Solidity SimpleDAO Smart Contract Security Audit

Contract name: SimpleDAO.sol

Solidity version: 0.8.13

Auditor: Alejandro Durá

Date: January 2026

1. Summary.

This report presents the results of a security audit of the smart contract SimpleDAO.sol

In the repository, you can find both SimpleDAO.sol and SimpleDAO_Original.sol These contracts are the same, but SimpleDAO.sol is the patched and refactored one and SimpleDAO_Original.sol is the non audited contract. The original contract name is SimpleDAO.sol, but we updated to the repository both versions so you can see the changes and corrections.

We reference **SimpleDAO_Original.sol as the V1 version**, and **SimpleDAO.sol as the V2 version**.

These corrections are related to security and gas optimizations. You probably will see some differences in the audited code that are not related with security but with gas consumptions. In this file you will find the security audit documentation. But there is another one explaining gas optimizations in the github repository. All security and gas optimization improvements are merged in the SimpleDAO.sol smart contract, which is the final V2 version.

This audit is focused on finding logical flaws, security vulnerabilities and deviations from Solidity best practices.

All security tests can be found in the **test/security/VersionsSecurityTests.t.sol** file in the github repository.

A total of 5 issues were identified:

High: 1

Medium: 2

Low: 1

Informational: 1

2. Scope and methodology.

Scope

- **Contract audited:** SimpleDAO_Original.sol.
- **Solidity version:** 0.8.13
- **Excluded:** External dependencies or contract addresses like the token.

Methodology

- We performed a manual code review and also applied some tests to check if the studied vulnerability is solved. See the **VersionsSecurityTests.t.sol** file to see all security tests performed to detect these vulnerabilities. Not all of them are shown or explained here.
- Analysis of common attack vectors: reentrancy, access control, input validation (whitelist checks), token power manipulation and governance safe behaviors.
- Review of solidity best practices: documentation, CEI patterns, variable naming, variable and function visibility, etc…

3. Severity classification.

High: Unexpected logic contract behavior, contract damage, direct loss of funds.

Medium: Indirect loss of funds, unexpected logic behaviors but without breaking the governance process.

Low: Best practices violation or minor risks.

Informational: Just information about possible risks if we want to update the contract in the future or things to take in mind that for now are not important. But may be reviewed in the future if new changes come.

4. Findings Summary.

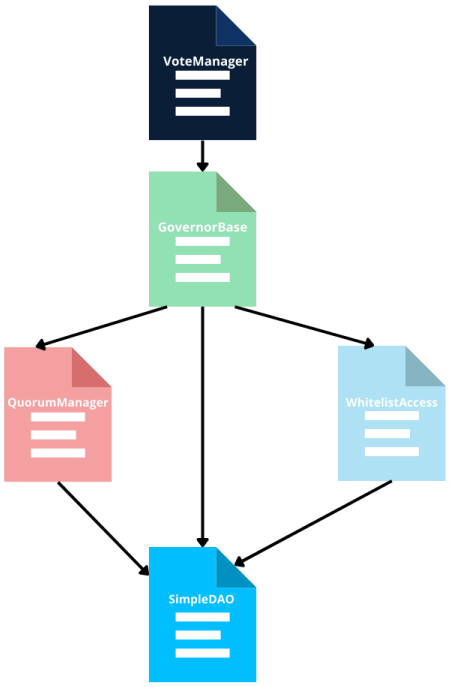
ID	Title	Severity	Status
V-01	Reentrancy Vulnerability	High	Fixed
V-02	Missing target validation	Low	Fixed
V-03	Voting power transfer	Medium	Fixed
V-04	Missing quorum	Medium	Fixed
V-05	State bloat attack or storage griefing	Informational	Not fixed

5. Context of new base improvements and architecture.

V2 base improvements and architecture.

Once all vulnerabilities were detected, in order to make all these security improvements possible we decided to refactor the V1 version. The final V2 works the same as V1, but its architecture is completely new. We needed to create a GovernorBase contract where all the governance core logic is placed, and then create multiple abstract contracts to inherit security policies and logic to make the governance process secure.

The V2 version architecture diagram looks like:



We decided to refactor V1 and create a new architecture in order to separate responsibilities and make a clear and scalable code. This new V2 version allows a programmer to extend the code and add new policies such as timelocks without changing too much the base code placed in GovernorBase.

To accomplish this goal, GovernorBase has function hooks which allow the programmers to add extra logic and functionality to the GovernorBase. These hooks are virtual, so can be overridden by any child inheriting GovernorBase. An example of such declaration founded at the end of the GovernorBase is:

```
function _beforePropose(
    address _target,
    bytes calldata _callData
) internal virtual {}
```

These hooks are executed before any action the DAO is performing. For example in our case, we are interested in adding two hooks: _beforePropose() and _beforeExecute(). This means that any time the DAO is executing a createProposal() or execute() function, hooks will be executed first to perform

some previous checks or logic. In the vote() function there is no need to do that because we did not detect any possible malicious behavior.

