USENIX'23 Artifact Appendix: UNCONTAINED: Uncovering Container Confusion in the Linux Kernel

Jakob Koschel†
Vrije Universiteit Amsterdam
j.koschel@vu.nl

Pietro Borrello†
Sapienza University of Rome
borrello@diag.uniroma1.it

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Herbert Bos
Vrije Universiteit Amsterdam
herbertb@cs.vu.nl

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

† Equal contribution joint first authors

A Artifact Appendix

A.1 Abstract

In this artifact we provide the means to reproduce our main results. Specifically, we show that our framework, UNCONTAINED, finds container confusion, both dynamically while fuzzing and statically with dataflow tracking. We have evaluated our artifact on an Ubuntu 22.04.1 (stock Linux kernel v.5.15) with 16 cores @2.3GHz (AMD EPYC 7643) using a total of 16 QEMU-KVM virtual machines with 4GB RAM. Our source code is available at: github.com/vusec/uncontained.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Since UNCONTAINED is only used for bug finding either statically or dynamically but running within VMs it does not impose any machine security, data privacy or other ethical concerns.

A.2.2 How to access

The files for the artifact evaluation are available at: https://github.com/vusec/uncontained/releases/tag/ae.

A.2.3 Hardware dependencies

UNCONTAINED does not impose any strict hardware requirements but we assume a recent x86_64 machine with enough RAM (minimum 64GB, or enough swap) to compile LLVM/Linux and run virtual QEMU machines for fuzzing with syzkaller.

A.2.4 Software dependencies

We expect certain packages from the Ubuntu package manager to be installed, which are required to compile LLVM, Linux, syzkaller, etc. We describe the necessary packages in the Set-up section.

If you use a different distribution you need to make sure to fulfil the necessary dependencies using your dedicated package manager.

A.2.5 Benchmarks

None.

A.3 Set-up

In general, we recommend using a bare-metal desktop system running Ubuntu 22.04. Make sure that you have KVM support and your user is allowed to use KVM. The following packages are required:

```
# go-task
sh -c "$(curl -ssL https://taskfile.dev/install.sh)" \
  -- -d -b ~/.local/bin
# llvm-project
sudo apt install build-essential clang-12 lld-12 ninja-build \
    ccache cmake
# linux
sudo apt install bison flex libelf-dev libssl-dev coccinelle
# syzkaller
sudo apt install debootstrap
# install golang 1.20.5
GO_VERSION=go1.20.5.linux-amd64
wget https://go.dev/dl/$GO_VERSION.tar.gz
sudo rm -rf /usr/local/go
sudo tar -C /usr/local -xzf $GO_VERSION.tar.gz
rm -f $GO_VERSION.tar.gz
# qemu
sudo apt install qemu-system-x86
# evaluation
pip3 install scipy pandas
```

Then make sure that `~/.local/bin` and `/usr/local/go/bin` are in your `PATH` to find `go` and the `task` binaries:

```
export PATH=$HOME/.local/bin:/usr/local/go/bin:$PATH
```

A.3.1 Installation

1. Obtain the artifact source and necessary dependencies:

```
git clone --recurse-submodules \
    https://github.com/vusec/uncontained.git
```

2. Create the `kernel-tools/.env` file with the following content (replace `/patch/to/uncontained` with the actual absolute path):

```
REPOS=/path/to/uncontained
LLVMPREFIX=/path/to/uncontained/llvm-project/build
KERNEL=/path/to/uncontained/linux
ENABLE_KASAN=1
ENABLE_DEBUG=1
ENABLE_SYZKALLER=1
ENABLE_GDB_BUILD=1
ADDITIONAL_LLVM_VARIABLES=-DLLVM_ENABLE_EH=ON -DLLVM_ENABLE_RTTI=ON
```

3. Compile all the necessary dependencies (this will take a while to compile llvm-project and Linux with fullLTO):

```
scripts/compile.sh
```

