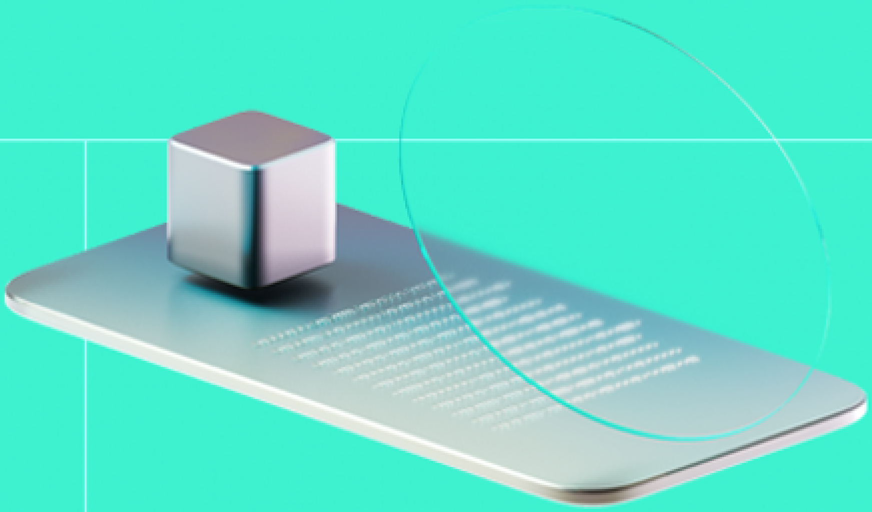
HACKEN

Smart Contract Code Review And Security Analysis Report



Customer: Grace Protocol

Date: 29.04.24

We express our gratitude to the Grace Protocol team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Grace is an L2-first cross-margin lending protocol that fairly distributes losses if they occur, making it resilient to oracle manipulation, volatility and exploits.

Platform: EVM

Language: Solidity

Tags: Lending, ERC20, Oracle

Timeline: 12/04/2024 - 25/04/2024

Methodology: https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/nourharidy/grace-protocol
Commit	bb62016

Audit Summary

10/10	9/10	63%	6/10
Security Score	Code quality score	Test coverage	Documentation quality score

Total 8/10

The system users should acknowledge all the risks summed up in the risks section of the report

5	3	2	0
Total Findings	Resolved	Accepted	Mitigated

Findings by severity

Critical	0
High	0
Medium	2
Low	2

Vulnerability	Status
F-2024-1736 - Chainlink's latestRoundData might return stale or incorrect results	Accepted
F-2024-1787 - Privileged roles should follow two step ownership transfer pattern	Accepted
F-2024-1629 - The usage of the precompile ecrecover can lead to signature mailability	Fixed
F-2024-1747 - Use of transfer() instead of call() to send native tokens	Fixed
F-2024-1751 - Missing call to moveDelegates in GTR burn	Fixed

Document

Name	Smart Contract Code Review and Security Analysis Report for Grace Protocol
Audited By	David Camps Novi, Viktor Lavrenenko
Approved By	Przemyslaw Swiatowiec
Website	https://app.grace.loans
Changelog	29/04/2024 - Preliminary Report ; 16/05/24 - Final Report

Table of Contents

System Overview

Grace is an L2-first cross-margin lending protocol that fairly distributes losses if they occur, with the following contracts:

BorrowController — Handles borrowing related logic and calculations.

Pool —Pools are deployed for each borrowable token via the core contract only. Pool contracts are the user-facing entry-points for users looking for debt-related operations such as lending and borrowing.

Collateral — Collaterals are deployed for each token used as collateral via the core contract only. Collateral contracts are the user-facing entry-points for users looking for collateral-related operations such as depositing or withdrawing collateral.

Oracle — The Oracle contract provides the core with prices of all collateral and pool tokens.

RateProvider — The rate provider manages the interest rate model contracts for each pool and fee rate model for each collateral.

Vault — Vaults allow Pool share token holders to deposit their shares into the vault in order to earn GTR rewards.

VaultFactory — The factory is used by the owner to deploy new vaults. It is also used by the owner in order to assign different reward weights for each vault.

Reserve —The reserve acts as the `feeRecipient` of the core contract. It receives all interest and collateral fees generated by the protocol.

Core — The core contract is a monolithic contract that contains hook functions.

GTR — The token used to pay for interests.

PoolDeployer — Deploys pool contracts.

CollateralDeployer — Deploys collateral contracts.

