Software Integrity Protection Lab
Documentation Phase 2

Alexander Küchler and Martin Morel

Technical University of Munich
Department of Computer Science
kuechler@in.tum.de, martin.morel@tum.de

1 Docker image

The public docker hub site can be found at this link: https://hub.docker.com/r/kuechlera/labinstrumenter/

You can pull the image with

```
docker pull kuechlera/labinstrumenter
```

2 Individual effort

Alexander:

–

Martin:

–

3 Tool usage documentation

4 Design decisions

In this section, we describe the design of our llvm pass as well as the stack verification.

4.1 LLVM Pass

Collect all legitimate calls by building a call graph. Store the graph in a file

4.2 Stack verification

To verify if the stack trace is legitimate, a couple of approaches are possible. In this subsection, we discuss some ideas and explain our implementation.

For the stack verification, two criteria are important. First, it should be precise and thus neither introduce false positives nor false negatives. Second, it should be efficient with regard to memory and time consumption because it is performed on runtime of the program.

A first idea makes use of hashing the stack trace and compare it against known good values. This works fine for programs without loops in the call graph. However, it cannot be applied on programs with loops in the call graph because the known good hashes would explode and require an infinite number of hashes. It is possible to reduce the number of hashes by additional analysis on the call graph and only using the first occurrence of the function in the stack trace. Still, it requires a large number of hash values for a big call graph with many paths and different loops that lead to a sensitive function.

While this allows an efficient verification, it is easy to attack this scheme. It requires complex graph analysis which is in general imprecise, so that a couple of false negatives (that is the verification might fail for valid stack traces). Also, it is easy to find false positives (that is a valid verification for invalid stack traces). E.g. if the functions f_1, f_2, f_3 are a cycle in the call graph and function f_2 has the right to access a sensitive function. In any case, the stack contains f_1, f_2, f_3 which has to lead to a valid verification. However, this is also possible if the loop has been passed and f_1 tries to access the sensitive function. With the technique explained above, it is impossible to detect this invalid stack trace.

Note that this holds for any kind where hashing is applied and thus also affects an implementation based on Merkle trees.

As a call graph can be arbitrary complex, another possibility is to check the stack against a known good call graph. In contrast to hashing, we can ensure to be precise and neither introduce any false negatives nor false positives. Therefore, this approach as a better precision than hashing. However, the efficiency of the verification depends on the size of the stack trace and the functions which are stored in the call graph. Hence, we only store the edges which are on a path that can actually reach a sensitive function. Also, it requires less computations while building the stack trace because we do not have to remove any edges.

In summary, there is a tradeoff between performance and precision. As we expect only a small difference in the overhead for the second approach but can ensure a way better precision, we decided to use the second approach.