ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

USENIX Security '25 Artifact Appendix: Await() a Second: Evading Control Flow Integrity by Hijacking C++ Coroutines

Marcos Bajo
CISPA Helmholtz Center for Information Security

Christian Rossow
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This document details the artifacts available for reproducing the results presented in our paper. We explain 1) how to obtain our artifacts and navigate them; 2) the setup required for reproducing our results; and 3) how to run the PoCs and exploits described in the paper.

Our artifact collection consists of four PoC CFOP exploits on sample programs:

- *gcc*: showcases how to call arbitrary functions with arbitrary arguments using a *silver gadget* in a program compiled with GCC.
- *clang*: showcases how to call arbitrary functions with arbitrary arguments using a *silver gadget* in a program compiled with Clang/LLVM.
- *msvc*: showcases how to call arbitrary functions with arbitrary arguments in Windows, compiled with MSVC.
- *doa*: showcases how to leverage a Data Only Attack (DOA) for altering the program execution without hijacking any frame pointers, reading from an arbitrary file and compiled with GCC.

In addition, we showcase two CFOP exploits in real-world programs:

- *vulnerable_serenityos*: an exploit for SerenityOS after reintroducing the CVE-2021-4327 vulnerability. Showcases Infinite Coroutine Chaining (ICC).
- *vulnerable_scylladb*: an exploit for ScyllaDB with an incorporated vulnerability and a modification in the database client to exploit it. Showcases *golden gadgets* to call arbitrary functions with arbitrary arguments.

Every artifact program has been compiled with Intel CET (and Control Flow Guard (CFG) for *msvc*). Therefore, the exploits work while considering the restrictions introduced by these schemes: 1) return addresses in the stack can never be modified, and 2) the control flow cannot be redirected to any address that is not the start of a function.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifacts do not pose any security, privacy, or ethical risks for evaluators while executing them. The provided exploits are limited to the programs we prepared and do not perform any destructive actions.

A.2.2 How to access

We provide access to every previously described artifact in our in our Zenodo record (*10.5281/zenodo.14738036*) and our *GitHub repository*. The contained README files provide an overview of the artifacts and how to run them.

A.2.3 Hardware dependencies

Our artifacts are prepared to run on standard x86-64 machines and do not require any special hardware features. The exploits are designed to circumvent Intel CET and CFG restrictions and can be tested on any modern Windows/Linux machine (see A.2.4); we do not require evaluators to run them on a machine with CET support.

Optionally running the artifacts with an *active* enforcement of these schemes requires additional hardware support. Specifically, evaluators need a CPU with Intel CET support, that is, Intel Tiger Lake (11th gen) or newer. We recommend a machine with at least 8GB of RAM to compile ScyllaDB and SerenityOS smoothly; we used a machine with 32GB of RAM and a 20-core i9-12900H CPU for our experiments. Around 50 GB of free disk space are needed in total throughout the experiments.

For the purpose of evaluating the artifacts in an environment that meets every hardware and software requirement (including the optional *active* CET enforcement), we provide access to our testing machine via SSH. This machine is restricted for evaluators during the USENIX artifact evaluation period and will be disabled after the evaluation. We incorporate to this artifact submission a WireGuard VPN configuration file to access the machine. The file `AECONNECT.md` in the repository provides instructions on how to connect to the machine.

A.2.4 Software dependencies

The artifacts have been tested Ubuntu 24.04 and Windows 11 (only for *msvc*).

Every exploit is prepared to run inside a Docker container and can be tested on any Linux distribution with Docker installed; specific software requirements for every artifact are automatically met inside each container. The exception is the Windows *msvc* exploit, that requires a Windows 10 or newer machine with MSVC installed.

A.2.5 Benchmarks

None.

A.3 Set-up

Every artifact in our collection includes its own README file (six in total) with detailed instructions on how to set up their own environment and run the corresponding exploit.

A.3.1 Installation

Docker is required to run the artifacts. For the Windows *msvc* exploit, Python3 is also required. Since each of the six artifacts are independent, here we detail the process of running the *gcc* exploit as an example.

We recommend running the artifacts in our experimental machine. Alternatively, to kickstart this evaluation, clone the repository and install Docker in any Linux machine.

A.3.2 Basic Test

1. Build the docker image:
   ```
   docker build -t cfop_gcc_poc .
   ```
2. Run the docker container
   ```
   docker run -privileged -security-opt
   seccomp=unconfined -it cfop_gcc_poc
   ```
3. Once inside the container, navigate to where the PoC files are saved
   ```
   cd opt/pocs/
   ```
4. Compile the coroutine program
   ```
   make
   ```
5. Run the exploit script
   ```
   python3 gccpocexploit.py
   ```

A successful exploitation results in the program printing *"ARBITRARY CODE EXECUTION!"* on screen, even when this code was not present in the original program.

A.4 Evaluation workflow

A.4.1 Major Claims

Our major claims are:

(C1): CFOP techniques can be used to exploit programs compiled with the latest versions of the three major compilers: GCC, Clang/LLVM, and MSVC.

(C2): Exploits using CFOP work while considering the restrictions introduced by Intel CET and Control Flow Guard CFI schemes: 1) return addresses in the stack can never be modified, and 2) control flow cannot be redirected to any address that is not the start of a function.

A.4.2 Experiments

We recommend following the steps detailed in the README of each artifact to reproduce the results of the paper.