Privileged roles

- The `BorrowController` contract has the following roles:
  - The owner, which can:
    - set a new owner
    - set a guardian
    - forbid the contracts
    - choose the allowed contracts
    - suspend the borrowing for a specific pool
    - set daily borrow limit in USD
  - The guardian, which can:
    - pause the borrowing for a specific pool
- The `Core` contract has the following roles:
  - The owner, which can:

        - set a new owner
        - set the address of the `Oracle`
        - set the address of the `BorrowController`
        - set the address of the `RateProvider`
        - set the address of the `PoolDeployer`
        - set the address of the `CollateralDeployer`
        - set the address of the fee receiver
        - set liquidation incentive in bps
        - set the maximum liquidation incentive in USD
        - set the BadDebtCollateralThresholdUsd
        - set the WriteOff incentive Bps
        - deploy a new pool
        - set a pool deposit cap
        - deploy a new collateral
        - set a collateral factor
        - set a collateral cap in USD
        - pull tokens from the `Core` contract
        - pull tokens from the `Pool` contract
        - pull tokens from the `Collateral` contract
- The `GTR` contract has the following roles:
    - The operator, which can:
        - set minters
        - set a new operator
- The `Oracle` contract has the following roles:
    - The owner, which can:
        - set a new owner
        - set an address of the collateral feed
        - set an address of the pool feed
        - set the pool fixed price
        - set the bps per week
- The `Pool` contract has the following roles:
    - The core, which can pull ERC20 tokens from the pool
- The `RateProvider` contract has the following roles:
    - The owner, which can:
        - set a new owner
        - set a default interest rate model
        - set a default collateral fee model
        - set an interest rate model
        - set a collateral fee model
- The `Reserve` contract has the following roles:
    - The owner, which can:
        - set a new owner
        - request and execute allowance
- The `VaultFactory` contract has the following roles:
    - The operator, which can:
        - create new vaults
        - set weights
        - set a new operator
        - set a reward budget

Executive Summary

This report presents an in-depth analysis and scoring of the customer's smart contract project. Detailed scoring criteria can be referenced in the scoring methodology.

Documentation quality

The total Documentation Quality score is **6** out of **10**.

- Functional requirements are limited.
  - Missing roles description.
  - Functional requirements are superficial.
- The technical description is not complete.
  - Missing NatSpec.
  - Technical specifications are not provided.

Code quality

The total Code Quality score is **9** out of **10**.

- Some best practices are followed: F-2024-1753, F-2024-1745, F-2024-1762.
- The development environment is configured.

Test coverage

Code coverage of the project is **63%** (branch coverage).

- Deployment and basic user interactions are not covered with tests.
- Negative cases coverage is missed.
- Interactions by several users are not tested thoroughly.

Security score

Upon auditing, the code was found to contain **0** critical, **0** high, **2** medium, and **2** low severity issues, leading to a security score of **10** out of **10**.

All identified issues are detailed in the “Findings” section of this report.

Summary

The comprehensive audit of the customer's smart contract yields an overall score of **8.** This score reflects the combined evaluation of documentation, code quality, test coverage, and security aspects of the project.

Risks

- Iterating over a dynamic array populated with custom length can lead to gas limit denial of service if the number of elements goes out of control. This scenario is possible in such as `Core::writeOff`, where several loops take place.
- The ERC20 contract contains extra vesting logic. This may cause it to be non applicable to generic ERC20 accepting procedures.
- The potential for future account abstraction mechanisms in Ethereum could alter the behavior of `tx.origin`, possibly bypassing this check.
- Despite conditional usage, a malicious contract could still exploit `tx.origin` under certain circumstances, such as through carefully crafted call chains.
- The system benefits different uint types, conversions and calculations between these different types may lead to unexpected results due to limitations.
- Users should be aware that the fee the system charges can change over time.
- The `GTR` does not have a maximum supply, only limited to 2**96. Given this limit is reached, functions that require a token minting will revert.
- The system allows any token to be used. However, the development team stated that supported tokens will be pre-approved to ensure that they're compliant with ERC20, do not introduce any reentrancy risk and are non-rebasing.
- The project utilizes Solidity version 0.8.20 or higher, which includes the introduction of the PUSH0 (0x5f) opcode. This opcode is currently supported on the Ethereum mainnet but may not be universally supported across other blockchain networks. Consequently, deploying the contract on chains other than the Ethereum mainnet, such as certain Layer 2 (L2) chains or alternative networks, might lead to compatibility issues or execution errors due to the lack of support for the PUSH0 opcode. In scenarios where deployment on various chains is anticipated, selecting an appropriate Ethereum Virtual Machine (EVM) version that is widely supported across these networks is crucial to avoid potential operational disruptions or deployment failures.
- The function `Reserve::requestAllowance` allows to override the `allowanceRequest`, which may result in undesired behavior.
- The protocol interacts with external, non-trusted, out-of-scope contracts.
- The project's contracts don't implement the emergency stop pattern, which might lead to problems in the future.
- The Vault contract sets an unlimited allowance for the Pool to use its asset tokens. It creates risks and can lead to unexpected behavior.
- The owner of the `Core` contract can pull stuck ERC20 tokens from the `Core`, `Pool` and `Collateral` contracts, which creates a risk of highly permissive role access. It is recommended to use a MultiSignature wallet.
- In the case that an oracle stops providing reliable token prices, the system will be at risk for a particular amount of time, until a new oracle is setup. The team implemented several measures to mitigate the scenario, explained in the finding `F-2024-1736` that should be reviewed.
- Zero address validation is handled off-chain. However, it can lead to problems during the direct interaction with the protocol.

