Interest Protocol

Interest Protocol Smart Contracts

Security Assessment Report

Version: 2.1

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Interest Protocol smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Interest Protocol smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Interest Protocol smart contracts.

Overview

Interest Protocol is a decentralized banking protocol on the Ethereum blockchain that applies fractional reserve banking to decentralized finance.

Interest Protocol issues a stablecoin called USDi, which is a liquidity provider (LP) token that represents a one-to-one claim on the protocol's USDC.

USDi can be minted by either depositing USDC into the protocol or borrowing USDi from the protocol. Interest Protocol generates revenue from interest paid by borrowers, and this revenue is distributed to all USDi holders.

Interest Protocol's governance is controlled by a decentralized autonomous organization, called Interest Protocol DAO (IP DAO). IP DAO manages the smart contracts associated with the protocol by updating parameters or upgrading contracts.

Interest Protocol Token (IPT) is the governance token of the protocol. When the DAO votes on a proposal, one IPT represents one vote. IPT is distributed to the community through a public sale, liquidity mining programs, and other methods.

Security Assessment Summary

This review was conducted on the files hosted on the Interest Protocol's repository and were assessed at commit 7a2d613.

Scope of testing included all files under `contracts` directory, except the following:

- `IPTsale/MerkleRedeem/`
- `governance/token/`
- `lending/CappedFeeOnTransferToken.sol`
- `lending/CappedRebaseToken.sol`
- `lending/CappedToken.sol`
- `lending/CappedSTETH.sol`
- `testing/`
- `upgrade/`
- `_external/`

Note: the OpenZeppelin libraries and dependencies were also excluded from the scope of this assessment.

The manual code review section of the report is focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. This includes their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [1, 2].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 22 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- Medium: 7 issues.
- Low: 8 issues.
- Informational: 6 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Interest Protocol smart contracts. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as “informational”.

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
INT-01	`updateRegisteredErc20()` modifies the wrong token	Critical	Resolved
INT-02	Interest accrual whilst the `VaultController` is paused	Medium	Closed
INT-03	`safeTransfer()` not used with transfer functions	Medium	Resolved
INT-04	Denial of Service (DoS) on multi-transaction proposals	Medium	Resolved
INT-05	`get_vault_borrowing_power()` could prevent vault's functionality	Medium	Closed
INT-06	`initialize()` functions are front-runnable	Medium	Closed
INT-07	Function `getPoints()` allows overbuying	Medium	Closed
INT-08	Incorrect sender in `Transfer` events	Medium	Resolved
INT-09	Proposal censorship by delegators	Low	Closed
INT-10	Unsafe casting in `initialize()` function of `TokenDelegate.sol`	Low	Resolved
INT-11	Calls to `updateRegisteredErc20()` could render vaults instantly insolvent	Low	Closed
INT-12	Contracts holding USDi may not be adapted to the gradually changing balances	Low	Closed
INT-13	Whitelisting can expire mid way through a proposal	Low	Closed
INT-14	Inaccurate `votingPeriod` and `votingDelay` time measure	Low	Closed
INT-15	Hardcoded voting and proposal thresholds	Low	Closed
INT-16	Insufficient checks for zero values and addresses	Low	Closed
INT-17	Malicious modification of `_usdi` or `_oracleMaster` would allow the removal of all user funds	Informational	Closed
INT-18	Recovery of lost USDi	Informational	Closed
INT-19	Direct usage of `ecrecover()` allows signature malleability	Informational	Resolved
INT-20	Sudden price drop renders vaults insolvent	Informational	Closed
INT-21	Gas inefficient implementation of a loop in `get_vault_borrowing_power()`	Informational	Closed
INT-22	Miscellaneous General Comments	Informational	Resolved

INT-01	`updateRegisteredErc20()` modifies the wrong token		
Asset	`VaultController.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

`updateRegisteredErc20()` updates incorrect collateral token's oracle address and LTV.

Line [**264**] and line [**266**] of `updateRegisteredErc20()` both update variables based on the token ID. However, these lines use the variable `_tokensRegistered` as the token ID, which is the token ID of the last token added, not necessarily the ID of the token being updated. This is likely code copied from the previous function, `registerErc20()`, where the ID of the token being updated is always the most recent token added, as that is the function which actually adds it.

```
function updateRegisteredErc20(
    address token_address,
    uint256 LTV,
    address oracle_address,
    uint256 liquidationIncentive
  ) external override onlyOwner {

    // (...)

    // SigP: Note _tokensRegistered is used below, instead of token_address passed in as a parameter to this function.

    // set the oracle of the token
    _tokenId_oracleAddress[_tokensRegistered] = oracle_address;
    // set the ltv of the token
    _tokenId_tokenLTV[_tokensRegistered] = LTV;
```

If `updateRegisteredErc20()` were called with the intention of updating one token but instead having the effect of updating another token's LTV, the results could significantly damage the protocol. Many vaults that were previously solvent could be immediately rendered insolvent and be liquidated.

