Reentrancy Detection in the Age of LLMs

Supplementary Material

Anonymous Authors

This document serves as supplementary material to the paper *Reentrancy Detection in the Age of LLMs*, currently under review for DSN 2026, and provides an illustrative example of the application of the reentrancy detection procedure (hereafter, *Labeling Procedure*) presented in Section IV-C of the original paper. The procedure captures a broad spectrum of reentrancy scenarios, ranging from simple to more sophisticated cases, including read-only and cross-contract attacks. Its ultimate goal is to systematically encompass the behaviors targeted by different detection tools and underlying definitions of reentrancy, providing a unified framework for their evaluation.

Example. Consider a single-function scenario with a mutex-based reentrancy guard. We are going to show that Step 2 of our Labeling Procedure detects a potential reentrancy and Step 3 eventually rules it out. On the left, the code (already unfolded by Step 1) exhibits two side effects after the external call, so the contract is potentially vulnerable according to the first two steps. On the right, a simple attack scheme: the attacker deposits 100, then calls `withdraw()` and initiates the mutual recursion, reentering the `withdraw()` again from the receive/fallback function.

```
contract Victim {      // already unfolded by Step 1
    bool private flag = false;
    mapping (address => uint256) public balances;

    function withdraw() public {
        require(!flag, "Locked");
        flag = true;
        uint amt = balances[msg.sender];
        require(amt > 0, "Insufficient funds");
        (bool success, ) = msg.sender.call{value:amt}("");
        require(success, "Call failed");
        balances[msg.sender] = 0;  // side effect
        flag = false;    // another side effect
    }

    function deposit() public payable {
        require(!flag, "Locked");
        flag = true;
        balances[msg.sender] += msg.value;
        flag = false;
    }
}
```

```
contract Attacker {
    Victim private c;

    constructor(address a} {
        c = Victim(a);
    }

    function attack() public {
        c.deposit{value:100}();
        c.withdraw();
    }

    receive() external payable {
        c.withdraw();
    }
}
```

State transitions can be written as follows, where function calls are annotated above arrows:

$$\sigma_0 \xrightarrow{\texttt{withdraw}} \sigma_1 \xrightarrow{\texttt{withdraw,receive}} \sigma_2 \xrightarrow{\texttt{withdraw,receive}} \sigma_3 \longrightarrow \dots$$

The first transition exhibits only one call to `withdraw` as it represents the initial call performed by the attacker in the `attack()` function; the rest are all bounces between `withdraw` and the receive/fallback function.

Assuming a few calls to `deposit()` have been previously performed by other users before the attack takes place, the execution yields to the following states:

$$\begin{aligned}
\sigma_0 = \quad & \langle \texttt{balances}_0, \texttt{flag}_0, \beta_0 \rangle \quad \texttt{balances}_0 = \langle \texttt{user}_1 \mapsto 200; \texttt{user}_2 \mapsto 500; \texttt{attacker} \mapsto 100 \rangle \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{flag}_0 = \texttt{false} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \beta_0 = 800 \\
\sigma_1 = \quad & \langle \texttt{balances}_1, \texttt{flag}_1, \beta_1 \rangle \quad \texttt{balances}_1 = \langle \texttt{user}_1 \mapsto 200; \texttt{user}_2 \mapsto 500; \texttt{attacker} \mapsto 0 \rangle \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{flag}_1 = \texttt{true} \\
& \qquad\qquad\qquad\qquad\qquad\qquad\quad \beta_0 = 700 \\
\sigma_2 = \quad & \bot
\end{aligned}$$

Thanks to the mutex guard, $\sigma_2 = \bot$, as the `require` at the beginning of the `withdraw()` function fails due to the `flag = true`. In other words, the execution incurs in an error state at the first reentrant call performed by the attacker, rendering any state beyond σ_2 unreachable.

Now let us consider the same chain of calls as if they were flattened, i.e. as if every call to `withdraw()` was performed in a separate transaction. This is operationally equivalent to a flat sequence of `withdraw()` invocations and an empty receive/fallback function, hence the following state transitions:

$$\sigma_0 \xrightarrow{\texttt{withdraw}} \sigma'_1 \xrightarrow{\texttt{withdraw}} \sigma'_2 \xrightarrow{\texttt{withdraw}} \sigma'_3 \longrightarrow \dots$$

The call flow is basically equivalent to its reentrant counterpart except for the missing receive/fallback function. With a major difference, though: it is not mutually recursive anymore, leading to the following states:

$$\begin{aligned}
\sigma_0 = \quad & \langle \texttt{balances}_0, \texttt{flag}_0, \beta_0 \rangle \quad && \texttt{balances}_0 = \langle \texttt{user}_1 \mapsto 200; \texttt{user}_2 \mapsto 500; \texttt{attacker} \mapsto 100 \rangle \\
& && \texttt{flag}_0 = \texttt{false} \\
& && \beta_0 = 800 \\
\sigma'_1 = \quad & \langle \texttt{balances}_1, \texttt{flag}_1, \beta_1 \rangle \quad && \texttt{balances}_1 = \langle \texttt{user}_1 \mapsto 200; \texttt{user}_2 \mapsto 500; \texttt{attacker} \mapsto 0 \rangle \\
& && \texttt{flag}_1 = \texttt{false} \\
& && \beta_0 = 700 \\
\sigma_2 = \quad & \bot
\end{aligned}$$

Notably, $\sigma_1 \neq \sigma'_1$, but our definition of Step 3 states that the divergence must appear in the final states σ_n and σ'_n. In this case, $n = 2$ because any state beyond σ_2 is unreachable, and $\sigma_2 = \sigma'_2 = \bot$, thus no divergence appear. This means that the contract is safe, which it actually is thanks to the mutex.