Findings

Vulnerability Details

F-2024-1629 - The usage of the precompile ecrecover can lead to signature mailability - Medium

Description: The functions `GTR::delegateBySig()`, `GTR::permit()` , `Collateral::permit()`, `Pool::permit()` and `Pool::permitBorrow()` are found to be vulnerable to a signature malleability issue. This vulnerability stems from the function's inability to discern between legitimately unique signatures and those that have been manipulated but are still considered valid by the Ethereum blockchain's signature verification standards. By exploiting this flaw, one of the users can create signatures that will be accepted by the system, enabling unauthorized and repetitive transactions. The vulnerable pieces of code can be found below:

- `GTR:delegateBySig()`

```
function delegateBySig(address delegatee, uint nonce, uint expiry, u
int8 v, bytes32 r, bytes32 s) public {
bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, kecc
ak256(bytes(name)), getChainId(), address(this)));
bytes32 structHash = keccak256(abi.encode(DELEGATION_TYPEHASH, deleg
atee, nonce, expiry));
bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSepara
tor, structHash));
address signatory = ecrecover(digest, v, r, s);
require(signatory != address(0), "GTR::delegateBySig: invalid signat
ure");
require(nonce == nonces[signatory]++, "GTR::delegateBySig: invalid n
once");
require(block.timestamp <= expiry, "GTR::delegateBySig: signature ex
pired");
return _delegate(signatory, delegatee);
}
```

- `GTR::permit()`

```
function permit(address owner, address spender, uint rawAmount, uint
deadline, uint8 v, bytes32 r, bytes32 s) external {
uint96 amount;
if (rawAmount == type(uint).max) {
amount = type(uint96).max;
} else {
amount = safe96(rawAmount, "GTR::permit: amount exceeds 96 bits");
}
bytes32 domainSeparator = keccak256(abi.encode(DOMAIN_TYPEHASH, kecc
ak256(bytes(name)), getChainId(), address(this)));
bytes32 structHash = keccak256(abi.encode(PERMIT_TYPEHASH, owner, sp
ender, rawAmount, nonces[owner]++, deadline));
bytes32 digest = keccak256(abi.encodePacked("\x19\x01", domainSepara
tor, structHash));
address signatory = ecrecover(digest, v, r, s);
require(signatory != address(0), "GTR::permit: invalid signature");
require(signatory == owner, "GTR::permit: unauthorized");
require(block.timestamp <= deadline, "GTR::permit: signature expired
");
allowances[owner][spender] = amount;
emit Approval(owner, spender, amount);
}
```

- `Collateral::permit()`

```
function permit(address owner, address spender, uint256 shares, uint
256 deadline, uint8 v, bytes32 r, bytes32 s) public {
require(deadline >= block.timestamp, "Collateral: EXPIRED");
bytes32 digest = keccak256(
abi.encodePacked(
"\x19\x01",
DOMAIN_SEPARATOR,
keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, shares, nonces
[owner]++, deadline))
)
);
address recoveredAddress = ecrecover(digest, v, r, s);
require(recoveredAddress != address(0) && recoveredAddress == owner,
"Collateral: INVALID_SIGNATURE");
allowance[owner][spender] = shares;
emit Approval(owner, spender, shares);
}
```

