A  APPENDICES

A total of 9 strongly correlated cases with ERC have been identified (issues arising from auditing ERC protocol code/ERC application) as well as 5 weakly correlated issues (problems arising from insecure coding practices by developers). For each case, the following information is provided: case code, repair suggestions, and repair code.

**A.1 Strongly Related Issues**

*A.1.1 Integer Overflow & Unsafe Compiler Version*

Taking the example from Consensys' audit report on ERC137, specifically the issue "SimplePriceOracle.price is susceptible to integer overflow" published on May 29, 2019.



Figure A-1

Issue: The vulnerability arises from the `price` function in the `SimplePriceOracle.sol` contract. Due to the lack of overflow protection in the calculations, the multiplication operation may result in a value that exceeds the maximum limit of the `uint` type in Solidity, which can lead to an overflow. Solidity restricts `uint` values to the range of [0, 2^256-1], and when values go beyond this range, integer overflow and underflow issues can occur. The `price` function does not perform a check on the calculations involving `uint` values, which results in the possibility of integer overflow.

Recommendation: To address this type of issue, it is recommended to add checks on the values involved in the calculations. Additionally, starting from Solidity version 0.8.0, the programming language itself includes checks for integer overflow, and if an overflow occurs, a `revert` will be triggered. It is advisable to consider the programming language version in the Ethereum Improvement Proposal (EIP) to account for the handling of integer overflows. The code below shows the repaired version.

Below is the repaired version of the code:



Figure A-2

A.1.2 - Logic Issue & Incorrect Transfer Logic

This case is taken from the Salus audit report on ERC3525, specifically the issue "ERC-3525 receiver contract may not be able to receive ERC-3525 values due to an issue in the \_checkOnERC3525Received() function."



Figure A-3

Issue: The issue arises in the `\_checkOnERC3525Received()` function of the ERC-3525 receiver contract, as highlighted in the project's code. ERC-3525 is a token that is compatible with ERC-721. During the transfer process, if the destination address is a contract, the `onERC721Received()` function of the target contract is called. However, the highlighted portion of the code includes an additional check that verifies if the target contract implements the `IERC165` interface. This check can result in a transaction error and cause the token to be locked if the target contract does not implement the `IERC165` interface.

Recommendation: To address this issue, it is recommended to remove the highlighted check in the code. This check is unnecessary as not all contracts will implement the `IERC165` interface. By removing this check, the problem can be resolved.

Below is the repaired version of the code:



Figure A-4

A.1.3 - Lack of validation & Signature/Replay attack

Issue:The issue arises when the signing functions, such as ERC-1271, are improperly used, which can lead to signature/replay vulnerabilities. In the given example, the code does not check whether the signature has been used before, allowing for unlimited usage of the same signer for an airdrop.

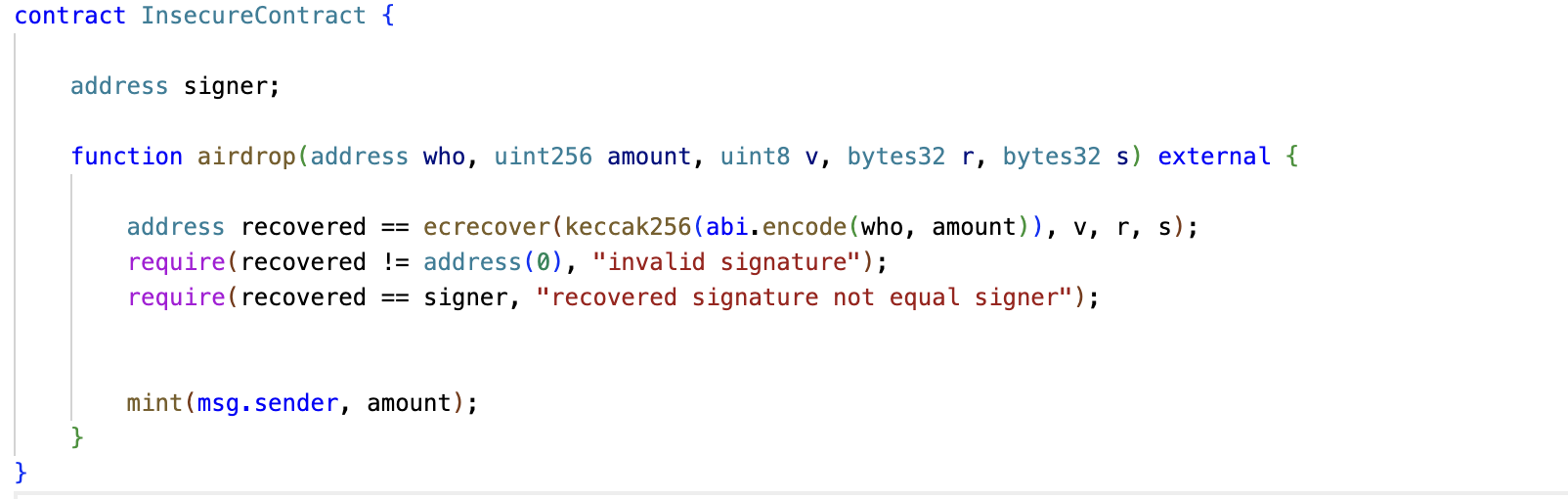


Figure A-5

Recommendation: To address this type of issue, the following repair suggestions are recommended:

(1) Use well-known and secure cryptographic functions and methods when dealing with cryptographic-related functions.

(2) Implement a check to verify whether the signature has been used before, to prevent replay attacks.

Below is the repaired version of the code:

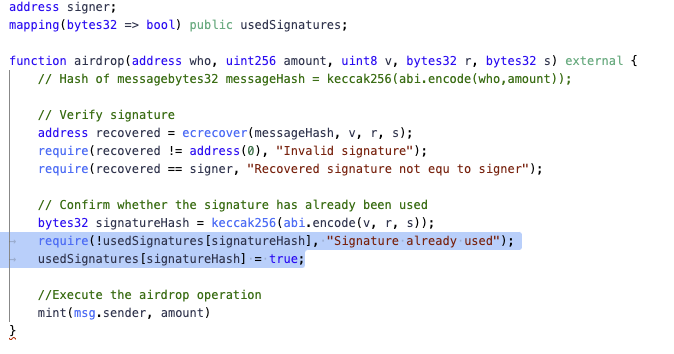


Figure A-6

A.1.4 - Access control & Lack of permission control

This case is taken from the insecure usage of the ERC20 interface and functions that led to the LEVUSDC hack incident.

