[Engineer]
Luis Lucena
[Audited]
https://github.com/luislucena16/lightcurve/blob/main/poc/VulnerableContract.sol (VulnerableContract)

[Summary]

1. **High: 1**
2. **Medium : 0**
3. **Low : 2**
4. **Informational: 1**
5. **Optimizations**

[Issues & POCs]

1.1 : High (reentrancy-eth)
Reentrancy in VulnerableContract.withdraw(uint256)
(poc/VulnerableContract.sol#21-27):

External calls:
- (success) = msg.sender.call{value: amount}()
(poc/VulnerableContract.sol#24)

State variables written after the call(s):
- balances[msg.sender] -= amount (poc/VulnerableContract.sol#26)

VulnerableContract.balances (poc/VulnerableContract.sol#5) can be used in cross function reentrancies:
- VulnerableContract.balances (poc/VulnerableContract.sol#5)
- VulnerableContract.deposit() (poc/VulnerableContract.sol#17-19)
- VulnerableContract.withdraw(uint256)
(poc/VulnerableContract.sol#21-27)

Issue Description:

In the VulnerableContract smart contract, a reentrant vulnerability has been identified in the withdraw(uint256) function that could allow an attacker to manipulate the state of the contract and make recursive calls to extract more funds than expected. The vulnerability is related to user balance manipulation and the call to msg.sender.call{value: amount}().

Proof of Concept:

- Alice deposits funds into her Wallet contract.
- Bob calls the startReentrancyAttack function of his MaliciousContract, sending an amount of ether.
- MaliciousContract makes a deposit into Alice's Wallet contract using Alice's deposit function.

- After depositing, MaliciousContract repeatedly calls the withdraw function of the Wallet contract. Each time it makes a withdrawal, the withdraw function of the Wallet contract is invoked.
- On each call to withdraw, the Wallet contract attempts to transfer funds to Bob's address. However, before completing the transfer, control reverts to the withdraw function in the MaliciousContract.
- In the withdraw function of the MaliciousContract, Bob can take additional actions before the Wallet contract completes the transfer. This includes making further deposits, which initiates a new cycle of withdraw calls.
- This cycle repeats, allowing Bob to perform additional actions on each iteration before the transfer completes. The re-entry attack continues until the funds in the Wallet contract are exhausted.

Recommended Mitigation Steps:

- Use the nonReentrant modifier: This modifier is used to prevent recursive calls to a function in the same contract.
- Do not allow contract addresses.

- Added nonReentrant modifier, to avoid re-entry attacks when withdrawing.

- Added two extra functions getBalance and getOwner. They act as an interface to get the balance, allowing to change the internal implementation in the future without affecting external contracts that use the function.

- Changed the balances and owner variable from public to internal, making the variable internal encapsulates the internal implementation and prevents other external contracts from accessing it directly.

- In terms of security, by limiting direct access, you can reduce the risk of malicious manipulation of the variable by external contracts.

Reference:

https://docs.openzeppelin.com/contracts/5.x/api/utils#ReentrancyGuard

https://docs.soliditylang.org/en/v0.8.20/smtchecker.html#external-calls-and-reentrancy

3.1 : Low (events-access)

VulnerableContract.transferOwnership(address) (poc/VulnerableContract.sol#29-31) should emit an event for:
- owner = newOwner (poc/VulnerableContract.sol#30)

Issue Description:

In the VulnerableContract smart contract, a missing event emission in the transferOwnership(address) function has been identified, specifically after changing the new owner (newOwner). The lack of event issuance can make it difficult to track critical changes in contract ownership and affect transparency in contract management.

Proof of Concept:

- Alice currently owns the VulnerableContract.
- Bob wishes to become the new owner and executes the transferOwnership function with his address as the argument.
- Bob becomes the new owner, but there is no public record of this change.
- Alice and other observers do not receive notifications about the change in contract ownership.
- The lack of visibility on the transfer of ownership can cause confusion and lack of transparency.

Recommended Mitigation Steps:

- Make sure to issue important events, such as changes in ownership (OwnershipTransferred in this case). Events are crucial for transparency and allow users and other contracts to observe and react to significant changes.