To apply this hook structure, we need to make function wrappers to the following DAO functions: createProposal(), vote() and execute(). A function wrapper is a public function that wraps an internal or private function of the contract. Generally with similar names to the internal function it is calling. In this case, the original V1 createProposal(), vote() and execute() functions were renamed with an underscore "_" and mark them as private. So now, no one can call these functions externally. Now we have privatized these internal functions, which contain the core logic of proposal creation, voting, and proposal execution, we create functions wrappers to wrap them. The wrappers are called exactly as the internal functions but without the underscore "_" and they are marked as external to be called by anyone. This function's wrappers are acting as "function's frontend". Here is where we can apply modular and extensive custom logic to add extra policies or code to accomplish some goals such as performing security checks, extra logic to extend the base internal functions, etc… By doing this, a programmer can easily extend the createProposal(), vote() and execute() DAO's functions to add more logic.

So the mentioned hooks are placed inside the wrappers functions and called before the internal function. An example of that:

```
function execute(uint256 _proposalId) external nonReentrant {
    _beforeExecute(_proposalId);
    _execute(_proposalId);
}
```

See that execute() is the wrapper that wraps _execute() function, which is the internal execute function that contains the execute proposal core logic. Before _execute() function there is a hook called _beforeExecute() that is being executed before the internal _excute() function. The hook can be overridden in child contracts and extend the hook's logic and functionality.

It is inside these hooks where we are going to add extra checks and logic to achieve the security objectives we have with this governor DAO.

On the other hand, we created an abstract contract for each new policy we want to add: QuorumManager to handle the quorum, VoteManager to handle proposal votes and WhitelistAccess to handle a proposal target whitelist. These abstract contracts inherit from the GovernoBase contract. Therefore, they can override its hooks to extend their functionality.

The final SimpleDAO contract you can see in the diagram, inherits all the GovernorBase and policies functionalities to merge them in a contract. SimpleDAO is the final child of this structure and the final contract. It represents the V2 version.

Another key aspect of this V2 version to point out, is that we migrated the voteYes and voteNo of the Proposal struct to another struct placed in an abstract contract. This abstract contract is VoteManager.

VoteManager handles the votes a proposal has. Additionally we need to add another variable to the proposal data which is the token supply amount there was at the moment of the proposal creation. This information will be very useful when we add a quorum policy in the QuorumManager contract. So voteYes, voteNo and the new proposalInitialTokenSupply variables are now placed in a new struct called Votes inside the VoteManager.

See how it looks this new struct:

```
struct Votes {
    uint256 voteYes;
    uint256 voteNo;
    uint256 proposalInitialTokenSupply;
}
```

This migration allows control of the proposal votes data separately and it will become easier for future audits and possible code extensions. Moreover, this migration reduced the load in the main Proposal struct data in the GovernorBase and reduced possible gas consumptions due to multiple storage access.

Finally this migration will become very handy when we want to access the proposal votes from the QuorumManager. So, the main purpose of this VoteManager is to reduce the Proposal data load and to modularize the contract architecture in order to be more efficient in terms of gas consumptions, security audits and future code extensions if desired. This contract modularization in an abstract contract also allows us to not stick all the code in the same contract and separate responsibilities in various contracts. Thus, it will be more readable and you will understand better what each contract part is doing.

All these abstract contracts and hooks architecture will define the base to start optimizing and securing the V1 version and finally crafting the final V2 version.

6. Detailed Findings.

V-02 Missing target validation

- **Severity:** Low.
- **Location:** createProposal().
- **Description:** Anyone can propose a proposal without verifying the target.
- **Impact:** An attacker can start spamming proposals with random targets and not having an idea about what the governance process is working on.
- **Recommendation:** Adding a type of validation or a whitelist to allow or ban targets via proposals. We want the governance participants to have control about the allowed targets the DAO can work with.

Analyzing the code, we discovered that any target and any calldata can be supplied to the proposal. In our case this is not a great risk because we are not using call with value. So a malicious ETH transfer can't occur. However, to prevent spamming targets randomly during the governance, we should introduce a solution where the players are who handles the allowed targets in the governance. Then the players become the main judges of the DAO.

To achieve this we need to whitelist the targets that are allowed to interact. The whitelist policy must be implemented in the createProposal function where a new proposal can be created and inserted to the system.