IMG_257

Figure A-7

Issue: The issue arises from the insecure usage of the ERC20 interface and functions, leading to the LEVUSDC hack incident. The `approveToken` function in the code is using `safeApprove` in an insecure manner, allowing anyone to call it and transfer the tokens of the `ProxyDEPUSDT` contract. This vulnerability has resulted in financial losses.

Recommendation: To address this case, the following repair suggestions are recommended:

(1) Discontinue the use of the `approveToken` function and instead use the `approve` function directly, called by `msg.sender`.

(2) Add restrictions or permissions to the `approveToken` function, such as allowing only privileged addresses to call it.

A.1.5 - Memory Overflow

Issue: The vulnerability arises from the `init` function in the code. The memory is only reserved for `capacity` bytes, but the actual calculation for the length of the prefix array of `bytes` requires `capacity + 32` bytes. If the surrounding functions are implemented correctly, this will corrupt the adjacent memory, leading to incorrect assignment of subsequent variables.

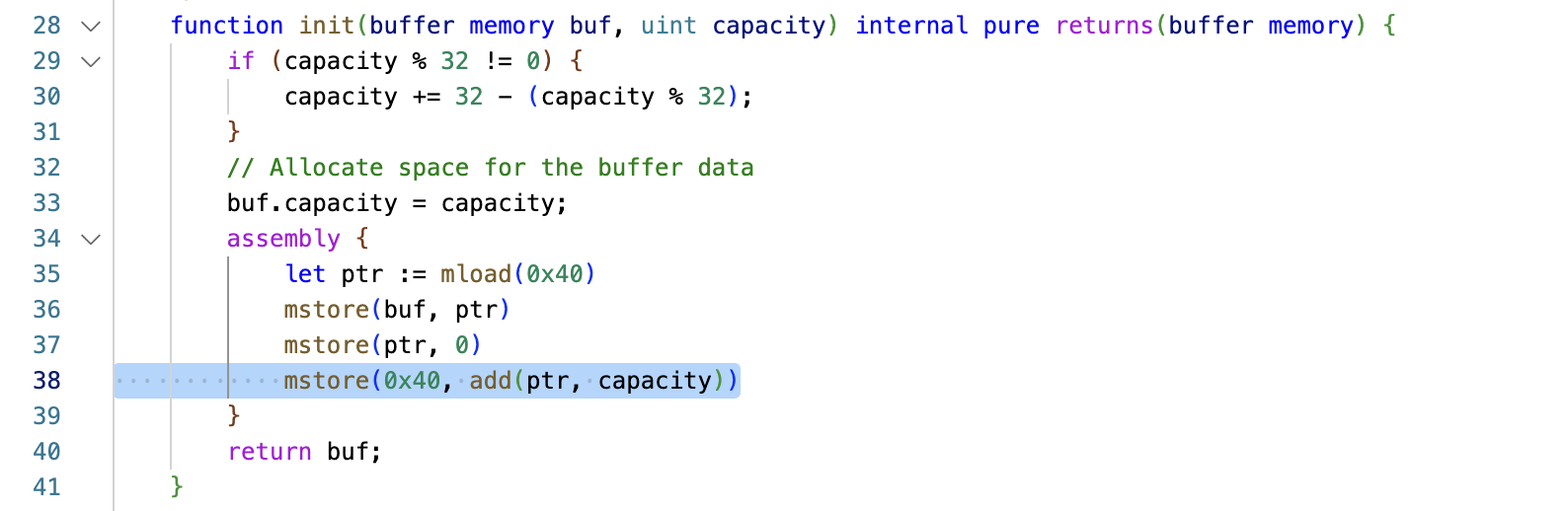


Figure A-8

Recommendation: To address this type of issue, the following repair suggestions are recommended:

(1) Avoid using statements like assembly for memory allocation.

(2) Estimate the memory requirements in advance and perform thorough testing, including overflow testing for buffer memory, with appropriate test components.

(3) Conduct a security audit to identify and address potential vulnerabilities.

In this case, the repair suggestion is to increase the bytes allocation for the `uint256` variable by adding `0x32` bytes.

Below is the repaired version of the code:

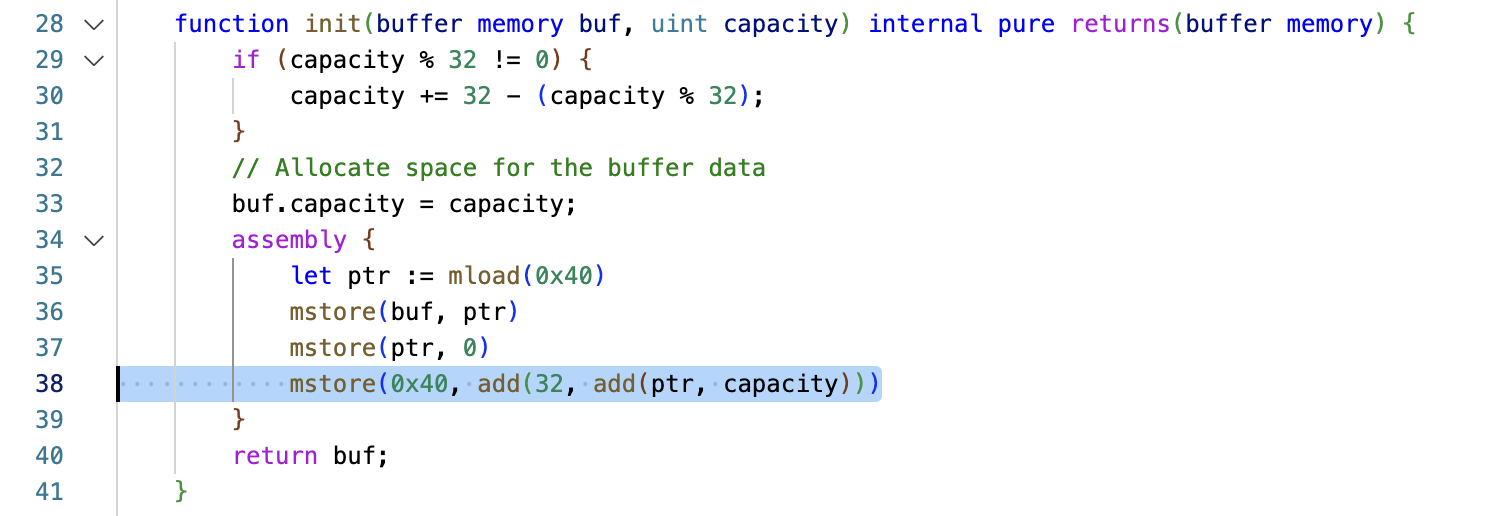


Figure A-9

A.1.6 - Front-Running

Issue: The vulnerability arises from the `ETHRegistrarController.register` function in the code. The vulnerability occurs due to the lack of verification for the caller of `commit` and `register` functions, which can result in a situation where the caller of `commit` is different from the caller of `register`, leading to front-running attacks.

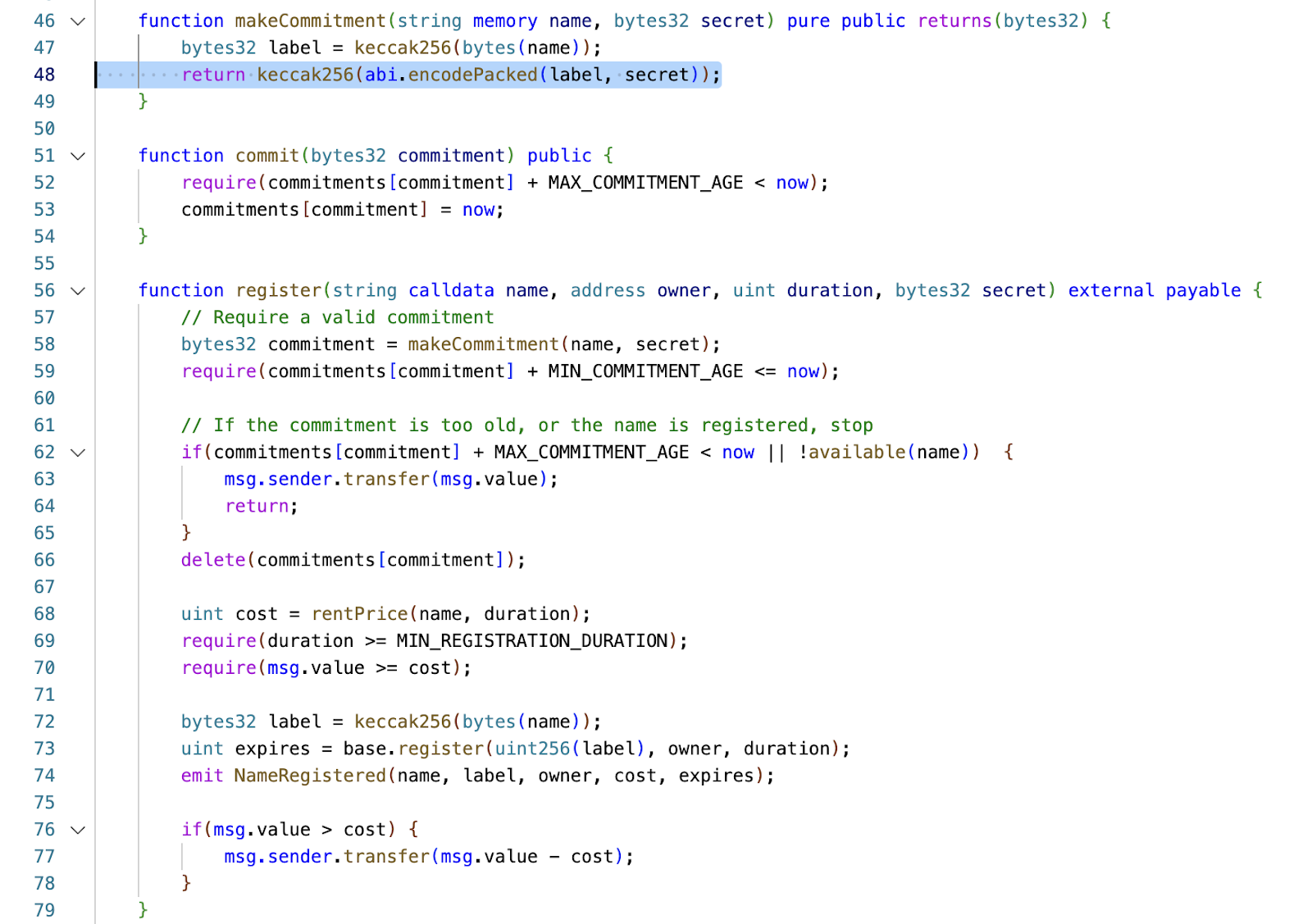


Figure A-10

Recommendation: To address this type of issue, the following repair suggestions are recommended:

- Add explicit verification and validation for each function call, ensuring that the logic of the contract corresponds correctly to the sequence of function calls.

- Implement access controls to ensure that only users with the appropriate permissions can call relevant functions.

In this case, the repair suggestion is to add verification for `msg.sender`.

Below is the repaired version of the code:

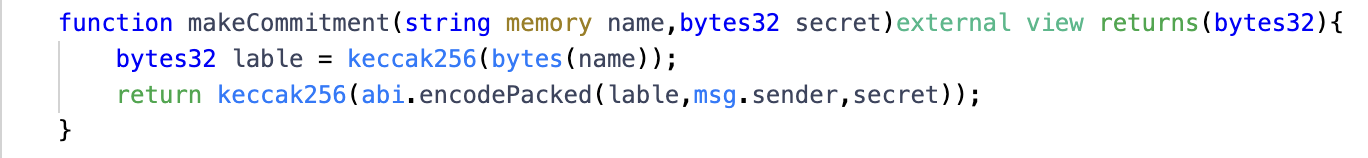


Figure A-11

A.1.7 - Reentrancy

Select the 2019 Uniswap reentrancy attack caused by the unsafe use of ERC777 due to hooks.