A.3.2 Basic Test

To test if the sanitizer and the static analyzers work as intended you can use the tests by running the following:

```
LLVM_DIR=$PWD/llvm-project/build tests/test.sh
LLVM_DIR=$PWD/llvm-project/build tests/testDF.sh
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The* UNCONTAINED *sanitizer finds new types of container confusions. This is proven by the experiment (E1).*

(C2): *The* UNCONTAINED *sanitizer comes with an acceptable performance runtime overhead. This is proven by the experiments (E2) and (E3).*

(C3): *The* UNCONTAINED *static analyzer has been used to uncover new bugs in the Linux kernel. This is proven by the experiments (E4).*

A.4.2 Experiments

(E1): *[fuzzing evaluation] [2 human-hours + 24 compute-hours]: This is the fuzzing experiment using the sanitizer while fuzzing with syzkaller. Expected results are a range of bugs reported.*
How to: *`kernel-tools` is responsible for starting the fuzzer with the kernel that has been instrumented with the sanitizer.*
Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the `./scripts/compile.sh` script).*
Execution: *You can compile the kernel with instrumentation and start the fuzzer with executing `./scripts/compile.sh && ./scripts/run.sh`. Then let it run for at least 24 hours to get some results.*
Results: *The result will be the crashes in the `kernel-tools/out/syzkaller-workdir/crashes` directory. We need to manually filter out bugs that are not triggered by* UNCONTAINED *(all that do not have three lines of `[UNCONTAINED]` before the `BUG:` line).*

(E2): *[2 human-hours + 30 compute-hours]: This is the fuzzing performance experiment using the sanitizer while fuzzing with syzkaller. Expected results are the overhead in terms of throughput of executed testcases.*
How to: *We need to run syzkaller 10 times for one hour for the baseline (stock syzkaller), with KASAN and with* UNCONTAINED.
Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the `./scripts/compile.sh` script).*
Execution: *You can compile the kernel with instrumentation and start the fuzzer with executing `./scripts/run-fuzzing-performance-evaluation.sh`. Then let it run for the 30 hours to get the results.*
Results: *The result will be the percentage of decreased executed testcases when running syzkaller. You can now look at the results with executing:*

```
./scripts/evaluation/syzkaller-bench.py --prefix \
    'evaluation/syzkaller/results/syzkaller-bench-'
```

(E3): *[1 human-hour + 1 compute-hour]: This is the LMBench experiment using the sanitizer while running the benchmarking suite to verify performance overhead.*
How to: *We need to run LMBench 10 times for the different configurations (baseline,* UNCONTAINED*, KASAN).*
Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the `./scripts/compile.sh` script).*
Execution: *You can compile the kernel with instrumentation and start LMBench with executing `./scripts/run-lmbench-performance-evaluation.sh`. Then let it run to get the results.*
Results: *The result will be the overhead numbers of the different configurations on top of the baseline for the LMBench testcases. You can now look at the results with executing:*

```
./scripts/evaluation/lmbench.py --prefix \
    'evaluation/lmbench/results'
```

(E4): *[1 human-hour + 3 compute-hours]: This is the static analyzers experiment using the static analyzer to find the necessary reports with static analysis.*
How to: *Compile the kernel with our static analyzers*

enabled to extract all the bug reports.

Preparation: *Make sure you setup everything from the Installation step, including building syzkaller and create the syzkaller image (should be done by the* `./scripts/compile.sh` *script).*

Execution: *You can generate all the reports with* `./scripts/run-static-analyzer.sh`*. Then let it run to get the results.*

Results: *The result will be the reports for the different rules. The results from the LLVM passes are in YAML and are not yet grouped by the source line (to remove duplicates). The results from the coccinelle script are text based and are already filtered based on uniqueness. You can load the YAML reports into the* `vscode-extension` *to look at them in a more convenient way and do the grouping based on the source code line.*

A.5 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.