Artifact Guide: Building Dynamic System Call Sandbox With Partial Order Analysis

This document serves as a guide for installing and using DynBox, a dynamic system call sandbox designed to isolate vulnerable programs. It is the accompanying artifact for the OOPSLA 2023 submission titled Building Dynamic System Call Sandbox with Partial Order Analysis. Within this guide, we provide a detailed description of how to reproduce the data presented in our **Evaluation** section, specifically Table 2, Table 3, Table 4, and Table 5.

Research Questions

- **RQ1**. How effective is DynBox in reducing the attack surface and protecting the operating system from malicious payloads?
    - The artifact answers the question by demonstrating that DynBox allows for minimal unnecessary syscalls and offers strong resistance against malicious payloads. For more detailed information, please refer to Table 2 and Table 3.
    - The artifact demonstrates that DynBox outperforms three state-of-the-art syscall sandboxes.
- **RQ2**. Which syscalls are restricted by DynBox, and what is the impact of these syscalls in vulnerability exploitation?
    - The artifact answers these research questions. As Table 4 shows, DynBox successfully restricts numerous critical syscalls that are commonly exploited by attackers to target vulnerabilities.
- **RQ3**. What are the size expansion, analysis time, and runtime overhead of DynBox?
    - The artifact addresses the question. Table 5 of the artifact showcases that DynBox does not impose significant runtime overhead nor lead to substantial binary size expansion.
    -

Requirements

The artifact utilizes Docker to build reproducible environments. Therefore, Docker is a prerequisite and can be installed by following the official installation instructions. We recommend a minimum of 16 GB of memory and 8 GB of hard disk space for machines.

Quick Evaluation

Before proceeding, please extract the file first. In addition to the `readme` file, it includes a `Dockerfile` as well as two directories named `DynBox` and `outputs`.

Docker Build

To build docker, please run the following command.

```
docker build . -t dynbox
```

Next, please start a container and launch a shell using the constructed image.

```
docker run -it -v ./outputs:/DynBox/outputs  --name dynbox  dynbox
```

The working path in image is `\DynBox`, which contains the source code and corresponding resources for artifact evaluation. Meanwhile, we mount the `outputs` directory into the container with path `\DynBox\outputs`, where the evaluation results are stored. Therefore, we can analyze the results in the `./outputs` directory on host machine.

Evaluation

The evaluation is performed within the Docker container, so please start the container and launch a shell first with commands in **Docker Build** Section.

All evaluation results can be produced with one command:

```
./runAll.sh
```

After execution, all output are stored in `./outputs` directory. The generated CSV tables can be found in the ./outputs/tables directory.

Table 2 & Table 3

Table 2 and Table 3 are shown in the Section 5.1, **Evaluation of Effectiveness**. In this section, we evaluate the effectiveness of DynBox in restricting the unnecessary syscalls and protecting the operating system from malicious payloads. The primary metrics include the number of permitted syscalls and the defense rate against malicious payloads.

Table 2 showcases the results obtained from DynBox, as well as two state-of-the-art syscall sandboxes, namely Chesnut and Temporal Specialization (Temp). Table 3 presents the comparative results between DynBox and C2C, a configuration-aware syscall sandbox, whose outcomes are dependent on the specific configuration of the application. Through these two tables, we demonstrate that DynBox can restrict more unnecessary syscalls and achieve better defense performance.

Table 4

Table 4 is shown in Section 5.2, **Syscall Analysis**, which analyzes how DynBox restricts critical syscalls. In the table, each row represents a critical syscall. The "Payload" column shows the proportion of payloads using the corresponding syscall. The number under each application indicates the restriction level of the corresponding syscall at all vulnerabilities' exploitation positions of the application. 1.00” indicates that syscall is forbidden on all vulnerability exploitation positions of an application, and “0.00” means none.

Table 5

Table 5 is presented in Section 5.3, Overhead, illustrating the runtime overhead, binary size expansion, and analysis time of DynBox.

Detailed Evaluation Steps

Run Partial Order Analysis

We write the partial order analysis as a LLVM pass, located at the `llvm/PartialOrderAnalysis.so`. To run the analysis, we should use the `opt` tool from llvm, which is installed in `llvm/llvm-12`. Then, we run the partial order analysis to build DynBox for each application with script `runDynBox.sh`.

```
./dynbox/runDynBox.sh
```

In the script, we utilize the `opt` tool to analyze LLVM's `bc` file of each application and perform instrumentation on them. These `bc` files are generated by LLVM during the link time optimization process. The instrumented `bc` files are stored in `targets/bcFiles/*.seccomp.bc`. Additionally, the script calculates the permitted syscalls for each exploitation position within the application. The results are outputted to the `./outputs/DynBox/*-cve.json` files.

To build sandbox for one application, please use the following command.

```
python3.8 dynbox/buildDybBox.py -t sqlite
```

The command will load the configuration from `/DynBox/dynbox/config.json`, configure the environment accordingly, and execute the analysis.