Figure A-12

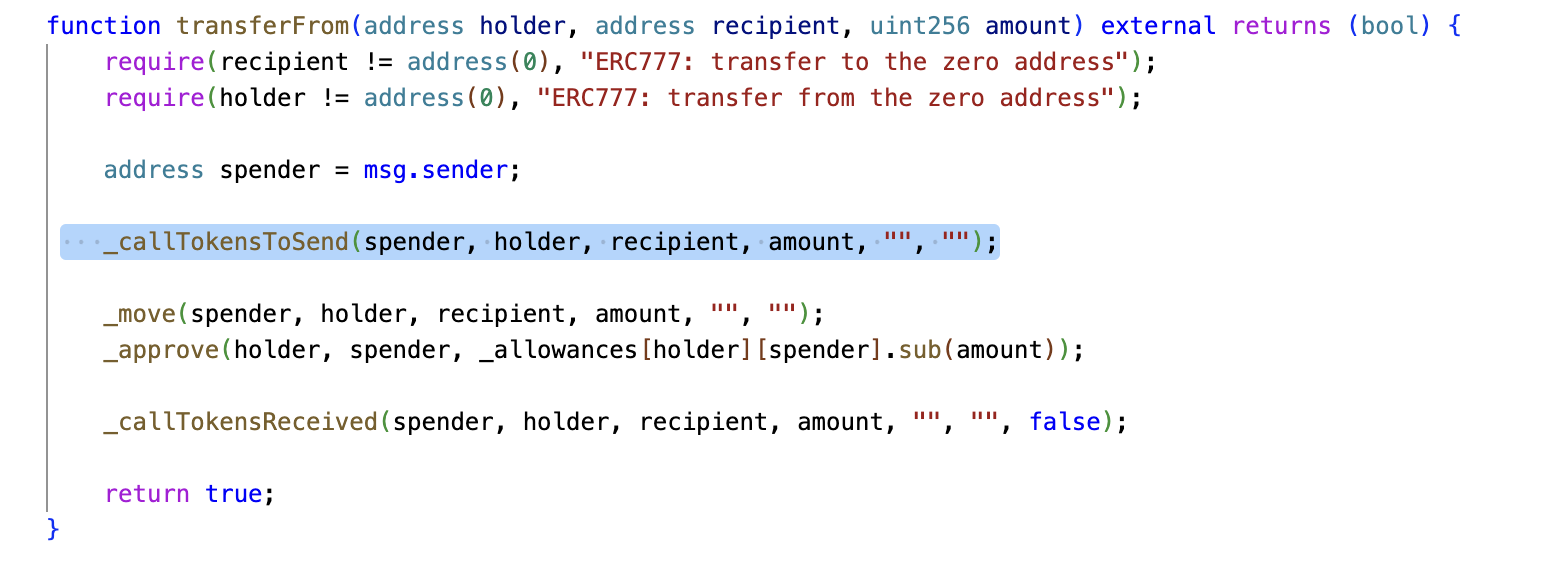


Figure A-13

Issue: The vulnerability arises from the unsafe usage of the ERC777 token contract, which allows reentrancy attacks due to the execution of the hook function before the transfer. This vulnerability has led to a reentrancy attack on the `tokenToEthInput` function in Uniswap.

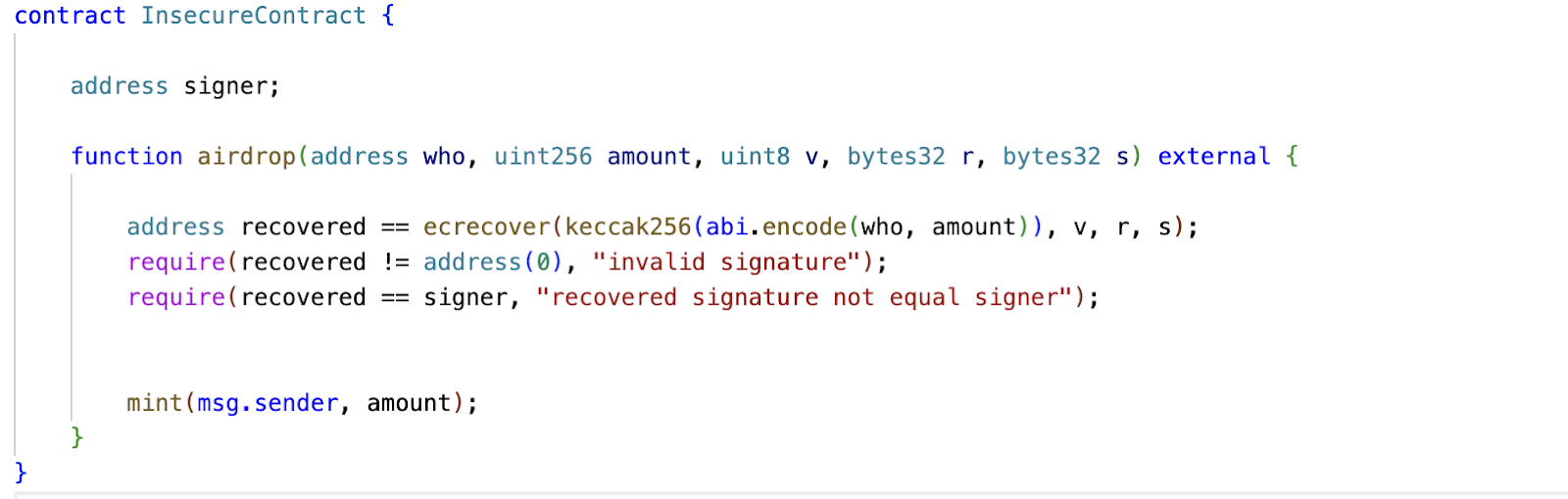


Figure A-14

Recommendation: To address this type of issue, the following repair suggestions are recommended:

1. Use safe functions, such as `transfer`, that do not allow reentrancy.

2. Implement the "checks-effects-interactions" pattern, where all necessary checks and state changes are performed before interacting with external contracts.