You can see how the V1 createProposal version looks like:

```
function createProposal(
    address _target,
    bytes calldata _callData,
    bytes32 _description
) external returns (uint256) {
    require(token.balanceOf(msg.sender) > 0, "Only holders can propose");

    proposals.push(
        Proposal({
            proposer: msg.sender,
            target: _target,
            deadline: uint64(block.timestamp + VOTTING_PERIOD),
            snapshotBlock: uint64(block.number),
            callData: _callData,
            description: _description,
            voteYes: 0,
            voteNo: 0,
            executed: false
        })
    );

    uint256 id = proposals.length - 1;

    emit ProposalCreated(id, msg.sender, _description);

    return id;
}
```

At the beginning we thought about setting up an admin who has the authority to introduce new targets to the whitelist. But doing this will break decentralization as an admin has a concentrated power over the DAO. Moreover the admin could lose their keys or be stolen and the dao probably will stop working as expected.

So we will stay focused on the first option where the DAO players can propose targets and judge them in a votation process using proposals for that. This will bring the DAO players the power to choose the targets they want to work with.

To do this, we could stick all the logic in the same contract. But for our convenience it is better that we make a specific module for this functionality/policiy because we don't really want to mix responsibilities in the same contract. This will be much more obvious when we add quorum policies.

So we created a WhitlistAccess abstract contract where you can check how this contract verifies if the target is allowed or not. This contract inherits from GovernorBase, so we can override the _beforePropose hook and add inside it the logic to check if the target is whitelisted. The check logic is inside a private function named _checkAccess(). So, the WhitelistAccess' hook includes a call to this private function and then a super call to follow the C3 linearization (this is because there can be more _beforePropose hooks to call from other parents).

The overridden _beforePropose() hook in WhitelistAccess looks like:

```
function _beforePropose(
    address _target,
    bytes calldata _callData
) internal virtual override {
    _checkAccess(_target, _callData);
    super._beforePropose(_target, _callData);
}
```

The _checkAccess function in WhitelistAccess looks like:

```
function _checkAccess(
    address _target,
    bytes calldata _callData
) private view {
    require(_callData.length >= 4, "callData too short");

    if (_target == address(this)) {
        bytes4 selector = bytes4(_callData[:4]);
        require(
            s_allowedSelectors[_target][selector],
            "Not allowed selector"
        );
    } else {
        require(s_allowedTargets[_target], "Target not whitelisted");
    }
}
```

You can see how it checks both the selector and the target. The selector verification is a special case to restrict the function/selector that can be called inside the SimpleDAO code via proposal. As we said, we are interested in that any player can call allowTarget() and disallowTarget() functions to allow or ban targets. But these functions are not allowed to be called freely. Instead, they must be called using proposals. Allowing these two functions to be called from a proposal execution will make that any player can make a proposal to whitelist/allow a target. And that is the behavior we really wanted from the beginning.

If the target is the SimpleDAO (this) then we check if the selector encoded in the calldata is allowed. There are only these two functions/selectors allowed in the own SimpleDAO address. You can check in the constructor how we set up this information:

```
constructor() {
    bytes4 allowTargetSelector = bytes4(
        keccak256(bytes("allowTarget(address)"))
    );
    bytes4 disallowTargetSelector = bytes4(
        keccak256(bytes("disallowTarget(address)"))
    );

    s_allowedSelectors[address(this)][allowTargetSelector] = true;
    s_allowedSelectors[address(this)][disallowTargetSelector] = true;
}
```

See that in the constructor we initialize the s_allowedSelector mapping with both allowTarget() and disallowTarget() from the WhitelistAccess. Remember that SimpleDAO inherits from WhitelistAccess, so the code inside WhitelistAccess is also the code of the SimpleDAO contract. That is because we are indexing these two allowed selectors with address(this) referencing to our DAO address contract.

Keep in mind that the selector checks are only performed if the incoming target is the own DAO address. If not, then we check if the target is indeed whitelisted or not. If not we revert. If it is allowed then we pass the check.

Building this contract structure we solved the random proposal targets, and now the DAO's players have the power to determine through a votation process if a target can be allowed or not. That's all for this case.

V-01 Reentrancy Vulnerability

- **Severity:** High.
- **Location:** execute().
- **Description:** A reentrancy vulnerability occurs when a contract function uses Solidity's low-level call function without properly validating or securing the interaction. This can allow the function to be executed multiple times before the initial execution is completed. This is why it is called a reentrancy vulnerability, because the execution flow can be reentered multiple times within the function that contains the external call.
- **Impact:** An attacker can craft a malicious contract with a function that points to the execute DAO function. Then, the attacker will create a proposal with the malicious contract as a target and its attack function as the calldata. When the proposal is executed, a reentrancy vulnerability will happen between the DAO and the malicious contract. This may lead to contract loss of funds or contract damage.
- **Recommendation:** Apply the pattern CEI (Checks-Effects-Interactions) and OpenZeppelin nonReentrant mutex.

