USENIX Security '24 Artifact Appendix: A Formal Analysis of SCTP: Attack Synthesis and Patch Verification

Jacob Ginesin
ginesin.j@northeastern.edu
Northeastern University

Max von Hippel
vonhippel.m@northeastern.edu
Northeastern University

Evan Defloor
defloor.e@northeastern.edu
Northeastern University

Cristina Nita-Rotaru
c.nitarotaru@northeastern.edu
Northeastern University

Michael Tüxen
tuexen@fh-muenster.de
FH Münster

A Artifact Appendix

A.1 Abstract

SCTP is a transport protocol offering features such as multi-homing, multi-streaming, and message-oriented delivery. A recent vulnerability, reported in CVE-2021-3772, showed SCTP is not immune to attacks. The protocol was patched to address the vulnerability in RFC 9260. In this artifact, we formally model SCTP with and without the patch, and specify ten Linear Temporal Logic (LTL) correctness properties for it in PROMELA. Using SPIN, we show that in the absence of an attack, the protocol satisfies all ten correctness properties, with or without the patch. We then define four attacker models, specifying the placement and capabilities of an attacker: Off-Path, On-Path, Evil-Server, and Replay. We modify the attack synthesis tool KORG to support our SCTP model and use it to show that the CVE attack can be automatically synthesized in the version of the model without the patch, but is not possible once the patch is applied. In the four attacker models we study and across the ten properties we define, we find that the patch does not introduce any new vulnerabilities. We do not find any other Off-Path attacks besides the one reported in the CVE, and although we do find attacks in the other attacker models, SCTP was not designed to be secure in these, so we only report them to understand how the security of SCTP degrades for successively more powerful attacker types. Finally, we identify two ambiguities in the RFC 9260 text. For each one, consulting with the lead SCTP RFC author, we determine the correct interpretation, and then model the incorrect version and attack it with KORG. We find that both enable vulnerabilities if misinterpreted and we suggest a textual fix for each in the form of RFC errata, of which one was accepted so far. Accordingly, the artifact consists of the SCTP model (with a boolean to enable or disable the patch), ten LTL correctness properties, four attacker models, modifications we made to KORG to support SCTP, and scripts to reproduce our results.

A.2 Description & Requirements

Our software has been tested and confirmed to run (and reproduce our results) on a 16GB M1 Macbook Air (natively), a 36GB M3 Macbook Pro (in Docker, in an image with 16GB of RAM), and on two comparable Intel Linux machines each with >16GB of RAM (running Linux Mint and Arch Linux, respectively). Generally, the results should be easily reproducible on any modern Linux-based laptop with at least 16GB of RAM.

A.2.1 Security, privacy, and ethical concerns

There are no risks to running the code.

A.2.2 How to access

The code is available at `https://github.com/sctpfm/sctpfm/tree/usenix`. The officially submitted version is git commit `1a0d9e5333b9c3e21de3d1682855cb2f8f1b8aa3`.

A.2.3 Hardware dependencies

The the code requires at least 16GB of RAM. If the code is run inside the Docker image, the image should be configured with 16GB of RAM.

A.2.4 Software dependencies

The code is most easily reproducible using Docker. We provide a Dockerfile which will build an environment in which our results can be easily reproduced. However, you can also build and run the software by repeating the steps outlined in the Dockerfile on your local machine.

A.2.5 Benchmarks

None.

A.3 Set-up

To run using Docker, you need to install Docker on your machine, then build the included Dockerfile. The entrypoint for the Dockerfile is bash, which you can use to interact with the included Makefile. The Makefile has targets to reproduce each result.

A.3.1 Installation

Simply clone the repository and build the Dockerfile.

A.3.2 Basic Test

Upon building the Dockerfile, e.g., `docker build . -t sctpfm`, run it, e.g., `docker run -it sctpfm`, and then in bash inside the Docker image, run `make sctpOffPath`. This will reproduce our Off-Path results. Once it terminates, you can `ls out` to find folders with the output attacks.

A.4 Evaluation Workflow

Once you've built the Docker image and entered it, use the following Makefile targets.

`sctpOffPath`: Generates the Off-Path attacks for all 10 properties, with and without the patch.

`sctpEvilServer`: Generates the Evil-Server attacks for all 10 properties, with and without the patch.

`sctpOnPath`: Generates the On-path attacks for all 10 properties, with and without the patch.

`sctpReplay`: Generates the Replay attacks for all 10 properties, with and without the patch.

In addition, you can generate attacks against the first ambiguity by running:

```
python3 korg/Korg.py \
    --model=demo/SCTP/ambiguity1/SCTP-9260.pml \
    --phi=demo/SCTP/ambiguity1/phi.pml \
    --Q=demo/SCTP/ambiguity1/Q.pml \
    --IO=demo/SCTP/ambiguity1/IO.txt \
    --max_attacks=10 \
    --with_recovery=true \
    --name=ambiguity1 \
    --characterize=false
```

or for the second by running:

```
python3 korg/Korg.py \
    --model=demo/SCTP/ambiguity2/SCTP-9260.pml \
    --phi=demo/SCTP/ambiguity2/phi.pml \
    --Q=demo/SCTP/ambiguity2/Q.pml \
    --IO=demo/SCTP/ambiguity2/IO.txt \
    --max_attacks=10 \
    --with_recovery=true \
    --name=ambiguity2 \
    --characterize=false
```

Once you've produced all the attacks you can analyze them by looking at the resulting attacker code saved in `out` inside the Docker image.

A.4.1 Major Claims

(C1): We find the CVE attack using the Off-Path model when the patch is disabled. We do not find any other Off-Path attacks with or without the patch.

(C2): When the patch is enabled, no attacks are found which were not found with it disabled.

(C3): We find an attack with each ambiguity.

A.4.2 Experiments

Running all of the targets in the Makefile, inside the Docker image, should reproduce the results saved in the `results` directory of our artifacts (or produce equivalent outputs). Doing this from start to finish might take 24 hours. Note, the targets cannot be run simultaneously; you must wait until one finishes before running the next. This is for two reasons. First, SPIN creates intermediary files, and if two SPIN instances are run at once in the same folder, one can gobble the files created by the other, leading to crashes or unsound results. And second, model checking involves constructing and storing a large state space in memory, an operation that is inherently difficult and out of the scope of this project to parallelize.

A.5 Notes on Reusability

The improvements we made to KORG, included in the `korg-changes` folder, will be merged into the main KORG codebase in a pull request following the publication of this paper. KORG has been used in multiple research projects and so this contribution helps improve its functionality for future work.

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/.