3. Introduce a mutex lock to prevent reentrancy attacks.

For this specific case, implementing a mutex lock can be an effective solution.

A.1.8 - Unchecked return value

Issue: The code provided has a problem due to the lack of checking the return value of a function call in the `withdraw` function. The code makes a function call to transfer funds, but does not check the return value. If the transfer fails, the user will not receive the ETH, but their balance will still be deducted, resulting in financial losses for the user.

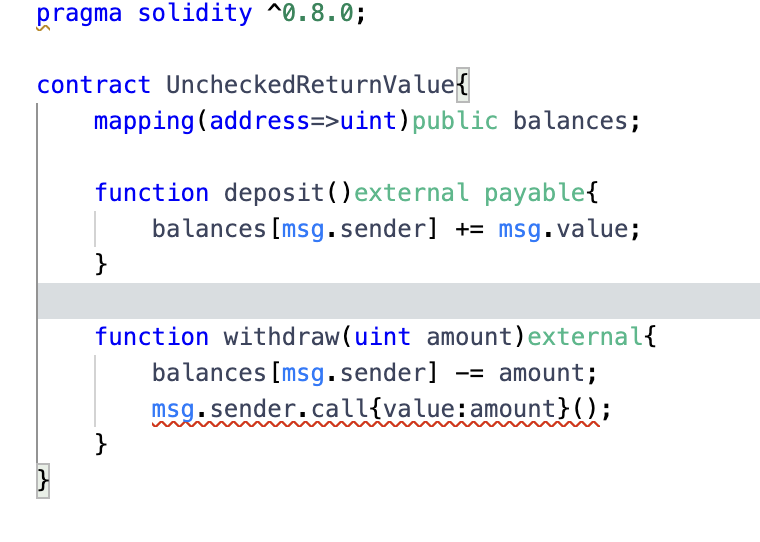


Figure A-15

Recommendation: To address this type of issue, it is recommended to check the return value of the function call after the transfer and handle any exceptions appropriately to prevent user's financial losses.

Below is the repaired version of the code:

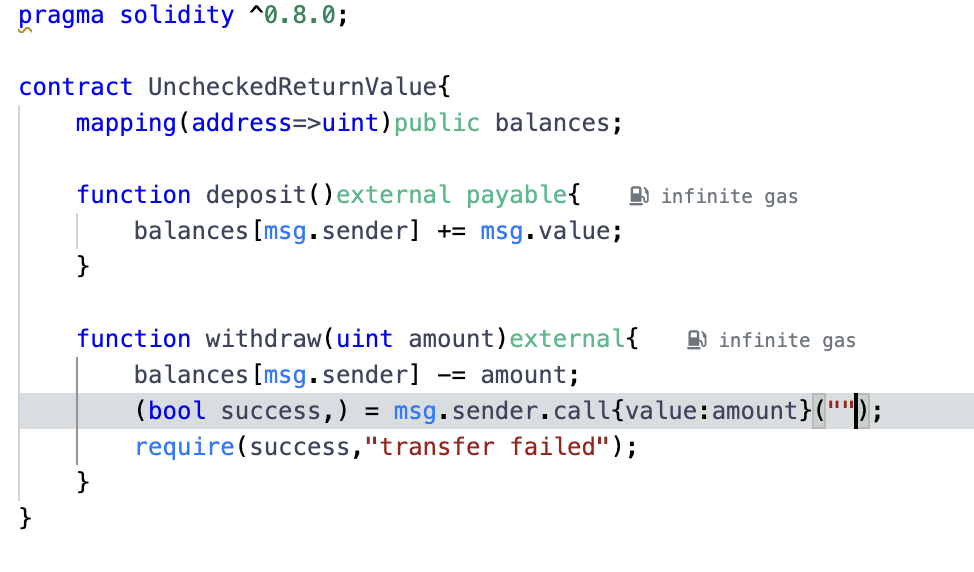


Figure A-16

**A.2 Weakly Related Issues**

A.2.1 Usage of delegatecall

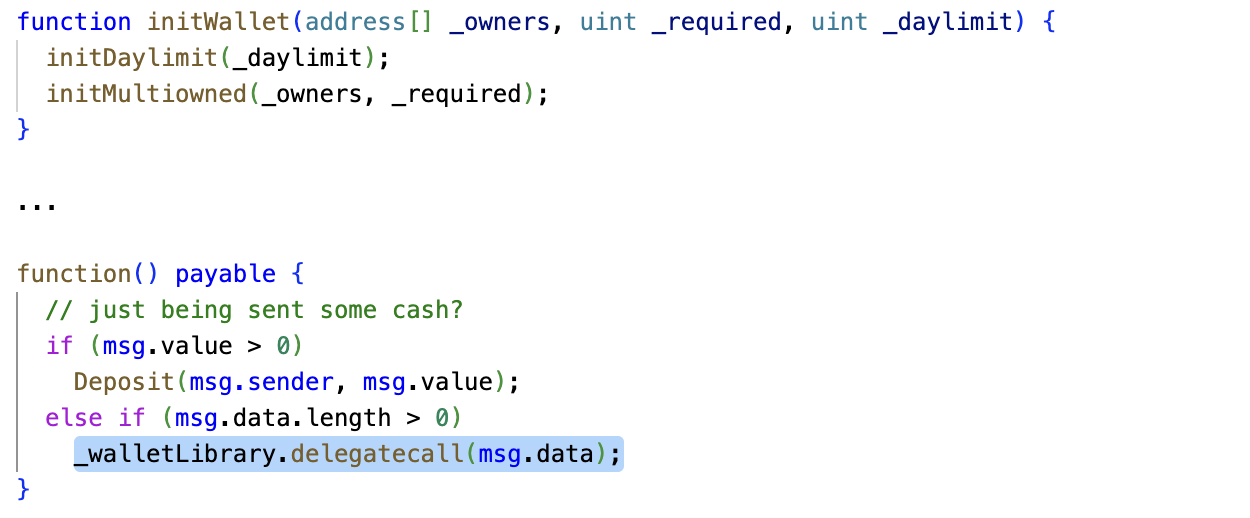


Figure A-17

Issue: The vulnerability arises from the usage of the `delegatecall()` function in the Parity multi-signature wallet contract. All public functions, including the `initWallet` function, are visible to everyone, and no protective measures are implemented in the `initWallet` function. This allows an attacker to use `delegatecall` to set themselves as the owner and call any public function.