We successfully identified a potential reentrancy Vulnerability in V1 version inside execute() function.

```
function execute(uint256 proposalId) external {
    require(proposalId < proposals.length, "Proposal index out of bounds!");

    Proposal storage proposal = proposals[proposalId];

    require(
        block.timestamp >= proposal.deadline,
        "The proposal voting has not finished!"
    );
    require(
        !proposal.executed,
        "The proposal has already been executed! Can not reexecute again"
    );
    require(proposal.voteYes > proposal.voteNo, "Proposal not aproved");

    (bool success, ) = proposal.target.call(proposal.callData);
    require(success, "The target call has failed!");

    proposal.executed = true;

    emit Executed(proposalId);
}
```

See that we are updating the proposal state after the target call. This breaks the CEI pattern. During the call, we are not sending ETH or retrieving it to any user. The problem is that if we want to add more functionality before the call, such as writing some contract's state variable or even calling external calls to another contract, then the reentrancy attack could set an inconsistent contract state. This attack may break the contract's state and its normal behavior.

Another structural problem, but solved with the previous whitelist solution, is that in the createProposal function you can provide any target address you want and also, the function you want to execute.

Considering the vulnerable V1 version, a malicious user can create a custom contract with a malicious function, then create a proposal, supply the malicious contract address in the target parameter and the malicious function encoded in the callData parameter. When the proposal is executed, an execution flow will occur between both the execute() DAO function and the malicious contract function.

The result: more than one successful call to the execute() DAO function performed using the same proposalId in the same transaction. In little words, a proposal is executed more than one time. This happens because we are able to call the target call function more than one time before the proposal state is updated.

How can we prove this attack? Performing a visual check, you can see the reentrancy vulnerability pattern: first call then updates the state. This is what is happening here so you can suspect that this function is indeed vulnerable. Moreover, the execute() function does not follow the CEI pattern.

To test this, we are going to craft a **Malicious** contract with a malicious function named **attack()**. The attack function keeps track of a counter and tries to reenter the execute() DAO function with the same proposal id. If the timesEntered counter is greater than one during the transaction means that we reentered successfully to the execute() DAO function with the same proposal id. Thus our DAO contract is vulnerable to reentrancy attacks. Here you can see the attack function placed inside our Malicious contract.

```
function attack() external {
    timesEntered += 1;

    //Try to reenter
    if (timesEntered < 2) {
        //Execute
        dao.execute(proposalId);
    }
}
```

We only need to prove that it enters two times. But theoretically, we can reenter till all the transaction gas runs out.

The next step is to set up a test that creates a proposal using our Malicious contract as a target and the attack function as a callData. Then execute the proposal and check the state of the timesEntered counter inside our Malicious contract.

This is the test for checking if the vulnerability is present:

```
function testReentrancy_Vulnerable() public {
    address target = address(malicious);
    bytes memory callData = malicious.encodedAttackCall();
    bytes32 description = keccak256("desc");
    uint256 proposalId;

    vm.prank(userA);
    proposalId = dao.createProposal(target, callData, description);

    malicious.setProposal(proposalId);

    vm.roll(block.number + 1);

    vm.prank(userA);
    dao.vote(proposalId, true);

    vm.warp(block.timestamp + 4 days);

    dao.execute(proposalId);

    assertGt(malicious.timesEntered(), 1);
    assertEq(malicious.timesEntered(), 2);
}
```

You can see two asserts at the end of the test. These asserts detect if the internal counter (timesEntered) inside the malicious contract is greater than one. If this happens that means that we successfully reentered the DAO execute function with the same proposal id.

When we run the test, we see the following results:

```
alejandro@alejandro:~/solidity/foundry/simple-dao$ forge test --match-path test/T_Reentrancy.t.sol -vv
[⠒] Compiling...
[⠢] Compiling 1 files with Solc 0.8.24
[⠆] Solc 0.8.24 finished in 4.16s
Compiler run successful!

Ran 1 test for test/T_Reentrancy.t.sol:T_Reentrancy
[PASS] testReentrancy_Vulnerable() (gas: 244292)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 106.65ms (3.18ms CPU time)

Ran 1 test suite in 114.04ms (106.65ms CPU time): 1 tests passed, 0 failed, 0 skipped (1 total tests)
```

We successfully passed the test. So, that means we exploited the reentrancy vulnerability because the timesEntered counter in the Malicious contract is greater than one.

