USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The Cost of Performance: Breaking ThreadX with Kernel Object Masquerading Attacks

Xinhui Shao and Zhen Ling, *Southeast University;* Yue Zhang, *Drexel University;* Huaiyu Yan and Yumeng Wei, *Southeast University;* Lan Luo and Zixia Liu, *Anhui University of Technology;* Junzhou Luo, *Southeast University;* Xinwen Fu, *University of Massachusetts Lowell*

https://www.usenix.org/conference/usenixsecurity25/presentation/shao

This artifact appendix is included in the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium and appends to the paper of the same name that appears in the Proceedings of the 34th USENIX Security Symposium.

August 13–15, 2025 • Seattle, WA, USA

978-1-939133-52-6

Open access to the Artifact Appendices to the Proceedings of the 34th USENIX Security Symposium is sponsored by USENIX.

USENIX Security '25 Artifact Appendix: The Cost of Performance: Breaking ThreadX with Kernel Object Masquerading Attacks

Xinhui Shao[†], Zhen Ling[†,*], Yue Zhang[‡], Huaiyu Yan[†], Yumeng Wei[†], Lan Luo[¶], Zixia Liu[¶], Junzhou Luo[†], Xinwen Fu[§]
[†] Southeast University, Email: {xinhuishao, zhenling, huaiyu_yan, yumeng5, jluo}@seu.edu.cn
[‡] Drexel University, Email: zyueinfosec@gmail.com
[¶] Anhui University of Technology, Email: {lluo, zxliu}@ahut.edu.cn
[§] University of Massachusetts Lowell, Email: xinwen_fu@uml.edu

A Artifact Appendix

A.1 Abstract

In this artifact, we provide a repository of the source code for our symbolic execution engine and the Proof of Concept (PoC) experiment, along with the necessary materials to replicate the experiments described in the paper. To facilitate the evaluation process, we also offer corresponding Docker images, allowing users to set up the experiment environment and perform the related experiments with ease. This document outlines the steps to reproduce the results presented in Sections §6.2 and §A.2 of the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All experiments are conducted within a Docker container, except for one PoC experiment, which is performed on an actual board. The reproducer will need to execute several commands on the host machine during the experiment. These commands are commonly used and safe, ensuring they will not harm the system.

A.2.2 How to access

The code and experimental materials are available on Zenodo (DOI: `10.5281/zenodo.14643155`) and GitHub (`https://github.com/x-codingman/KOM-experiments.git`). Reproducers can either build the Docker images themselves, pull them directly from Docker Hub using the link provided in our GitHub repository, or download them from the Zenodo platform.

*Corresponding author: Prof. Zhen Ling of Southeast University, China.

A.2.3 Hardware dependencies

- **Processor**: We recommend using a machine with two Intel Xeon E5-2620 v2 CPUs (12 cores, 24 threads) to reproduce the experiment. However, comparable hardware may also suffice.
- **Memory**: At least 64GB of RAM.
- **Storage**: At least 256GB.
- **Board (Optional)**: NUCLEO-U575ZI-Q is used as one of the platform in the PoC experiment.

A.2.4 Software dependencies

Most experiments depend on Ubuntu 24.04, git, Qemu, and Docker. For the PoC involving the actual board, STM32CubeIDE is required to reproduce **E3**.

A.2.5 Benchmarks

Target system calls. We get the ThreadX source code (version 6.2.1) from GitHub. To facilitate the system call compilation, we modify the system call wrappers to generate the intermediate representation code. These modifications are referenced in the repository's **modification-ThreadX** file. To evaluate the symbolic execution engine, we provide the intermediate representation (IR) code, which has been compiled from the source code of ThreadX, in the GitHub repository.

A.3 Set-up

A.3.1 Installation

After cloning or downloading the repository, please refer to the "Preparation" section of the `README` files in the corresponding subdirectories (i.e., `symbolic-execution-engine` and `Proof-of-Concept`). All experiments should be conducted within the Docker

image. If you are unable to build the image, we also provide a pre-built Docker image available on both Docker Hub and Zenodo.

A.3.2 Basic Test

If the Docker image is built or downloaded successfully, you can run `./run_docker.sh` in the repository, which will spawn a shell within the container on success. Note that the container will continue running after you exit the shell. You can reattach to the container,or spawn a new shell to create a new container by running `./run_docker.sh` again later. Please be aware that the `./run_docker.sh` command will delete the existing container and create a new one.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Vulnerable system call Detection. Using the symbolic execution engine, we can identify the modification fields and path constraints of the target system calls, enabling us to further determine the vulnerable system calls. This is proven by the experiment (**E1**) whose results are illustrated in §6.2 (**RQ1**, Table 2 and Table 3).

(C2): Efficiency. The symbolic execution engine is efficient. This is proven by the experiment (**E2**) whose results are illustrated in §6.2 (**RQ4**, Table 2).

(C3): Attack Implications. KOM can be launched on various platforms. This is proven by the experiment (**E3**) whose results are illustrated in §6.2 (**RQ5**, Table 6).

A.4.2 Experiments

Please refer to the "Replicating Our Experiments" section of the `README` file in the repository for detailed steps to reproduce our experiments.

We design experiments (**E1-E2**) for the symbolic execution engine to confirm **C1** and **C2**. We design PoCs (**E3**) to confirm **C3**.

(E1): *[Vulnerable System Call Detection.] [5 human-minutes + 10 compute-hours]:* The vulnerable system calls can be evaluated by performing symbolic execution on the system calls of ThreadX. Please refer to the "(E1) Vulnerable System Call Detection" section of the `README` file in the `symbolic-execution-engine` folder of the repository. The results can be found in `M[1,2,3]_vulnerable_system_calls.xlsx`. **C1** is confirmed if the acquired results are consistent with those in Table 2 and Table 3.

(E2): *[Efficiency.] [2 human-minutes + 1 compute-minute]:* The efficiency of the symbolic execution engine is evaluated by checking the execution results (e.g., the consumed time and the number of executed instructions). This experiment should be conducted after **E1**. Please refer to the "(E2) Efficiency" section of the `README` file in the `symbolic-execution-engine` folder of the repository. The results can be found in `symbolic-execution-run-time-evaluation.xlsx`. **C2** is confirmed if the acquired results are comparable with those in Table 2.

(E3): *[Attack Implications] [5 human-minutes + 10 compute-minutes]:* The attack implications are evaluated by performing the PoC experiments on various platforms. Please refer to the `README` file in the `Proof-of-Concept` folder of the repository. **C3** is confirmed if these PoC experiments can be successfully executed.

A.5 Notes on Reusability

The under-constrained symbolic execution engine is designed to identify vulnerable system calls in ThreadX, specifically for KOM attacks. Currently, this tool does not support the latest version of ThreadX or other RTOSs.

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/usenixsec2025/.