CANTINA

Venus protocol: governance contracts

Competition

April 25, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

A competition provides a broad evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While competitions endeavor to identify and disclose all potential security issues, they cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities, therefore, any changes made to the code would require an additional security review. Please be advised that competitions are not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Venus Protocol pushed the edges of DeFi finance through its composition of two pre-existing solutions (the stablecoin minting facility introduced by Maker and algorithmic money markets developed by Compound) simplified the user experience and provided core capabilities that enables DeFi to flourish in a single application.

From Mar 22nd to Apr 5th Cantina hosted a competition based on VenusProtocol/governance-contracts. The participants identified a total of **64** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 0
- Medium Risk: 1
- Low Risk: 57
- Gas Optimizations: 0
- Informational: 6

The present report only outlines the **critical**, **high** and **medium** risk issues.

3 Findings

3.1 Medium Risk

3.1.1 Some proposals can be queued when governanceexecutor is paused

Submitted by zigtur, also found by SBSecurity, TamayoNft and deth

Severity: Medium Risk

Context: OmnichainGovernanceExecutor.sol#L297

Description: `OmnichainGovernanceExecutor` automatically queues the proposals received as message through LayerZero.

The pausing mechanism of `OmnichainGovernanceExecutor` is designed to deny the queueing of proposals when the contract is paused.

However, specific conditions make the queueing of proposals still possible when the contract is paused.

Impact: Proposals can be queued when `GovernanceExecutor` is paused.

Proof of concept: A message that includes a proposal is received from LayerZero and `_blockingLzReceive` is executed. The `whenNotPaused` modifier ensures contract is not paused and a call is made to `nonblockingLzReceive` to queue the proposal. When this call fails, a "proof" of message failure is stored (see `@POC` comments):

```
function _blockingLzReceive(
    uint16 srcChainId_,
    bytes memory srcAddress_,
    uint64 nonce_,
    bytes memory payload_
) internal virtual override whenNotPaused {
    uint256 gasToStoreAndEmit = 30000; // enough gas to ensure we can store the payload and emit the event

    require(srcChainId_ == srcChainId, "OmnichainGovernanceExecutor::_blockingLzReceive: invalid source chain
↪  id");

    (bool success, bytes memory reason) = address(this).excessivelySafeCall(
        gasleft() - gasToStoreAndEmit,
        150,
        abi.encodeCall(this.nonblockingLzReceive, (srcChainId_, srcAddress_, nonce_, payload_))
    );
    // try-catch all errors/exceptions
    if (!success) {
        bytes32 hashedPayload = keccak256(payload_);
        failedMessages[srcChainId_][srcAddress_][nonce_] = hashedPayload; // @POC: proof of message failure
        emit ReceivePayloadFailed(srcChainId_, srcAddress_, nonce_, reason); // Retrieve payload from the src
↪  side tx if needed to clear
    }
}
```

Here, the owner of the contract pauses the contract through `OmnichainGovernanceExecutor.pause()`.

The LZ failed message can be retried through the `retryMessage` function inherited from `NonblockingLzApp`. This function will call `_nonblockingLzReceive`:

```
function retryMessage(
    uint16 _srcChainId,
    bytes calldata _srcAddress,
    uint64 _nonce,
    bytes calldata _payload
) public payable virtual {
    // assert there is message to retry
    bytes32 payloadHash = failedMessages[_srcChainId][_srcAddress][_nonce];
    require(payloadHash != bytes32(0), "NonblockingLzApp: no stored message");
    require(keccak256(_payload) == payloadHash, "NonblockingLzApp: invalid payload");
    // clear the stored message
    failedMessages[_srcChainId][_srcAddress][_nonce] = bytes32(0);
    // execute the message. revert if it fails again
    _nonblockingLzReceive(_srcChainId, _srcAddress, _nonce, _payload);// @POC: call to nonblocking
    emit RetryMessageSuccess(_srcChainId, _srcAddress, _nonce, payloadHash);
}
```

The issue is that `_nonblockingLzReceive` doesn't require the contract to be unpaused. It will queue the proposal even if the contract is paused:

```
function _nonblockingLzReceive(uint16, bytes memory, uint64, bytes memory payload_) internal virtual override
↪  { // @POC no pausing check
    (bytes memory payload, uint256 pId) = abi.decode(payload_, (bytes, uint256));
    (
        address[] memory targets,
        uint256[] memory values,
        string[] memory signatures,
        bytes[] memory calldatas,
        uint8 pType
    ) = abi.decode(payload, (address[], uint256[], string[], bytes[], uint8));
    require(proposals[pId].id == 0, "OmnichainGovernanceExecutor::_nonblockingLzReceive: duplicate proposal");
    require(
        targets.length == values.length &&
            targets.length == signatures.length &&
            targets.length == calldatas.length,
        "OmnichainGovernanceExecutor::_nonblockingLzReceive: proposal function information arity mismatch"
    );
    require(
        pType < uint8(type(ProposalType).max) + 1,
        "OmnichainGovernanceExecutor::_nonblockingLzReceive: invalid proposal type"
    );
    _isEligibleToReceive(targets.length);

    Proposal memory newProposal = Proposal({
        id: pId,
        eta: 0,
        targets: targets,
        values: values,
        signatures: signatures,
        calldatas: calldatas,
        canceled: false,
        executed: false,
        proposalType: pType
    });

    proposals[pId] = newProposal;
    lastProposalReceived = pId;

    emit ProposalReceived(newProposal.id, targets, values, signatures, calldatas, pType);
    _queue(pId);
}
```

Recommendation: Consider setting the `whenNotPaused` modifier on `_nonblockingLzReceive` instead of `_blockingLzReceive`. This will deny replaying previously failed proposal queueing when the contract is paused.