Recommendations:To address this issue, the following repair suggestions are recommended:

1. Exercise caution when using the `delegatecall()` function. Understand the potential risks and ensure proper security measures are in place.

2. Clearly define the visibility of functions. By default, functions are public, so consider using the `external` keyword to prevent internal calls from external access.

3. Strengthen access control. Implement access control mechanisms, such as using the `onlyOwner` modifier, for sensitive functions.

Below is the repaired version of the code:

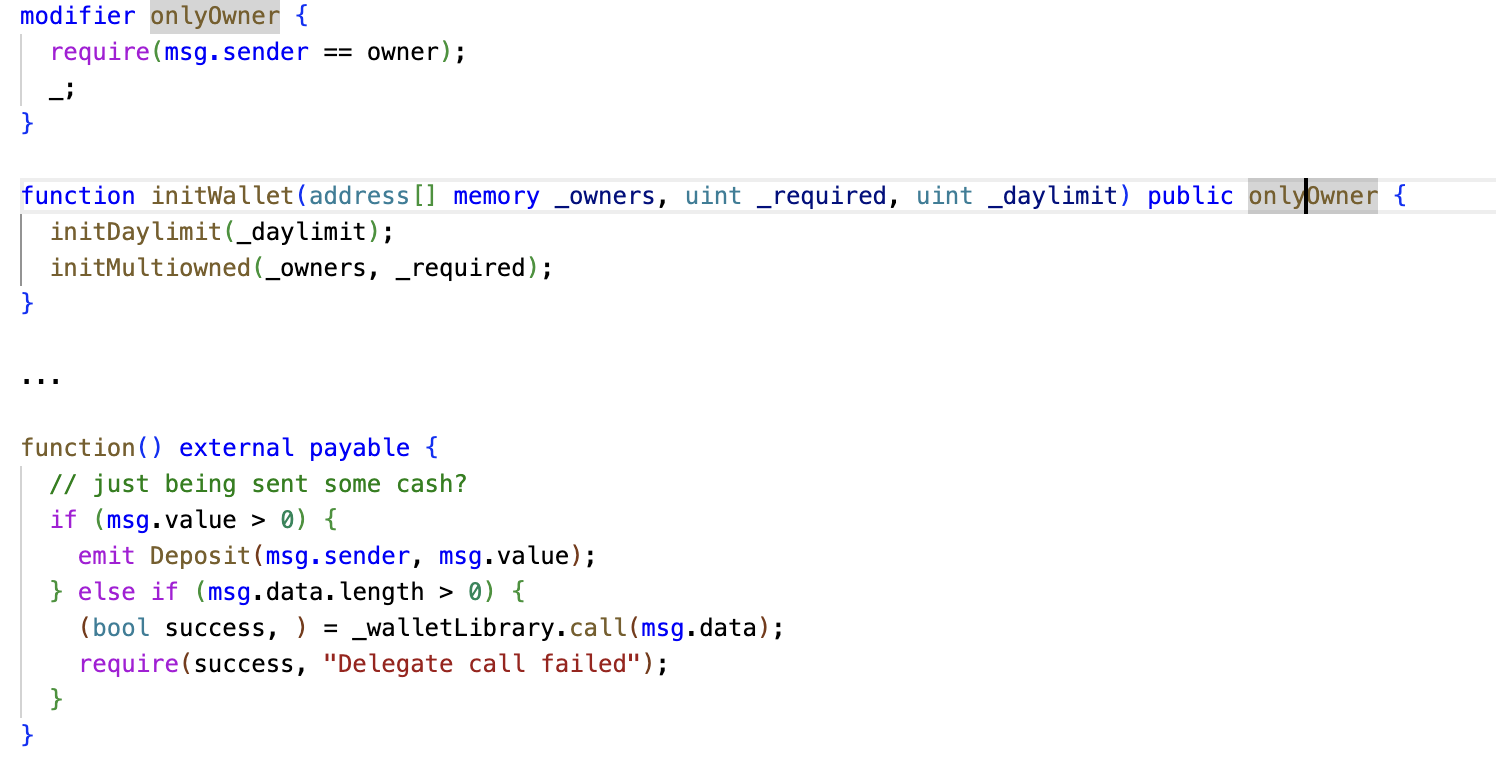


Figure A-18

A.2.2 DoS Caused by External Calls

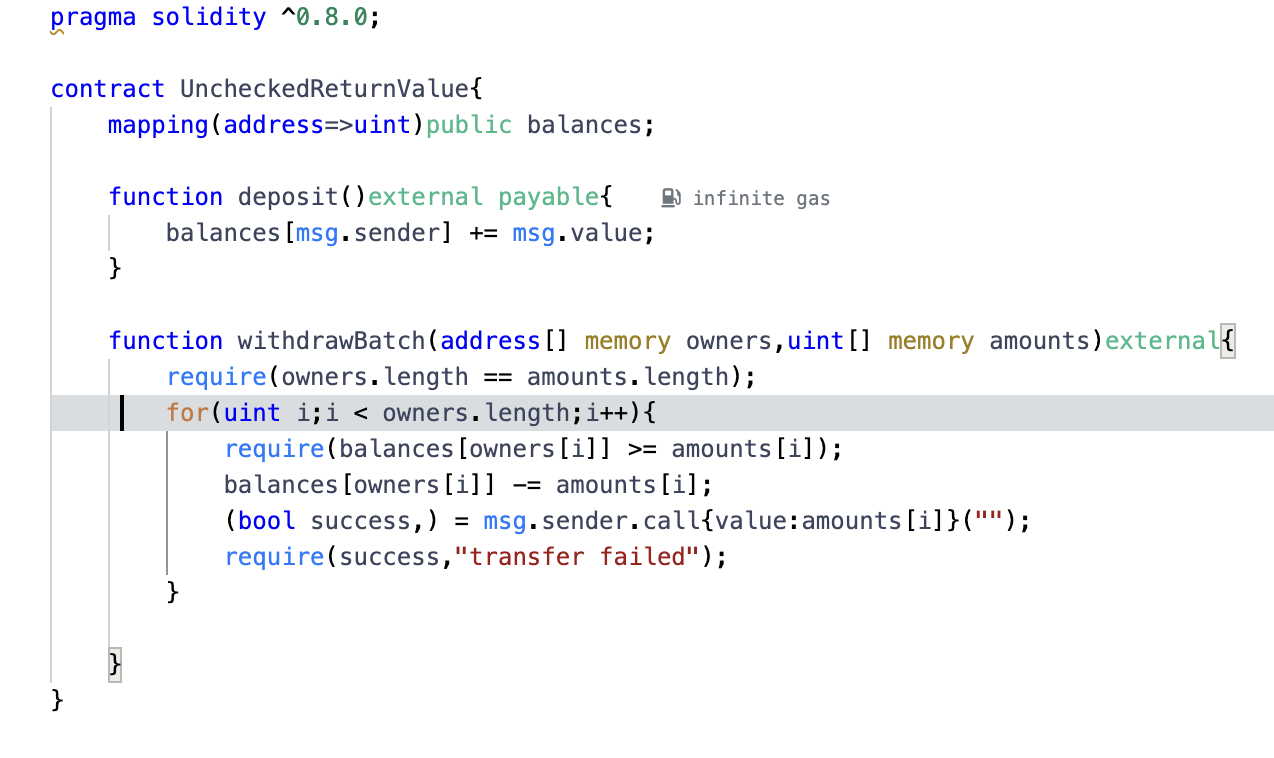


Figure A-19

Issue: The issue arises in the `withdraw` function. The contract first checks if the caller's account has sufficient balance and then transfers the specified amount of Ether to the caller. However, before transferring the Ether, the contract uses an external call `msg.sender.call{value: amount}("")` to send Ether to the caller. This external call is a low-level call and does not pass any remaining gas to the caller. As a result, if the caller is a malicious contract that refuses to receive Ether, it can cause the contract's execution to terminate.

