of a Go compiler will be also needed as instructed[9] if the optional local nameserver is chosen to be installed.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Please be advised that, after installation of Docker Engine, it is recommended to enable to use Docker as a non-root user[10]. With this setting, docker commands do not have to be prefaced with sudo. Otherwise, sudo privilege is required for all docker commands.

Docker image installation. After installing the Docker Engine, the images needs to be pulled from the Docker hub. All Docker containers ("containers") are created from those images.

First, we need to first pull images of 6 DNS software, and tag them for local use:

```
docker pull qifanz/resolverfuzz-bind9:9.18.0
docker pull qifanz/resolverfuzz-unbound:1.16.0
docker pull qifanz/resolverfuzz-knot:5.5.0
docker pull qifanz/resolverfuzz-powerdns:4.7.0
docker pull
        qifanz/resolverfuzz-maradns:3.5.0022
docker pull
        qifanz/resolverfuzz-technitium:10.0.1

docker tag qifanz/resolverfuzz-bind9:9.18.0
            bind9:9.18.0
docker tag qifanz/resolverfuzz-unbound:1.16.0
            unbound:1.16.0
docker tag qifanz/resolverfuzz-knot:5.5.0
            knot:5.5.0
docker tag qifanz/resolverfuzz-powerdns:4.7.0
            powerdns:4.7.0
docker tag
        qifanz/resolverfuzz-maradns:3.5.0022
        maradns:3.5.0022
docker tag
        qifanz/resolverfuzz-technitium:10.0.1
        technitium:10.0.1
```

Then, we need to pull the images of the attacker client, the authoritative server and DNSTap Listener:

```
docker pull qifanz/resolverfuzz-dnstap-listener
docker pull qifanz/resolverfuzz-attacker
docker pull qifanz/resolverfuzz-auth-srv

docker tag qifanz/resolverfuzz-dnstap-listener
            dnstap-listener
docker tag qifanz/resolverfuzz-attacker
            attacker
docker tag qifanz/resolverfuzz-auth-srv
            auth-srv
```

Docker network configuration. To create a isolated environment, a separate Docker network is used solely for ResolverFuzz. All the queries and responses generated by ResolverFuzz are transmitted via the Docker network. To create such a Docker network named `test_net_batch` with a subnet 172.22.0.0/16, run the command:

```
docker network create --subnet "172.22.0.0/16"
                        test_net_batch
```

Since the authoritative server is implemented to send response packets via monitoring network traffic, enabling ICMP will automatically send back ICMP packets before our generated DNS responses are sent back. In consequence, the resolvers will never receive the packets with generated DNS responses. Therefore, we need to drop all the ICMP packets within the network.

[6] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/test_infra/README.md
[7] https://docs.docker.com/engine/install/ubuntu/
[8] https://docs.anaconda.com/free/anaconda/install/linux/
[9] https://go.dev/doc/install
[10] https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user

To drop all the ICMP packets, We need to first check the interface of the Docker network via the command:

```
ip addr
```

Then, all the network interfaces will be displayed. We need to identify the interface with the IP range 172.22.0.1/16 assigned. For example, on our workstation, we could find:

```
6: br-0ed6b350123e:
<NO-CARRIER,BROADCAST,MULTICAST,UP>
mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:6e:e1:47:92
        brd ff:ff:ff:ff:ff:ff
    inet 172.22.0.1/16 brd 172.22.255.255
        scope global br-0ed6b350123e
       valid_lft forever preferred_lft forever
```

In this case, the network interface in the OS for the Docker network `test_net_batch` is `br-0ed6b350123e`. Then, we drop all the ICMP packets on the network interface with the command:

```
sudo iptables -I FORWARD -i [network_interface]
                -p icmp -j DROP
```

On our workstation, for example, the command will be:

```
sudo iptables -I FORWARD -i br-0ed6b350123e
                -p icmp -j DROP
```

Optional local nameserver. We implemented a local nameserver in Go to avoid possible effects on other remote nameservers. Installation of this local nameserver is optional, and will not affect the performance of RESOLVERFUZZ.

To install the local nameserver, a Go compiler should first be installed. After that, to compile the local nameserver, we first need to change the terminal directory to the `local_ns`[11] folder:

```
cd local_ns
```

Then, initialize the packet dependencies for the local nameserver, and compile it:

```
go mod init local_ns
go mod tidy
go build -o local_ns local_ns.go
```

The executable binary `local_ns` will be created.