- `Pool::permit()`

```
function permit(address owner, address spender, uint value, uint dea
dline, uint8 v, bytes32 r, bytes32 s) external {
require(deadline >= block.timestamp, 'permitExpired');
bytes32 digest = keccak256(
abi.encodePacked(
'\x19\x01',
DOMAIN_SEPARATOR,
keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[
owner]++, deadline))
)
);
address recoveredAddress = ecrecover(digest, v, r, s);
require(recoveredAddress != address(0) && recoveredAddress == owner,
'Pool: INVALID_SIGNATURE');
allowance[owner][spender] = value;
emit Approval(owner, spender, value);
}
```

and

- `Pool::permitBorrow()`

```
function permitBorrow(address owner, address spender, uint value, ui
nt deadline, uint8 v, bytes32 r, bytes32 s) external {
require(deadline >= block.timestamp, 'permitExpired');
bytes32 digest = keccak256(
abi.encodePacked(
'\x19\x01',
DOMAIN_SEPARATOR,
keccak256(abi.encode(PERMIT_BORROW_TYPEHASH, owner, spender, value,
nonces[owner]++, deadline))
)
);
address recoveredAddress = ecrecover(digest, v, r, s);
require(recoveredAddress != address(0) && recoveredAddress == owner,
'Pool: INVALID_SIGNATURE');
borrowAllowance[owner][spender] = value;
emit BorrowApproval(owner, spender, value);
}
```

Assets:

- Collateral.sol

- GTR.sol
- Pool.sol

Status:	Fixed

Classification

Impact:	4/5
Likelihood:	3/5
Exploitability:	Independent
Complexity:	Complex

Likelihood [1-5]: 3

Impact [1-5]: 4

Exploitability [0-2]: 0

Complexity [0-2]: 1

Final Score: 3.3 (Medium)

Severity:	Medium

Recommendations

Remediation: To enhance the security of your Solidity smart contracts and mitigate the risk of signature malleability attacks, it is advisable to use OpenZeppelin's ECDSA library instead of the built-in `ecrecover` function. The ECDSA library provides robust and reliable signature verification, reducing the vulnerability to replay attacks and ensuring the integrity of the contract interactions.

Resolution: Fixed in the commit **bd1216b**: `ecrecover()` was replaced with `ECDSA.recover()`.

F-2024-1736 - Chainlink's latestRoundData might return stale or incorrect results - Medium

Description: The `getNormalizedPrice()` function calls out to a Chainlink oracle receiving the `latestRoundData()`. If there is a problem with Chainlink starting a new round and finding consensus on the new value for the oracle (e.g. Chainlink nodes abandon the oracle, chain congestion, vulnerability/attacks on the chainlink system) consumers of this contract may continue using outdated stale or incorrect data (if oracles are unable to submit no new round is started).

```
function getNormalizedPrice(address token, address feed) internal vi
ew returns (uint normalizedPrice) {
(,int256 signedPrice,,,) = IChainlinkFeed(feed).latestRoundData();
uint256 price = signedPrice < 0 ? 0 : uint256(signedPrice);
uint8 feedDecimals = IChainlinkFeed(feed).decimals();
uint8 tokenDecimals = IOracleERC20(token).decimals();
if(feedDecimals + tokenDecimals <= 36) {
uint8 decimals = 36 - feedDecimals - tokenDecimals;
normalizedPrice = price * (10 ** decimals);
} else {
uint8 decimals = feedDecimals + tokenDecimals - 36;
normalizedPrice = price / 10 ** decimals;
}
}
```

This function is being used in the system's oracle implementation to calculate high and low values for pools and collaterals. So any outdated results may directly affect the system's prices.

Assets:

- Oracle.sol

Status: Accepted

Classification

Impact: 5/5

Likelihood: 1/5

Exploitability: Semi-Dependent

Complexity: Simple

Likelihood [1-5]: 1
Impact [1-5]: 5
Exploitability [1-2]: 1
Complexity [0-2]: 0
Final Score: 0.7 (Low)

Severity: Medium

Recommendations

Remediation:

Add checks and timeout mechanisms to make sure the acquired result is the latest one. Given the following returned values from Chainlink's `latestRoundData()`:

```
(
/*uint80 roundID*/,
int price,
/*uint startedAt*/,
uint256 updatedAt,
/*uint80 answeredInRound*/
) = priceFeed.latestRoundData();
```

