Artifact Appendix: Introduction to Procedural Debugging through Binary Libification

Jonathan Brossard
Conservatoire National des Arts et Metiers, Paris

A Artifact Appendix

A.1 Abstract

The Witchcraft Compiler Collection (WCC) is a reverse engineering framework aimed at manipulating ELF binaries, published under a dual MIT/BSD permissive open-source license. The present article focuses on the wld tool of this framework, named the Witchcraft linker: wld.

This tool aims at libifying dynamically linked ELF executables, meaning transforming them into shared libraries, that may, in turn, be used and, for instance, loaded within executable programs via calls to the dlopen() function of dynamic linkers.

Unlike control-flow-based reverse engineering techniques such as disassembly and decompilation, libification does not involve solving undecidable problems, as seen in figure 1 representing libification as a reverse engineering technique. Libification merely relies on modifying metadata within ELF binaries, leaving .data and .text sections untouched. As such, libification is expected to be both faster and more reliable.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The Witchcraft linker, wld, modifies its input ELF binary files. This may render them unusable.

While libification itself is a relatively safe operation that does not involve running untrusted code, loading shared libraries in memory is inherently an unsafe operation when said libraries come from untrusted parties. It is worth noting, in particular, that the dlopen() function may run constructors, even if no library call to the shared library is made. Dynamically linking against libified binaries exposes to the same risks. As such, libifying and loading unknown or possibly hostile code such as malware is not advised.

Since libification modifies compiled programs, it may infringe on the intellectual property of proprietary software. Users are advised to refer to their End User License Agreements on a case-by-case basis.

A.2.2 How to access

The source code of the Witchcraft Compiler Collection is available from https://github.com/endrazine/wcc under a dual MIT/BSD license.

The specific version of the codebase tested in this article, tagged v0.0.6 and release "woot24", is available from the following DOI: https://zenodo.org/doi/10.5281/zenodo.11298208.

A test repository has been created at the address https://github.com/endrazine/wcc-tests/releases/tag/WOOT24 to evaluate the Witchcraft linker, featuring the following DOI : https://zenodo.org/doi/10.5281/zenodo.11301408.

A.2.3 Hardware dependencies

To facilitate the evaluation, all tests have been dockerized. Evaluation should be performed on an AMD or Intel-compatible 64-bits CPU, powerful enough to run Docker images. As such, a CPU i5 or better of generation 10 or more recent is recommended. The image itself shall be under 4 GB in size ; hence, that much disk space will be required. A machine featuring 16 GB of RAM or more is recommended.

A.2.4 Software dependencies

To facilitate the evaluation, all tests have been dockerized. As such, installing Docker itself on a GNU/Linux test machine is compulsory. Alternative host Operating Systems have not been tested. The author used Docker version 26.1.1, build 4cf5afa.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

Instructions to install Docker are available from https://docs.docker.com/engine/install/.

Once Docker is installed on a 64bits GNU/Linux Intel compatible machine, a Dockerfile may be downloaded from https://zenodo.org/doi/10.5281/zenodo.11301408.

To build the test Docker image, one may copy this Dockerfile to /tmp/woot24/Dockerfile, and then from a terminal, enter the /tmp/woot24 directory and type:

```
docker build . -t witchcraft:latest
```

This should produce a new image named witchcraft:latest. This image should be visible near the top when entering the following command from the terminal:

```
docker images
```

One can then run and enter the test container by running the following command from the same terminal:

```
docker run -it witchcraft:latest
```