We can see more clearly what happened inspecting the execution stack trace:

```
├─ [39094] SimpleDAO::execute(0)
│   ├─ [34878] Malicious::attack()
│   │   ├─ [0] console::log("Attack called! timesEntered =", 1) [staticcall]
│   │   │   └─ ← [Stop]
│   │   ├─ [5990] SimpleDAO::execute(0)
│   │   │   ├─ [1774] Malicious::attack()
│   │   │   │   ├─ [0] console::log("Attack called! timesEntered =", 2) [staticcall]
│   │   │   │   │   └─ ← [Stop]
│   │   │   │   └─ ← [Stop]
│   │   │   ├─ emit Executed(id: 0)
│   │   │   └─ ← [Stop]
│   │   └─ ← [Stop]
│   ├─ emit Executed(id: 0)
│   └─ ← [Stop]
├─ [491] Malicious::timesEntered() [staticcall]
│   └─ ← [Return] 2
├─ [491] Malicious::timesEntered() [staticcall]
│   └─ ← [Return] 2
└─ ← [Stop]
```

This is what happened: We created a proposal with the Malicious contract as a target and its attack function as the callData. Then, save this proposal id to identify the malicious proposal. When we execute this proposal, the DAO contract calls the attack function in our Malicious contract. Inside the attack function in Malicious, the counter timesEntered is incremented by one, then calls the SimpleDAO execute function with the same proposal id. During this time, the DAO contract has not already updated the proposal.executed property, which it is updated after the call. Thus a recursive loop starts happening: the stack trace continues in a recursive loop where both the execute DAO function and the Malicious contract attack function are executed till the gas runs out. Thus the following calls to the DAO execute function with that proposal id is valid because the proposal.executed state

is still false. So this require condition is met and the DAO contract re-executes the execute() function.

See the trace: DAO execute() -> Malicious attack() -> DAO execute() -> Malicious attack() and then stops all and finishes the transaction as correct.

We also added another test to check if the vulnerability is patched.

```
function testReentrancy_Patched() public {
    address target = address(malicious);
    bytes memory callData = malicious.encodedAttackCall();
    bytes32 description = keccak256("desc");
    uint256 proposalId;

    //Move one block forward
    vm.roll(block.number + 1);

    //Create proposal
    vm.prank(userA);
    proposalId = dao.createProposal(target, callData, description);

    //Set proposal in malicious contract
    malicious.setProposal(proposalId);

    //Vote
    vm.prank(userA);
    dao.vote(proposalId, true);

    //Execute
    vm.warp(block.timestamp + 4 days);
    vm.expectRevert("The target call has failed!");
    dao.execute(proposalId);

    assertEq(malicious.timesEntered(), 0);
}
```

In the patched test we expect a revert during dao.execute() because a revert will imply that the reentrancy is detected and aborted. The expected revert message is: “The target call has failed!”, we are really expecting the execute’s call function to, we will explain why later.

If we run the test with the vulnerable version, the revert is not happening because we reenter the dao successfully and didn’t revert. See the timesEntered counter is 2:

```
Logs:
  Attack called! timesEntered = 1
  Attack called! timesEntered = 2

Suite result: FAILED. 1 passed; 1 failed; 0 skipped; finished in 43.25ms (534.33µs CPU time)

Ran 1 test suite in 45.29ms (43.25ms CPU time): 1 tests passed, 1 failed, 0 skipped (2 total tests)

Failing tests:
Encountered 1 failing test in test/T_Reentrancy.t.sol:T_Reentrancy
[FAIL: next call did not revert as expected] testReentrancy_Patched() (gas: 247087)

Encountered a total of 1 failing tests, 1 tests succeeded
```

We also added a log in the attack function to keep track of the counter timesEntered increments.

We see that the counter increments two times during the execution. That means we reentered two times, and the test failed because no revert was triggered. Thus, the reentrance was executed.

How to solve the reentrancy problem?

This vulnerability is simple to solve, just we need to break the pattern: first call then update. In our case what is happening is: first call the proposal external function and then update the proposal state. This pattern needs to be switched to: first update the proposal state and then call the proposal external function.

Just by doing this permutation will solve the vulnerability, but it's also recommended to apply the OppenZepeling nonReentrant mutex in order to add an extra security layer to our contract. So we are going to implement both solutions.