Evaluate DynBox

Using the permitted syscall output obtained from the partial order analysis, we can further measure the defense rate and average permitted syscalls of each application by executing the following command.

```
python3.8 evaluation/evaluate.py -d outputs/DynBox -t all
```

Here, the `-d` option should be provided with the output directory that contains the permitted call of each application. Please use the `-t` to specify the target application of evaluation. When evaluating all applications, use all as the target. To evaluate one application individually, please execute the following command.

```
python3.8 evaluation/evaluate.py -d outputs/DynBox -t nginx
```

To analyze the critical syscalls, please run the script `evaluate_syscall.py` with the following command:

```
python3.8 evaluation/evaluate_syscalls.py -d ./outputs
```

Similarly, by specifying the `-t` option, we can calculate the results for a specific application.

```
python3.8 evaluation/evaluate_syscalls.py -d ./outputs/ -t nginx
```

Evaluate Other Syscall Sandboxes

In our paper, we compare three state-of-the-art syscall sandboxes, which are stored in `/DynBox/others` directory. To evaluate them, please follow the steps below.

Chesnut

To evaluate Chesnut, please navigate to the folder `/DynBox/others/Chesnut` first. Then, we can run `evaluate_chesnut.py` to evaluate it.

```
python3.8 evaluate_c2c.py -s ./permittedSyscall -r ../../outputs -t all
```

We still utilize `-t` option to indicate the target application. The corresponding results are stored in `./outputs/chesnut`.

Temporal Specialization

We place the codes of Temporal Specialization (Temp) in `/DynBox/others/Temporal-Specialization`. Please get in this directory before running scripts. Temp is evaluated with the following command.

```
./runTemp.sh
```

The results are stored in `./outputs/temp`.

C2C

The evaluation of C2C is performed in the `/DynBox/others/c2c` directory. Please execute the `evaluate_chesnut.py` script to evaluate C2C.

```
python3.8 evaluate_chesnut.py -s ./permitted_syscalls -t all -r ../../outputs/
```

We can find the corresponding results in `outputs/C2C`.

Generate Tables

The scripts used for generating tables are located at `/DynBox/tables/`. These scripts collect the outputs of DynBox and other sandboxes to generate tables. Therefore, before generating each table, please ensure that all syscall sandboxes have undergone evaluation and their results have been stored in the appropriate directories. To generate Table 2, please run the following command.

```
cd /DynBox/tables/ && python3.8 drawTable2.py
```

As for Table 3, Table 4 and Table 5, we can generate them with similar commands as mentioned above. The generated tables can be found in the `outputs/tables` directory.

Adopting DynBox to A New Program

To construct DynBox for a new program, the process consists of four essential steps.

Compiling Program With LTO Enabled

In order to develop DynBox for a new program, an inter-procedural static analysis of the entire program is required. This analysis can only be carried out during the Link Time Optimization (LTO) phase of LLVM. To accomplish this, the program must be built with LTO enabled and a `bc` (bytecode) file containing the entire program must be generated. To accomplish this, during the compilation process, it is necessary to utilize LLVM as the compiler and append the `-flto` flag to the `CFLAG`. Additionally, the `-Wl,-plugin-opt=save-temps` should be added to the `LDFlag`.

Indirect Call Analysis

Subsequently, we rely on the Static Value-Flow Analysis Framework (SVF) to address the issue of Indirect Calls. To utilize SVF effectively, please refer to the instructions provided in its original repository. With SVF, the resolution of indirect calls is facilitated through the usage of its `wpa` tool, which can be found in its installation path. The command to execute this tool is as follows:

```
wpa -print-fp -ander -dump-callgraph /path/to/bc/file
```

Dynamic Libraries Analysis

Next, DynBox needs the analysis of all dynamic libraries that the target program relies on. To accomplish this task efficiently, we utilize the `angr` tool for binary analysis. We have provided a script to facilitate this step, and the command is as follows:

```
python dso_process/ExtractLibraryCalls.py -b /path/to/binary    -o
./dso_process/mappings -a -c ./dso_process/cfg -s ./dso_process/mappings
```

By executing the provided script, we can obtain the syscalls that utilized by each interface of dynamic libraries. The result is stored in the `./dso_process/mappings` and is named after the target program.

DynBox Construction

Upon completing the aforementioned analyses, we could implement the partial order analysis to construct DynBox. Before the construction, the paths to the results obtained from the preceding steps should be spicified in `dynbox/config.json`. For each program, the `config.json` file should include five items:

1. The `bcfile` item should be the path to the target program's `bc` file that is generated during the Link Time Optimization (LTO) process.
2. The `DynLibMapping` item specifies the location of the dynamic libraries analysis results.
3. The `IndirectCalls` item is set to the path where the results of the indirect call resolution are stored.
4. The `vulnerabilities` item indicates the path to the manually gathered vulnerability information. This item is solely utilized for evaluation purposes and is not necessary for the construction of DynBox.
5. The `evaluationOutput` item designates the path where the evaluation results should be stored. This item is also used for evaluation only.

By having a properly configured `config.json` file and the corresponding preprocessing results in place, we can successfully construct DynBox for a new program.

Artifact Structure

This section gives a brief overview of the files in this artifact.

- `dsoProcess/`: scripts for analyzing the mapping from interfaces of dynamic libraries to their utilized syscalls.
- `dynbox/`: scripts and configs for running partial order analysis.
- `evaluation/`: scripts for perform evaluation of DynBox.
- `indirectCalls/`: scripts for resolving indirect calls in applications.
- `llvm/`: directory to store LLVM tool and partial order analysis.
- `others/`: scripts for evaluating Chesnut, Temp, and C2C.
- `syscallProcess/`: scripts for process system call and their id.
- `tables/`: scripts for generating tables.
- `targets/`: binaries and `bc` files for target applications.
- `critical_syscalls`: list of critical syscalls.