The following two checks should be introduced:

- To prevent the usage of old values, a new parameter (e.g. `heartbeat`) should be created as the desired minimum refresh date, and compared with the returned `updatedAt` time. You can reach out to the corresponding feed to get the refresh rate:

```
require(block.timestamp - updatedAt <= heartbeat, "Data is not fresh
");
```

- To prevent the use of an out-of-service oracle value, only positive values should be accepted::

```
require(price > 0, "Stale price");
```

Resolution:

The finding was accepted by the Grace team with the following information:

```
After careful consideration and after reading oracle contract implem
entations of many other protocols, we've decided the following:
- heartbeat will be ignored as it's less risky to use an outdated pr
ice than have no price reference at all. This is a practice followed
by Compound, Aave and Morpho who all completely ignore heartbeat val
ues.
- A 0 or negative collateral price will be returned as 0 and will no
t cause a revert. This allows liquidations and write-offs to occur w
ithout interruption
- A 0 or negative debt price will also be returned as 0. However, as
we already use the recorded high price for debt tokens, the existing
high price will continue to be used for some time and will slowly be
ramped down to 0.
```

Furthermore, the team implemented a couple of features to deal with external oracles and borrowers with debt with an invalid price feed in the commit `4135aa7`

```
- We now also handle reverts of external feed contracts and replace
them with 0 price. This prevents feed contracts from being used to a
ttack the protocol by DoSing liquidations, collateral withdrawals, w
rite-offs, etc.
- As we continue to risk underpricing debt tokens by the oracle in t
he case of an invalid feed, borrowers who have any debt with an inva
lid price feed are forbidden from borrowing any tokens or withdrawin
g any collateral until they repay all off the debt priced by an inva
```

```
lid feed. This is a feature enforced in Core.onPoolBorrow() and Core
.onCollateralWithdraw()
```

F-2024-1747 - Use of transfer() instead of call() to send native tokens - Low

Description:

The project utilizes the `transfer()` method for sending native currency to a recipient. However, if the recipient is a smart contract, the fixed gas limit associated with `transfer()` in Solidity might prevent the transaction from being completed.

In earlier Solidity versions, the `transfer()` function was favored for its straightforwardness and built-in reentrancy guard. Yet, it has become known for issues tied to its rigid gas ceiling of 2300.

Leveraging `transfer()` could inadvertently cause the transaction to revert if the recipient contract's `receive()` or `fallback()` methods require more than 2300 Gas for execution or if the gas price will be higher in the future.

The usage of the `transfer()` method was found in the following functions:

- Vault::withdrawETH()
- Pool::borrowETH()
- Pool::repayETH()
- Collateral::withdrawETH()
- Collateral::redeemETH()

Assets:

- Collateral.sol
- Pool.sol
- Vault.sol

Status: Fixed

Classification

Impact:	4/5
Likelihood:	1/5
Exploitability:	Independent
Complexity:	Simple

Likelihood [1-5]: 1

Impact [1-5]: 4

Exploitability [0-2]: 0

Complexity [0-2]: 0

	Final Score: 2.0 (Low)
Severity:	Low

Recommendations

Remediation:	It is recommended to use built-in `call()` function instead of `transfer()` to transfer native assets. This method does not impose a gas limit, it provides greater flexibility and compatibility with contracts having more complex business logic upon receiving the native tokens. When working with then `call()` function ensure that its execution is successful by checking the returned boolean value. It is also recommended to fallow the Check-Effects-Interactions (CEI) pattern in every case to prevent reentrancy issues.
Resolution:	The finding was fixed in commit **bd1216b**: `transfer` calls were updated to `call` with return value checked.

F-2024-1787 - Privileged roles should follow two step ownership transfer pattern - Low

Description:	The privileged roles: `owner` and `operator` carry numerous important abilities for the system. However, the following functions allow the addresses of these roles to be errantly transferred to the wrong addresses as they do not use a two-step transfer process:

- GRT::setOperator()
- VaultFactory::setOperator()
- BorrowController::setOwner()
- Core::setOwner()
- Oracle::setOwner()
- RateProvider::setOwner()
- Reserve::setOwner()

Assets:

- GTR.sol

Status: Accepted

Classification

Impact:	3/5
Likelihood:	2/5
Exploitability:	Independent
Complexity:	Simple

Likelihood [1-5]: 2

Impact [1-5]: 3

Exploitability [0-2]: 0

Complexity [0-2]: 0

Final Score: 2.5 (Low)

Severity: Low

Recommendations

Remediation: It is recommended to implement a two-step “push” and “pull” ownership transfer process for privileged roles.