First lets put the **proposal.executed = true** statement before the call:

```
proposal.executed = true;

(bool success, ) = proposal.target.call(proposal.callData);
require(success, "The target call has failed!");
```

Just by doing this the problem is solved and the CEI pattern starts being followed. When we execute the test again we will notice that the testReentracy_Vulnerable which detects the reentracy vulnerability now fails. That means that the second call reverted internally with the **require(!proposal.executed, "The proposal has already been executed! Can not reexecute again");** This revert provokes that

the **proposal.target.call** in the execute function returns false. We can see that the log shows us that timesEntered = 1.

```
[FAIL: The target call has failed!] testReentrancy_Vulnerable() (gas: 241830)
Logs:
  Attack called! timesEntered = 1
```

On the other hand, the testReentrancy_Patched test which detects if the vulnerability is solved passed successfully. This test expects a revert, and because the revert is executed and detected, the test passes successfully. This is a symptom that the call failed in the second entry and reverted the transaction.

```
Ran 2 tests for test/T_Reentrancy.t.sol:T_Reentrancy
[PASS] testReentrancy_Patched() (gas: 245357)
Logs:
  Attack called! timesEntered = 1
```

But the question here is: ¿Why does the system enter 1 time in the Malicious contract, and in the test we assert that timesEntered is equal to 0?. Well, this is happening because the transaction was reverted at the end and all storage changes were discarded. To see this more clearly we need to check the testReentrancy_Patched test execution trace:

```
├─ [34285] SimpleDAO::execute(0)
│   ├─ [30853] Malicious::attack()
│   │   ├─ [0] console::log("Attack called! timesEntered =", 1) [staticcall]
│   │   │   └─ ← [Stop]
│   │   ├─ [1956] SimpleDAO::execute(0)
│   │   │   └─ ← [Revert] The proposal has already been executed! Can not reexecute again
│   │   └─ ← [Revert] The proposal has already been executed! Can not reexecute again
│   └─ ← [Revert] The target call has failed!
├─ [2491] Malicious::timesEntered() [staticcall]
│   └─ ← [Return] 0
└─ ← [Stop]
```

See the trace: First executes SimpleDAO execute() function -> then the Malicious contract attack() function and prints timesEntered = 1 -> then attack() function calls SimpleDAO execute() function. Here in the second execution of execute() the transaction reverts thanks to the execute() condition: **require(!proposal.executed, "The proposal has already been executed! Can not reexecute again");** See that it reverts because the proposal was marked as executed in the previous call to execute(), and now it figures as executed, so the requiere reverts the transaction. This revert is returned to its parent execution which comes from the call function from SimpleDAO. Then, the call in SimpleDAO detects that its call has failed and returns false. And that is because we appreciate

the final revert is **"The target call has failed!"** and not **"The proposal has already been executed! Can not reexecute again".** We catch the final revert triggered, not the previous ones.

This execution trace: execute() -> attack() -> execute() -> REVERT is being performed in the same transaction. So whatever change made to the storage along with this transaction will be discarded because the transaction where all these changes were performed was reverted. This is the main reason because we see the log timesEntered = 1, and then the assert test timesEnters == 0 pass successfully. Because timesEntered = 1 was only temporal during the transaction. When the transaction was reverted, all changes made to the storage, like the timesEntered state update was also reverted and discarded. See at the end of the execution trace, when the transaction was already reverted, how the test calls Malicious.timesEntered() function and returns 0.

Moreover, to add another extra security layer we are going to implement the reentrancy guard from OpenZeppelinn. We will use the nonReentrant modifier from ReentrancyGuard.sol which acts as a mutex. This modifier helps us to avoid and neutralize reentrancy in the functions we put in it.

Just import it:

```
import {ReentrancyGuard} from "@openzeppelin/contracts/utils/ReentrancyGuard.sol";
```

And apply the modifier to the execute function:

```
function execute(uint256 proposalId) external nonReentrant {
```

If we run the test, the result will not change. Just putting the proposal.execute = true before the call solved the problem. But this adds another extra security layer which is very safe and effective to avoid reentrancy in the function. This modifier works as a mutex in the function we specify it. The functions marked as nonReentrant will not be re-executed in the same transaction.

V-03 Voting power transfer

- **Severity:** Mid.
- **Location:** vote().
- **Description:** This is not a vulnerability by itself but it can provoke unexpected logical behaviors outside the governor logic. A voting power transfer occurs when a user votes for a proposal and in the same block rotates its tokens to another account they control. This allows different accounts to vote on a proposal with the same tokens. Thus, the votes are duplicated.
- **Impact:** An attacker using multiple accounts can gain significant influence over the proposal votes.
- **Recommendation:** Replace ERC20 with ERC20Votes and use token.getPastVotes() instead of token.balanceOf() functions.

During the testing phase we detected a voting power transfer vulnerability. This vulnerability allows a user to vote and immediately transfer its tokens to another controlled user and vote again for the proposal. Thus, a user can vote massively on a proposal using the same tokens but rotating them into different accounts. Using this vulnerability a user can control the proposal voting power.