A.3.2 Basic Test

Provided the above commands worked properly, typing the command "ls" followed by the "enter" key from the terminal should list the following content within the test container:

```
Makefile README.md chroot-test loader loader.c
scripts test_all
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Copying, libifying, and loading the 435 executables in the default PATH of a Linux Ubuntu 22.04 LTS 64bits is practical and takes less than 3 seconds in total.

(C2): The following complex binaries can be libified: OpenSSH server, Apache2, Nginx, GCC. Libification takes less than 0.01 seconds.

(C3): The Google Chrome web browser can be libified. Libification takes less than 0.01 seconds.

A.4.2 Experiments

(E1): [Ubuntu 22.04] [1 human-minute + 1 compute-minute]: Libify via wld and load via dlopen() the 435 binaries in the default PATH of a Ubuntu 22.04 LTS distribution.
How to: Testing the libification of an entire Ubuntu 22.04 LTS distribution is facilitated by the test Docker image detailed in chapter A.3.
Preparation: Enter the Docker test image as described in chapter A.3 by running the command:

```
docker run -it witchcraft:latest
```

Execution: From the terminal prompt, type:

```
time make test
```

Results: Execution shall produce no error messages, and run under 3 seconds.

(E2): [Complex binaries] [5 human-minutes + 5 compute-minutes]: Libify via wld and load via dlopen() the complex binaries OpenSSH server, Apache2, Nginx, GCC.
How to: Testing the libification of this complex set of binaries in a replicable fashion is facilitated by the test Docker image detailed in chapter A.3.
Preparation: Enter the Docker test image as described in chapter A.3 by running the command:

```
docker run -it witchcraft:latest
```

Then install the required packages to be libified:

```
apt install -y openssh-server apache2 \
nginx gcc
```

Execution: From the terminal prompt, enter the following command:

```
for testbin in /usr/sbin/sshd /usr/sbin/apache2 \
/usr/sbin/nginx ; do cp ${testbin} ./test ; \
time wld -libify -noinit ./test ; \
./loader ./test ; done
```

Results: Libification and loading shall produce no error messages. Lbification of each binary shall take less than 0.01 seconds. The following message shall be displayed exactly three times:

```
Loading of Library successful
```

(E3): [Chrome] [5 human-minutes + 5 compute-minutes]: Libify via wld and load via dlopen() the Google Chrome web browser.
How to: Testing the libification of Google Chrome in a replicable fashion is facilitated by the test Docker image detailed in chapter A.3.
Preparation: Enter the Docker test image as described in chapter A.3 by running the command:

```
docker run -it witchcraft:latest
```

Chrome has some dependencies that must be installed before the Chrome package itself. To install those dependencies within the test Docker container, enter the following command:

```
apt install -y fonts-liberation libasound2 \
libatk-bridge2.0-0 libatk1.0-0 libatspi2.0-0 \
libcairo2 libcups2 libdrm2 libgbm1 libgtk-3-0 \
libgtk-4-1 libnspr4 libnss3 libpango-1.0-0 \
libu2f-udev libvulkan1 libxcomposite1 \
libxdamage1 libxfixes3 libxkbcommon0 \
libxrandr2 xdg-utils
```

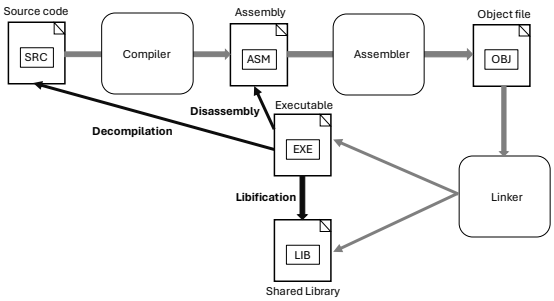

Figure 1: Libification as a reverse engineering technique.

To download and install the Google Chrome web browser within the test Docker container, enter the following commands:

```
wget https://dl.google.com/linux/direct/\
google-chrome-stable_current_amd64.deb
dpkg -i google-chrome-*.deb
```

Execution: From the terminal prompt, type:

```
cp /opt/google/chrome/chrome .
time wld -libify ./chrome
./loader ./chrome
```

Results: The libification time shall take less than 0.01 seconds. The following output shall be displayed by the last command, indicating that Google Chrome has indeed been libified and loaded via dlopen():

```
Loading of Library successful
```

A.5 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/.