Resolution: The finding was accepted by the Grace team with the following information:

> The recommended push-pull ownership transfer pattern is incompatible with potential future on-chain governance contracts which would require a governance vote to be passed in order to approve an ownership transfer.

F-2024-1751 - Missing call to moveDelegates in GTR burn - Info

Description: When GTR tokens are burnt via `burn()`, the function `_moveDelegates()` should be called in order to update the delegates.

The `burn()` function implementation:

```
/**
* @notice Burn caller's tokens
* @param rawAmount The number of tokens to be burned from the caller
's balance
*/
function burn(uint rawAmount) external returns (bool) {
uint96 amount = safe96(rawAmount, "GTR::mint: amount exceeds 96 bits
");
balances[msg.sender] = sub96(balances[msg.sender], amount, "GTR::bur
n: burn amount exceeds balance");
totalSupply = safe96(totalSupply - amount, "GTR::burn: burn amount e
xceeds totalSupply");
emit Transfer(msg.sender, address(0), amount);
return true;
}
```

Assets:

- GTR.sol

Status: Fixed

Classification

Impact: 2/5

Likelihood: 4/5

Exploitability: Semi-Dependent

Complexity: Medium

Likelihood [1-5]: 4

Impact [1-5]: 2

Exploitability [0-2]: 1

Complexity [0-2]: 1

Final Score: 2.3 (Low)

Severity: Info

Recommendations

Remediation: Perform a call to `_moveDelegates` when tokens are burnt.

Resolution: The finding was fixed in commit **bd1216b**: a call to `_moveDelegates` was introduced in token burn.

Observation Details

F-2024-1741 - Checks-effects-interactions pattern violation - Info

Description: When depositing funds, the checks-effects-interactions pattern behaves differently. In such cases, the token transfers should be performed before updating the balances. Therefore, the following functions do not comply with the best practices for this pattern implementation.

```
function deposit(uint256 assets, address recipient) public lock retu
rns (uint256 shares) {
uint _lastAccrued = accrueFee();
require(core.onCollateralDeposit(recipient, assets), "beforeCollater
alDeposit");
require((shares = previewDeposit(assets)) != 0, "zeroShares");
balanceOf[recipient] += shares;
totalSupply += shares;
addToDepositors(recipient);
asset.safeTransferFrom(msg.sender, address(this), assets);
lastBalance = asset.balanceOf(address(this));
require(lastBalance >= MINIMUM_BALANCE, "minimumBalance");
emit Deposit(msg.sender, recipient, assets, shares);
updateFee(_lastAccrued);
}
```

The affected functions are:
`src/Collateral.sol:: deposit, depositETH, mint.`
`src/Pool.sol:: deposit, mint, repay, repayETH.`
`src/Vault.sol:: depositShares.`

Assets:

- Collateral.sol

Status: Fixed

Recommendations

Remediation: It is recommended to update the balances after performing the token transfer.

Resolution: The finding was fixed in commit **bd1216b**: a `nonReentrant` modifier was added into the `Vault` contract's reported functions. The reported functions from `Collateral` and `Pool` contracts are covered with the `lock` nonReentrant modifier.

F-2024-1745 - Missing zero address validation - Info

Description:

In Solidity, the Ethereum address `0x0000000000000000000000000000000000000000` is known as the "**zero address**". This address has significance because it is the default value for uninitialized address variables and is often used to represent an invalid or non-existent address.

The "**Missing zero address Validation**" issue arises when a Solidity smart contract does not properly check or prevent interactions with the zero address, leading to unintended behavior.

For instance, consider a contract that includes a function to change its owner. This function is crucial, as it determines who has administrative access. However, if this function lacks proper validation checks, it might inadvertently permit the setting of the owner to the zero address. Consequently, the administrative functions will become unusable.