To solve this problem, we are going to use the ERC20Votes token instead of ERC20. The ERC20Votes keeps track of the user's balances in history. So each time we vote on a proposal we will use the function getPastVotes which checks the token amount the user had in a certain block of the chain. In this case, we will use as a reference the block when the proposal was created. This information is saved in the snapshotBlock proposal struct variable when the proposal is created. The snapshotBlock info was not present in the original V1 proposal struct. This variable was added during the audit in order to make possible this security improvement.

```
function vote(uint256 proposalId, bool support) external {
    require(proposalId < proposals.length, "Proposal index out of bounds!");

    Proposal storage proposal = proposals[proposalId];
    uint256 voterBalance = token.getPastVotes(
        msg.sender,
        proposal.snapshotBlock
    );
    require(voterBalance > 0, "Yo dont have tokens to vote");
```

Moreover, using this ERC20Votes solution we can solve the flash-loan vulnerability in which a user can receive tokens in the same transaction or block we are creating

the proposal. This could allow a user to create a proposal with a non real voting power. Using this method in the createProposal we force a user to have in possession tokens at least in the last block.

```
function createProposal(
    address _target,
    bytes calldata _callData,
    bytes32 _description
) external returns (uint256) {
    require(
        token.getPastVotes(msg.sender, block.number - 1) > 0,
        "Only holders can propose"
    );
```

V-04 Missing Quorum

- **Severity:** Mid.
- **Location:** execute().
- **Description:** This is not a vulnerability, but it can produce unexpected proposal behaviors. For example, a unique user can vote and execute the same proposal by itself. This behavior allows a user to have a great power over the proposal execution. We don't want that, we want that more users were implied in the votation process before executing a proposal.
- **Impact:** A single user can vote and execute a proposal.
- **Recommendation:** Implement a quorum system to prevent a single user from having excessive power.

The original V1 contract did not have a quorum. This could be dangerous because if a single user votes a proposal and then executes it, it will have success. We don't really want a single player to have such power over the votation process. Moreover, without a quorum if a proposal has voteYes > 0 or voteNo > 0, no matter the amount of power concentrated in one of them, the proposal execution will decide if the proposal passes or not.

To solve this, we need to implement a quorum policy. And that is because we make a QuorumManager which handles the necessary logic to implement a quorum mechanism.

The QuorumManager contract inherits from GovernorBase, so the way we are going to implement a quorum policy in our GovernorBase contract is using the hooks we mentioned before. This time we are going to use the _beforeExecute() hook placed in GovernorBase. So, we are going to override this hook in the QuorumManager and implement inside this hook the quorum logic. Doing that, each time we execute a proposal, the quorum logic will be triggered because in the execute() function in GovernorBase the _beforeExecute() hook is placed before the execute core logic.

This is the execute function in GovernorBase, see that the _beforeExecute() hook is before the _execute() function which contains the execute core logic:

```
function execute(uint256 _proposalId) external nonReentrant {
    _beforeExecute(_proposalId);
    _execute(_proposalId);
}
```

This is the _beforeExecute() overridden hook in the QuorumManager:

```
function _beforeExecute(uint256 _proposalId) internal virtual override {
    _checkQuorum(_proposalId);
    super._beforeExecute(_proposalId);
}
```

See that we call _checkQuorum which contains the quorum logic and then a super to call whatever overridden _beforeExecute() hook there will be next, along the C3 linearization.

The _checkQuorum() function is just a simple quorum logic:

```
function _checkQuorum(uint256 _proposalId) private view {
    (
        uint256 voteYes,
        uint256 voteNo,
        uint256 proposalCreationTokenSupply
    ) = _getProposalVoteInfo(_proposalId);

    require(
        voteYes + voteNo >=
            (proposalCreationTokenSupply * QUORUM_BP) / 10_000,
        "Quorum not reached"
    );
}
```

We are using basis points to do the calculation. 10_000 basis points represents 100%, QUORUM_BP is equal to 1000 basis points, which is 10%. In little words, we are using a 10% quorum.

See that we are obtaining the proposal votes amount and the token supply at the proposal creation time using the _getProposalVoteInfo() which is a function from the previously mentioned and explained VoteManager. So here in the QuorumManager we are accessing the VoteManager to retrieve the proposal votes info. See how modularization works very well here.

Splitting these policies in abstract contracts allows modularization and separate responsibilities in isolated pieces.

V-05 State bloat attack or storage griefing

- **Severity:** Informational.
- **Location:** createProposal().
- **Description:** A state bloat attack implies that a malicious user starts filling the contract storage with garbage bytes with the main goal to make the contract unusable due to high gas reading costs.
- **Impact:** A malicious user could theoretically trigger a gas DoS attack.
- **Recommendation:** It will depend on the contract design. But avoid linear access to storage variables if no measure is implemented.