Finally, run the executable with two arguments. The first one is the network interface of the Docker network which we found in the earlier step, and the second one is the zone file in JSON format:

[11] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/local_ns/

```
sudo ./local_ns [network_interface] [zone_file]
```

For example, on our workstation, the command will be:

```
sudo ./local_ns br-35582c1d0a12 test-zone.json
```

Keep it running on the background during the execution of RESOLVERFUZZ so that all the NS referral queries for root servers, .com TLDs and attacker-controlled domains (i.e., `qifanzhang.com` and its sub-domains) will be answered locally by this program.

Python environment installation. After the installation of Anaconda, the Python environment named resolverfuzz could be imported from environment.yml[12] via the command:

```
conda env create -n resolverfuzz
                   --file environment.yml
```

To run the scripts, we first need to switch to the conda environment named `resolverfuzz`:

```
conda activate resolverfuzz
```

And then, get the path towards the Python interpreter of the `resolverfuzz` environment. This path will be used during evaluation.

```
which python
```

A.3.2 Basic Test

Running the Installation (Section A.3.1) without errors will ensure the testing infrastructure and differential analysis function properly. To validate, reviewers could run any of the four modes of the testing infrastructure listed in the `test_infra`[13] folder. For example, if you would like to run `main_cdns.py` for validation, you could run the following command in the `test_infra` folder:

```
sudo /path/to/conda/env/bin/python
                                main_cdns.py
```

where `/path/to/conda/env/bin/python` is the path you got when executing the command `which python` in Section A.3.1. If the program could finish execution without any error, it means the evaluation environment has been set up properly.

[12] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/environment.yml

[13] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/test_infra

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): RESOLVERFUZZ has managed to implement the workflow demonstrated in Figure 2 in the full paper. In detail, as introduced in Section 4.1 in the full paper, RESOLVERFUZZ could 1) first generate two-dimensional query-response test cases based on probabilistic context-free grammar (PCFG), and 2) collect the information about DNS cache, network traffic, and process information when each generated test case is tested.

A.4.2 Experiments

In general, reviewers should follow the README in the `test_infra` folder for **C1**. Each python script will automatically complete the evaluation of the testing infrastructure in each mode. There are in total 4 experiments involved, named as **E1-E4**, to address **C1**.

(E1): *[Conditional DNS (CDNS) without fallback mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
Execution: The reviewers should first change the terminal directory to the `test_infra` folder:

```
cd test_infra
```

After that, the Python script, `main_cdns.py`, should be executed:

```
sudo /path/to/conda/env/bin/python
    main_cdns.py
```

Notice that sudo privilege is required here since we need to control the Docker network infrastructure of the system. Replace "/path/to/conda/env/bin/python" with the path we got from the command `which python` in Section A.3.1.
Results: Once the testing is finished, you will get the results of testing in the structure:

```
./cdns_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

where `unit_no` refers to the number of a specific unit, `round_no` refers to the round number of a specific test, and `dns_sw_name` refers to the results of which DNS software (Bind9, Unbound, PowerDNS, Knot Resolver, MaraDNS or Technitium DNS Server) are stored in this folder. The detailed result folder structure should follow the one demonstrated in the Result structure section[14] in the `test_infra` folder.

(E2): *[CDNS with fallback mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
Execution: Still in the `test_infra` folder, `main_cdns_fallback.py` should be executed:

```
sudo /path/to/conda/env/bin/python
    main_cdns_fallback.py
```

Results: Once the testing is finished, you will get the results of testing in the structure:

```
./cdns_fallback_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

The result folder structure should also follow the one demonstrated in the Result structure section in the `test_infra` folder.

(E3): *[Forward-only mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
Execution: Still in the `test_infra` folder, `main_fwd_global.py` should be executed:

```
sudo /path/to/conda/env/bin/python
    main_fwd_global.py
```

Results: Once the testing is finished, you will get the results of testing in the structure:

```
./fwd_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

The result folder structure should also follow the one demonstrated in the Result structure section in the `test_infra` folder.

(E4): *[Recursive-only mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
Execution: Still in the `test_infra` folder, `main_recursive.py` should be executed:

```
sudo /path/to/conda/env/bin/python
    main_recursive.py
```

Results: Once the testing is finished, you will get the results of testing in the structure:

```
./recursive_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

The result folder structure should also follow the one demonstrated in the Result structure section in the `test_infra` folder.

A.5 Notes on Reusability

In README[15] in the `test_infra` folder, we demonstrated the customized settings of the testing infrastructure, including script arguments, IP address assignment, result structure, explanation of results, etc. We have also shared our implementation of the cache oracle and the resource consumption oracle in the `data_process`[16] folder. We hope this explanation will help the understanding and re-usability of RESOLVERFUZZ, in particular the testing infrastructure, better.

[14] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/test_infra#result-structure

[15] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/test_infra/README.md

[16] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/data_process

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.