Missing checks were observed in the following functions:

- GTR::constructor() → _operator
- GTR::setMinter() → minter_
- GTR::setOperator() → operator_
- Vault::constructor() → _pool, _gtr
- Vault::depositShares() → recipient
- Vault::depositAsset() → recipient
- Vault::depositETH() → recipient
- Vault::withdrawETH() → recipient
- Vault::withdrawShares() → recipient
- Vault::withdrawAsset() → recipient
- Vault::approve()
- VaultFactory::constructor() → _gtr, _weth
- Pool::constructor() → _asset, _core
- Pool::deposit() → recipient
- Pool::transfer() → recipient
- Pool::approve() → spender
- Pool::transferFrom() → recipient
- Collateral::constructor() → _asset, _core
- Collateral::deposit() → recipient
- Collateral::depositETH() → recipient
- Collateral::mint() → recipient
- Collateral::withdraw() → receiver
- Collateral::withdrawETH() → receiver
- Collateral::redeem() → receiver
- Collateral::redeemETH() → owner
- Collateral::seize() → to
- Collateral::pull() →dst
- BorrowController::setOwner() → _owner

- BorrowController::setGuardian() → _guardian
- BorrowController::setContractAllowed() → contractAddress
- RateProvider::setOwner() → _owner
- RateProvider::setDefaultInterestRateModel() → _defaultInterestRateModel
- RateProvider::setDefaultCollateralFeeModel() → _defaultCollateralFeeModel
- RateProvider::setInterestRateModel() → pool
- RateProvider::setCollateralFeeModel() → collateral
- Reserve::constructor() → _gtr
- Reserve::setOwner() → _owner

Assets:

- BorrowController.sol
- Collateral.sol
- GTR.sol
- Pool.sol
- RateProvider.sol
- Reserve.sol
- Vault.sol
- VaultFactory.sol

Status: Accepted

Recommendations

Remediation: Implement zero address checks for the aforementioned functions.

Resolution: `Accepted` with the following explanation:

> We believe that this kind of address input validation should be handled on the wallet level instead. Therefore, we won't be validating zero address inputs on the contract level.

F-2024-1753 - The EIP712 domain separator is missing the version field - Info

Description: The domain separator in `GTR.sol` is missing the `version` field defined in EIP-712. The standard states that not all fields are mandatory but adding them would add another layer of security for the usage of off-chain signed messages for the protocol. The affected code can be found below:

```
bytes32 public constant DOMAIN_TYPEHASH = keccak256("EIP712Domain(st
ring name,uint256 chainId,address verifyingContract)");
```

Assets:

- GTR.sol

Status: Accepted

Recommendations

Remediation: It is recommended to refer to the EIP712 doc and add all of the missing domain separator fields.

Resolution: The finding was `Accepted` by the Grace Protocol team:

> The GTR contract address and chainId are already included and the contract is not upgradable.

F-2024-1755 - Cache array length in loops to save gas - Info

Description:

Several loops calculate the length of storage arrays instead of caching such value in a memory value. As a consequence, the whole array will be accessed as a storage reading for each iteration, saving a big amount of gas unnecessarily.

```
function onPoolRepay(address recipient, uint256 amount) external ret
urns (bool) {
...
if(amount == debt) {
for (uint i = 0; i < borrowerPools[recipient].length; i++) {
...
}
```

Additionally, this extra amount of gas will contribute to the likelihood of reaching the block gas limit for such operations, as reported in the `Risk` section of this report.

Affected functions are:
`src/Core.sol: onCollateralWithdraw, onPoolBorrow, onPoolRepay, liquidate, writeOff.`

Assets:

- Core.sol

Status: Fixed

Recommendations

Remediation: It is recommended to cache the length of storage arrays during loops in order to save gas.

Resolution: The finding was fixed in the commit **bd1216b**: the recommended remediation was implemented.

F-2024-1762 - Missing events emitting for critical data - Info

Description:

Events for critical state changes should be emitted for tracking actions off-chain.

Events are crucial for tracking changes on the blockchain, especially for actions that alter significant contract states or permissions. The absence of events in these functions means that external entities, such as user interfaces or off-chain monitoring systems, cannot effectively track these important changes.

It was observed that events are missing events in the following functions:

- BorrowController::setOwner()
- BorrowController::setGuardian()
- BorrowController::setPoolBorrowPaused()
- BorrowController::setForbidContracts()
- BorrowController::setContractAllowed()
- BorrowController::setPoolBorrowSuspended()
- BorrowController::setDailyBorrowLimitUsd()
- Collateral::pull()
- Core::setOwner()
- Core::setOracle()
- Core::setBorrowController()
- Core::setRateProvider()
- Core::setPoolDeployer()
- Core::setCollateralDeployer()
- Core::setFeeDestination()
- Core::setLiquidationIncentiveBps()
- Core::setMaxLiquidationIncentiveUsd()
- Core::setBadDebtCollateralThresholdUsd()
- Core::setWriteOffIncentiveBps()
- Core::setPoolDepositCap()
- Core::setCollateralFactor()
- Core::setCollateralCapUsd()
- Core::pullFromCore()
- Core::pullFromPool()
- Core::pullFromCollateral()
- Oracle::setOwner()
- Oracle::setCollateralFeed()
- Oracle::setPoolFeed()
- Oracle::setPoolFixedPrice()
- Oracle::setBpsPerWeek()
- Pool::pull()
- RateProvider::setOwner()
- RateProvider::setDefaultInterestRateModel()
- RateProvider::setDefaultCollateralFeeModel()
- RateProvider::setInterestRateModel()

- RateProvider::setCollateralFeeModel()
- Reserve::setOwner()
- Vault::depositShares()
- Vault::depositAsset()
- Vault::depositETH()
- Vault::withdrawETH()
- Vault::withdrawShares()
- Vault::withdrawAsset()
- VaultFactory::setOperator()
- VaultFactory::setBudget()

Assets:

- BorrowController.sol
- Collateral.sol
- Core.sol
- Oracle.sol
- Pool.sol
- RateProvider.sol
- Reserve.sol
- Vault.sol
- VaultFactory.sol

Status: Accepted

Recommendations

Remediation: Consider adding events emitting to the aforementioned functions.

Resolution: The finding was `Accepted` given the following explanation in commit **bd1216b**:

> We've only added new events for user-facing functions in the `Vault` and `VaultFactory`. As for the `onlyOwner` functions, we chose to reduce code bloat by not emitting events for these functions as off-chain listeners can watch the owner address for new transactions instead of watching for contract events.

F-2024-1779 - Lack of consistency in assets redeemed amount - Info

Description:

Both `redeem` functions in `Pool` and `Collateral` contracts calculate `assets = previewRedeem(shares)`. However, only in the `Pool` contract the following check is added `require((assets = previewRedeem(shares)) != 0, "zeroAssets")`.

This raises a concern of consistency regarding the amount of `shares` and `assets` calculated in the aforementioned contracts, behaving differently for cases in which the result should be rounded down.

Assets:

- Collateral.sol
- Pool.sol

Status: Fixed

Recommendations

Remediation:

It is recommended to maintain a consistency, applying `require((assets = previewRedeem(shares)) != 0, "zeroAssets")` in all cases necessary when rounding down.

Resolution:

The observation was fixed in the commit **bd1216b** by applying the recommended check and updating the rounding direction.

F-2024-1784 - Missing feed address check may result in undesired price calculations - Info

Description: The functions `getCollateralPriceMantissa`, `getDebtPriceMantissa` and `viewCollateralPriceMantissa` return price `0` when no data feed is setup.

However, these functions play an important role in calculating the prices of assets in different parts of the systems, and returning a `0` value will be problematic, in case the feed is not previously set.

```
function getDebtPriceMantissa(address token) external returns (uint2
56) {
if(poolFixedPrices[token] > 0) return poolFixedPrices[token];
address feed = poolFeeds[token];
if(feed != address(0)) {
uint high = getPoolHigh(msg.sender, token);
if(high != poolHighs[msg.sender][token].price) {
poolHighs[msg.sender][token] = PriceLog(high, block.timestamp);
emit RecordPoolHigh(msg.sender, token, high);
}
return high;
}
return 0;
}
```

Assets:

- Oracle.sol

Status: Fixed

Recommendations

Remediation: Consider adding a safety mechanism to protect from the case an oracle is not setup.

Resolution: The issue was fixed in commit `c08af7c`: invalid oracle feeds cannot be passed in `Oracle::setCollateralFeed()` or `Oracle::setPoolFeed()`. Additional safe checks are implemented when interacting with the protocol to make sure no token is used without a proper feed address setup.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Severity Definitions

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution, do not affect security score but can affect code quality score.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details

Repository	https://github.com/nourharidy/grace-protocol
Commit	bb62016
Remediation commit	d961752
Whitepaper	n/a
Requirements	https://docs.grace.loans
Technical Requirements	https://github.com/nourharidy/grace-protocol

Contracts in Scope

BorrowController.sol

CollateralDeployer.sol

Core.sol

GTR.sol

Oracle.sol

Pool.sol

PoolDeployer.sol

RateModel.sol

RateProvider.sol

Reserve.sol

Vault.sol

VaultFactory.sol

Collateral.sol