In this scenario, if a malicious contract continuously calls the `withdraw` function, providing a very small balance each time, it can consume all the gas and make the contract unable to continue execution or directly refuse to accept ETH, resulting in a denial-of-service attack.

Recommendations: To address this issue, it is recommended to use the `transfer` or `send` functions instead of the low-level `call` function when transferring Ether to the caller. The `transfer` and `send` functions automatically handle the gas stipend and revert if the transfer fails.

Below is the repaired version of the code:

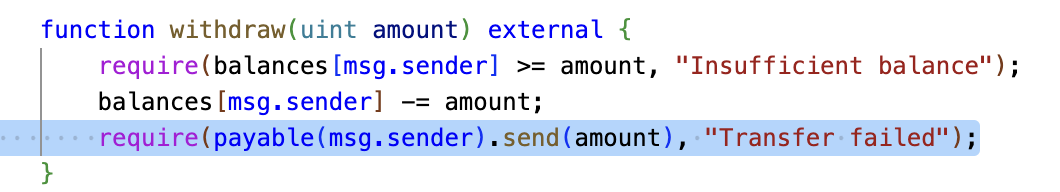


Figure 23

A.2.3 Race Condition

### IMG_272

Figure A-20

Issue: The issue arises in the `raceCondition` function when multiple accounts simultaneously call it. They can concurrently access and modify the `counter` variable, resulting in a race condition. In this scenario, the final value of the counter may differ from the expected value.

Recommendations: To address this issue and prevent the race condition, it is recommended to implement a mechanism that ensures only one account can access the critical section at a time. This can be achieved by using a mutex lock or a synchronization mechanism. Here, we will suggest using a mutex lock.

Below is the repaired version of the code:

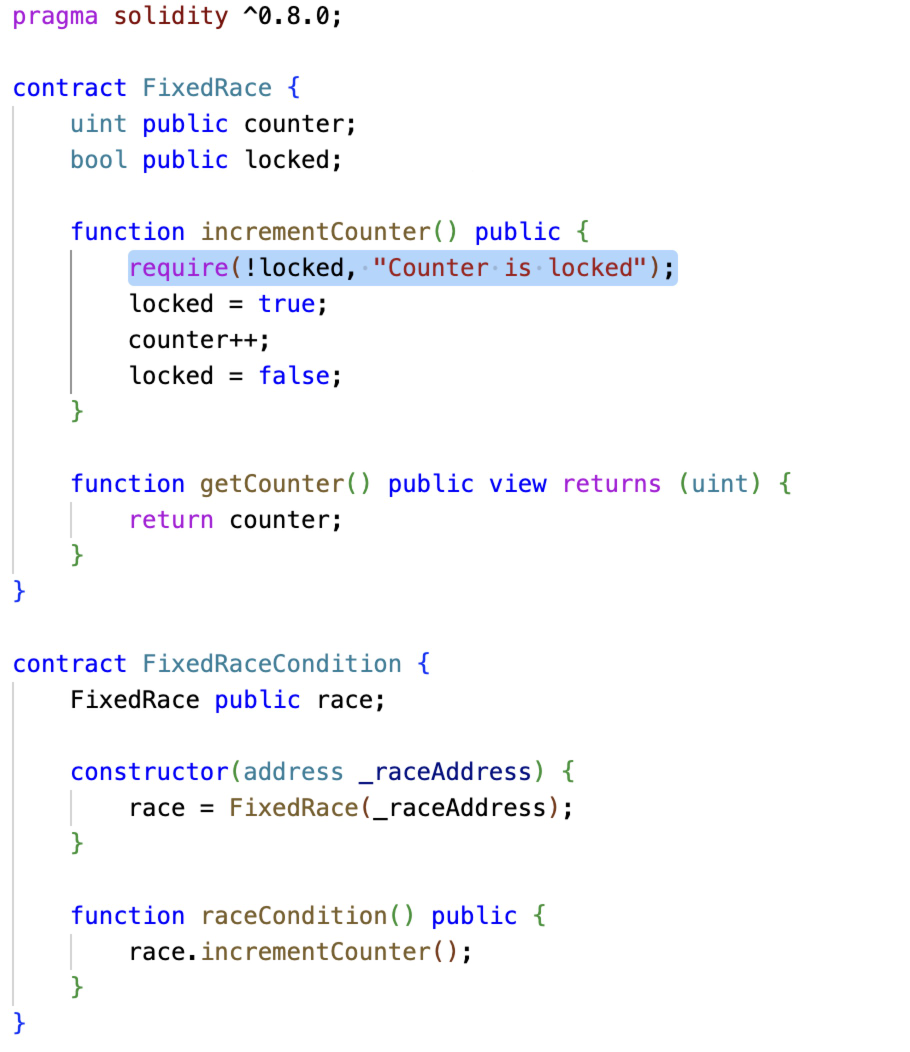


Figure A-21

A.2.4 DoS Due to Array Length

### IMG_274

Figure A-22

Issue: The issue arises in the `addToDataArray` function when a very large array length is passed. This can result in a significant amount of time and resources being consumed during the execution of the `dataArray.push` operation. As a result, a malicious user can exploit this by passing an excessively large array length to consume the contract's execution time and gas.

Recommendation: To address this issue and prevent excessive resource consumption, it is recommended to implement a maximum length limit for variables like arrays, where the length is unknown. This limit can help control the amount of resources consumed during operations on these variables.

Below is the repaired version of the code:

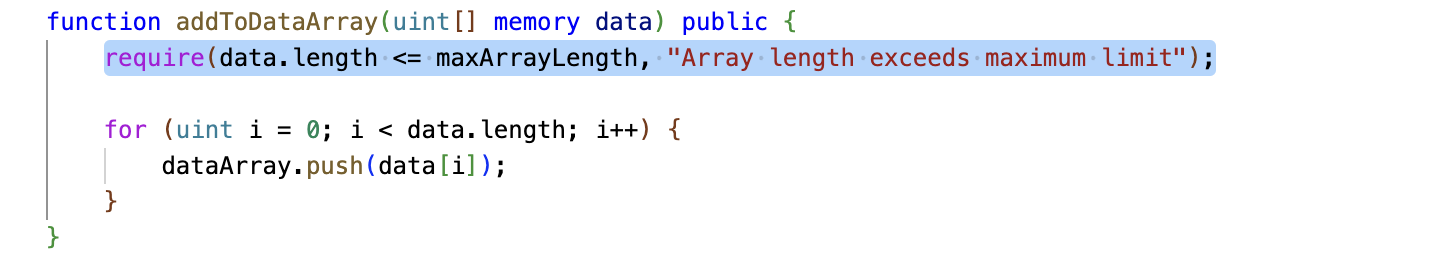


Figure A-23

The fixed code is shown in the diagram above.

A.2.5 - Ethereum Features

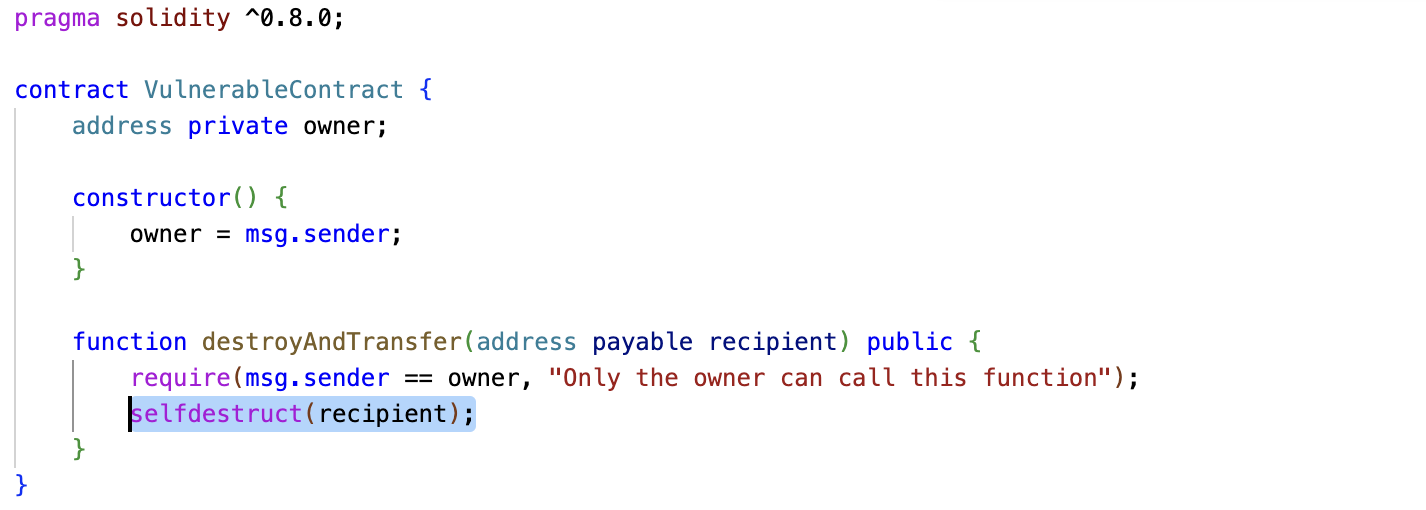


Figure A-24

Issue: For example, the `selfdestruct` function can forcibly transfer the contract's token balance to another address.

Recommendation: To address this issue, the following suggestions are recommended:

1. Avoid using ETH for logic review in scenarios where such issues may arise.

2. EIPs should consider the current characteristics of ETH.