(E1): *[gcc]* [5 human-mins + 1 compute-mins + 4GB disk]:
Preparation: Follow steps 1 to 4 at A.3.2.
Execution: Run the exploit script using `python3 gccpocexploit.py`.
Results: The program should print *"ARBITRARY CODE EXECUTION!"* on screen.

(E2): *[clang]* [5 human-mins + 1 compute-mins + 4GB disk]:
Preparation: Build the Docker image using `docker build -t cfop_clang_poc .`; run the container using `docker run -privileged -security-opt seccomp=unconfined -it cfop_clang_poc`; navigate to the PoC files using `cd opt/pocs/`; and compile the coroutine program using `make`.
Execution: Run the exploit script using `python3 clangpocexploit.py`.
Results: The program should print *"ARBITRARY CODE EXECUTION!"* on screen.

(E3): *[msvc]* [5 human-mins + 1 compute-mins + 1GB disk]:
Preparation: Acquire a Windows 10 or Windows 11 system. Install Python3. If necessary, update the path where the *.exe* file is located in line 10 of the scripts *msvc_silver_exploit.py* and *msvc_call_exploit.py*.
Execution: Run the first exploit using `python3 msvc_call_exploit.py`. Then, run the second exploit using `python3 msvc_silver_exploit.py`.
Results: For the first exploit, the calculator program (*calc.exe*) gets executed. For the second exploit, the parameters values are printed on screen with clearly arbitrary values (e.g., 0x4242424242424242).

(E4): *[doa]* [5 human-mins + 1 compute-mins + 4GB disk]:
Preparation: Build the Docker image using `docker build -t cfop_doa_poc .`; run the container using `docker run -privileged -security-opt seccomp=unconfined -it cfop_doa_poc`; navigate to the PoC files using `cd opt/pocs/`; and compile the coroutine program using `make`.

Execution: Run the exploit script using `python3 fileopening.py`.
Results: The program prints the contents of the file */etc/hosts* on screen, even when this filename was not present in the original program.

(E5): *[vulnerable_serenityos]* [10 human-mins + 30 compute-mins + 22GB disk]:
Preparation: We release the modified source code responsible of building SerenityOS with the CVE-2021-4327 vulnerability. With the goal of simplifying testing, we build and run the Ladybird inside a Ubuntu 24.04 docker - for which we provide the corresponding Dockerfile. If not using our experimental machine, since Ladybird is a program with a GUI, we recommend installing X-Server or a similar system in the host system. Our Docker container takes care of the rest of dependencies.
Execution: Build the Docker image using `sudo docker build -t vulnerable-serenityos .`; run the docker container and attach to it (this command is long and can be found in the README file); once in the container, you can build and run Ladybird `./Meta/serenity.sh run lagom ladybird`;
Once Ladybird is running, the GUI will be shown in screen. At this moment, we can navigate to the URL *file:///cfop/Base/home/anon/exploit.html*. Upon visiting this website, the command line will leak some address. This address is needed to be entered in the browser input as to run the exploit.
Results: As a result of running the exploit, the name of the current user will be printed on screen three times (executing *execve("whoami")*).

(E6): *[vulnerable_scylladb]* [10 human-mins + 40 compute-mins + 34GB disk]:
Preparation: ScyllaDB is made of numerous submodules - hosted in multiple repositories. In addition, its internal build system depends on such submodules for querying additional files at runtime, so it is not possible to offload all the code of ScyllaDB to one single repository. For this reason, our build system consists of 1) A *cfop_setup.sh* script, that clones a certain version of ScyllaDB from its official GitHub repository, patches the code with our vulnerability and exploit, and then builds ScyllaDB; and 2) a folder *cfop_mods* that incorporates the files that patch scyllaDB.
A particular OS version is not needed to run ScyllaDB; every dependency is included in its internal docker setup.
Execution: Run the build script *cfop_setup.sh*. As a result, you will have a new repository *vulnerable_scylladb*; compile ScyllaDB inside the docker container (command can be found in the README, as it is long); attach to the docker container using `podman container attach <id>`; run ScyllaDB from inside the container using `./build/release/scylla -workdir tmp -smp 8 -memory 4G -developer-mode=1`;
Once the ScyllaDB instance is running, we can launch the client program and trigger the exploit: `python3 tools/cqlsh/bin/cqlsh.py`; after it asks for input, introduce `EXPLOITPAYLOAD;`.
Results: Upon sending the previous input, the ScyllaDB instance will stop and show the current user's name on screen (as it runs *execve("bin/sh", "-c /usr/bin/whoami")*).

A.5 Notes on Reusability

The exploits we provide are representative of the attacks describe in the paper and showcase every Coroutine Frame Oriented Programming (CFOP) technique. We intend that these exploits serve as a starting point for replicating the attacks in other programs.

Here we detail which exploits showcase which CFOP technique, serving as a reference for future testers. We recommend consulting the corresponding exploit code to help develop exploits for other coroutine programs:

- *gcc*: Silver gadgets for arbitrary calls with multiple arbitrary arguments in GCC.
- *clang*: Silver gadgets for arbitrary calls with multiple arbitrary arguments in Clang/LLVM.
- *msvc*: Silver gadgets for arbitrary calls with multiple arbitrary arguments in MSVC.
- *doa*: Data Only Attack (DOA) for altering the program execution without hijacking any frame pointers.
- *vulnerable_serenityos*: Infinite Coroutine Chaining (ICC) for multiple arbitrary calls.
- *vulnerable_scylladb*: Golden Gadgets for arbitrary calls with arbitrary arguments.

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/.