- Where you use if - revert or require, provide clear messages indicating why the condition failed. This helps in debugging and in quickly understanding the reason for a failure.

Reference:

https://docs.soliditylang.org/en/v0.8.20/structure-of-a-contract.html#events

3.2 : Low (missing-zero-check)

VulnerableContract.transferOwnership(address).newOwner (poc/VulnerableContract.sol#29) lacks a zero-check on :
- owner = newOwner (poc/VulnerableContract.sol#30)

Issue Description:

The transferOwnership function in the VulnerableContract in poc/VulnerableContract.sol, specifically in lines 29-30, does not perform a zero-check on the address of the new owner (newOwner).

Proof of Concept:

If Alice attempts to transfer ownership to a zero address (0x0), the transferOwnership function will still run without checking this condition. This may result in an undesired scenario where ownership of the contract is transferred to a null address, thus losing control of the contract.

Recommended Mitigation Steps:

- It is recommended to add a check before the transfer of ownership to ensure that the new owner's address is not the zero address. Such a check helps prevent accidental or malicious transfers of ownership to invalid addresses.

- We also added an extra security function called isContract written in assembly to validate that the address entered is not a contract address.

Reference:

Address-zero(0) represents the zero address or the null address. It's an address value that denotes the absence of a valid Ethereum address and is used as the initial/default value of addresses in solidity. It looks like this: 0x0000000000000000000000000000000000000000.

4.1 : Informational (low-level-calls)

Low level call in VulnerableContract.withdraw(uint256) (poc/VulnerableContract.sol#21-27):
- (success) = msg.sender.call{value: amount}() (poc/VulnerableContract.sol#24)

Issue Description:

The withdraw function of the VulnerableContract contract, located in poc/VulnerableContract.sol on lines 21-27, uses a low-level call with msg.sender.call{value: amount}() to transfer funds.

The use of low-level calls is error-prone. Low-level calls do not check for [code existence](https://docs.soliditylang.org/en/v0.8.20/control-structures.html#error-handling-assert-require-revert-and-exceptions) or call success.

Proof of Concept:

- Alice deposits 5 Ether into the contract using the deposit function.
- Bob, the attacker, repeatedly executes the withdraw function in an attempt to perform re-entrancy and maliciously obtain funds.

- Because the withdraw function uses a low-level call (msg.sender.call{value: amount}()) to transfer funds. Bob exploits this to re-enter and execute malicious code on each call.

Recommended Mitigation Steps:

- It is recommended to replace low level calling with secure transfer methods such as transfer or send.

- In this case the decision was made to use transfer as it brings with it reentrancy protection.

- Avoid low-level calls. Check the call success. If the call is meant for a contract, check for code existence.

Reference:

The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

https://docs.soliditylang.org/en/v0.8.20/control-structures.html#error-handling-assert-require-revert-and-exceptions

5.1 Optimizations (Storage, Events, Handle Errors)

- The focus of the optimization is to offer a more user-friendly code for development, auditing, debugging and testing through a modular architecture, both alone and as a team, making the order of the code more understandable and therefore more verbose.

- This type of architecture offers the ease of working as a team without stepping on each other's toes, as the great flexibility of having distributed contracts in specific places provides that layer.

- This also adds better error handling, where you can understand more clearly why a process might fail and provides better semantics of the code.

Some examples are:

Folder: **contracts**
- Sub-folders: **test, test-1, test-2**
- **test:**
  - Test.sol
  - TestEvents.sol
  - TestStorage.sol
  - TestErrors.sol

Test.sol => base contract
TestEvents.sol => all events related to base contract
TestStorage.sol => all internal constants, mappings and structures of base contract
TestErrors.sol => abstract contract with all error messages using if - revert and related to base contract.

- Another benefit is that it prevents extremely large contracts that may exceed the excess space in the contract or generate a giant bytecode.

- Each contract function will have its NatSpec to facilitate the understanding of the functions in a quick way, it is very important for teamwork or in relation to audits.
https://docs.soliditylang.org/en/v0.8.20/natspec-format.html