We also have a problem where a malicious user can create massive proposals with no sense. Doing this the proposal's array could increase massively. This is not a logical vulnerability, but a memory manipulation. This kind of attack allows a malicious user to fill the storage with garbage and if no one can avoid this behavior

the contract storage will be plenty of bytes with no purpose. This can trigger future DoS due to high gas consumption, because of the length of the information the contract is trying to read from the storage. Thus, the contract could be hypothetically unable to use if the number of bytes is large enough to burn all the gas.

In our case this does not have a negative impact, because all proposals are indexed and accessed by their index, similar to a map. So no matter if a malicious attacker starts filling this array with large proposals because normal users will access the proposal by their known index. We never trigger linear searches in the contract logic, so we don't have to be worried about that.

In this case we are only indexing directly by the proposal Id. But if we want to read all the proposals array, then we will have a problem with the gas costs because we will read all the proposals including those which have a huge callData length. But again, this is theoretical, in our contract never happens.

The only impact is the gas cost to create such proposals, but again, normal users will not do that. If a malicious user wants to do this, then he will pay the costs in gas. Moreover, the whitelist access policy we implemented is an effective filter to avoid the creation of nonsense-proposals. But it is not definitive.

That concludes all detected vulnerabilities and their corresponding security solutions.

7. Changes made in V2.

You see that the V2 version has a completely new architecture but its functionality is the same. We added abstract contracts to modularize the project and to separate responsibilities. All these changes were necessary in order to implement the security and gas consumption improvements without breaking the readability of the code. Also avoiding putting all the responsibilities in the same contract, which is what we were trying to prevent while introducing these changes.

So here you have the changes introduced in the GovernorBase abstract contract, which contains all the governor core logic. All other responsibilities such as quorum, whitelist or votes were delegated to the corresponding abstract contracts.

A part form all the functions wrappers and hooks explained at the beginning of this report, the changes made in the governor functions are:

Proposal struct: We added **uint256 snapshotBlock** to save in the proposal the block number when it was created. We separated **uint256 voteYes** and **uint256 voteNo** from Proposal struct and placed now in the Vote struct in VoteManager contract. We added in this Vote proposal a new variable **uint256 proposalInitialTokenSupply** to register the token supply there was in circulation at the proposal time creation.

createProposal(): We change the token.balanceOf() to token.getPastVotes() to retrieve the token amount a user has in a block. Then at the end a new call to **_initializeNewProposalVotes(proposalId, proposalCreationTokenSupply),** which initializes a new proposal votes in the VoteManager.

vote(): We change the token.balanceOf() to token.getPastVotes() to retrieve the token amount a user has in a block. In this case, the token amount that the user has at the time the proposal is created. Then we replaced the vote-adding logic with the call **_addVotesToProposal(_proposalId, voterBalance, _support)**, which registers the votes for the proposal in the VoteManager. See here how we are separating responsibilities. In the V1 version we execute all the logic in the vote() function. In V2, we perform the necessary checks to determine whether a user is allowed to vote, and if they pass, we add the user's votes to the VoteManager.

execute(): In V2 version, instead of getting the votesYes and voteNo from the governor Proposal struct, we retrieve them from the VoteManager through *getProposalVotes(*proposalId) call. We again access the VoteManager to get the proposal votes.

And that's all the changes we made in the governor core code, which in V2 is placed in GovernorBase abstract contract. In the V1, all logic was placed in the same contract. In V2, the governor core is in the GovernorBase and all policy extensions are added via abstract contacts. You can check QuorumManager, WhitelistAccess and VoteManager logic on your own, but their main parts were explained during this audit. So that's all for the V2 version.

8. Minor changes.

We changed some statements or code structure in order to follow Solidity best practices, such as CEI pattern, changes in visibility of functions and variables and code documentation. We also re-ordered the requires statements to trigger those with the highest relevance first, reducing the execution of some other requires that can be performed later. This will help to save some gas.

Style rules: We renamed all the variables applying the following prefix rule: Add a “s_” prefix to storage variable, add a “i_” prefix to immutable variables and add “_” prefix to parameters. Constant variables are in capital letters. We also added the “_” prefix to internal and private functions. Also added documentation to variables and functions.

We may also have replaced **requiere** reverts with **error** reverts. This will reduce gas costs and will make debugging more easy. However we decided to maintain **requiere** reverts as it is and maybe change them by **error** reverts in future versions.

9. Post-Audit Summary.

All identified issues have been addressed and solved. No critical vulnerabilities or issues remain. We refactor the V1 version to build the V2 version. The refactorization was necessary to separate responsibilities, apply all new security measures in an organized way and to reduce gas consumption. This V2 new version is more readable, auditable and extensible than the previous one.