Similarly, installing the oracle address for the wrong token could result in a thousand dollar token suddenly being valued at a couple of dollars. Vaults borrowing against that token would then go insolvent, be fully liquidated, and the token would be transferred to the liquidator for a thousandth of its value.

Recommendations

Modify `updateRegisteredErc20()` to retrieve a token's ID based on the parameter `token_address`. Then use this token ID in place of `_tokensRegistered`.

Resolution

✓ Resolved in commit [b447c19]

INT-02	Interest accrual whilst the `VaultController` is paused		
Asset	`VaultController.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

When `VaultController` is paused, users are unable to repay borrowed funds, but the interest is still accrued, potentially forcing users into liquidation.

`VaultController` is a pausable contract and amongst its functions which are inaccessible when paused are `repayUSDi()` and `repayAllUSDi()`, both of which bear the `whenNotPaused` modifier. However, `calculateInterest()` and `pay_interest()` are both still callable when the contract is paused, with interest accruing as a function of time (and the reserve ratio).

As a result of this, it is possible that a user could mint a vault, deposit into it, borrow USDi against it and then accrue interest, potentially putting the vault on the edge of insolvency.

Consider a situation where, just before this user is about to call `repayUSDi()`, the protocol is paused (e.g. in a situation where the reserve ratio is very low and so the interest rate is very high). In this situation, the user is locked out from repaying USDi into their vault, but interest keeps accruing over time, eventually forcing the user into insolvency. They are then liquidated as soon as the protocol is unpaused and lose their assets.

Recommendations

Modify `pay_interest()` so that it does not accrue any interest whilst the protocol is paused. In doing so, it is recommended not to make an interest calculation a precursor to pausing, as any issue with interest calculation could potentially render the contract unpausable.

Resolution

This risk was accepted by the project team. No mitigations have been implemented.

INT-03	`safeTransfer()` not used with transfer functions		
Asset	`CappedFeeOnTransferToken.sol`, `CappedGovToken.sol`, `Slowroll.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

`CappedFeeOnTransferToken`, `CappedGovToken` and `SlowRoll` do not use `safeTransfer()` functions:

- `CappedFeeOnTransferToken` calls `_underlying.transferFrom()` on line [96].
- `CappedGovToken` calls `_underlying.transferFrom()` on line [84].
- `SlowRoll` calls `takeFrom()`, which in turn calls `_pointsToken.transferFrom()` on line [149].

This may cause reverted transactions if the underlying token does not properly implement the ERC20 standard.

Note that the token best known for not complying to this standard is USDT, which does have code capable of implementing a fee on transfer facility, and so could conceivably be contained in one of these token wrappers.

Recommendations

Use `safeTransferFrom()` from `SafeERC20Upgradeable` in place of `transferFrom()` in the locations mentioned above.

Resolution

✓ Resolved in commit [b447c19]

In the case of `SlowRoll`, the development team are aware of the issue and do not intend to use any affected tokens.

INT-04	Denial of Service (DoS) on multi-transaction proposals		
Asset	`GovernorDelegate.sol`		
Status	**Resolved:** Addressed at commit b447c190. See Resolution for more information.		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Anyone can execute an individual transaction from a multi-transaction proposal via `external` function `executeTransaction()`, rendering the overall proposal unusable.

A proposal can consist of multiple transactions (or "calls") to be executed. Once the proposal successfully passes through and is queued for execution, anyone can execute it through calling `execute()` function, which will go in sequence through every individual transaction in the proposal and call `executeTransaction()` on it.

As the `executeTransaction()` function is `external`, it is also possible to call it directly to execute any individual transaction separately.

One of the first checks performed in `executeTransaction()` is if the transaction has been queued for execution:

```
require(queuedTransactions[txHash], "tx hasn't been queued.");
```

And the transaction is then removed after successful execution:

```
queuedTransactions[txHash] = false;
```

If a proposal with multiple transactions is submitted, an attacker could call `executeTransaction()` to execute a single transaction from the proposal. Subsequent `execute()` calls on the proposal will fail and revert, as the previously executed transaction will fail the queued check discussed above, rendering the entire proposal unusable and 'stuck'.

Note, in such case, it would still be possible to manually call `executeTransaction()` on every other remaining transaction in the affected proposal to execute it fully. However, it would only be feasible if the proposal does not rely on a specific order of transaction execution.

Recommendations

Change visibility of `executeTransaction()` to `internal` to prevent it from being called directly by any external party.

Resolution

The development team has resolved this issue by performing an additional check to ensure the `executeTransaction()` can only be called by the `GovernorCharlieDelegate` contract itself:

```
require(msg.sender == address(this), "execute must come from this address");
```

INT-05	`get_vault_borrowing_power()` could prevent vault's functionality		
Asset	`VaultController.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

External calls from `get_vault_borowing_power()` could revert, disabiling ability of borrowing against a vault or liquidating a vault.

`get_vault_borrowing_power()` contains a loop, starting on line [**580**], which loops through all of the registered tokens and queries the vault for its balance through a call to `vault.tokenBalance()` on line [**589**] which in turn makes the external call `IERC20(addr).balanceOf(address(this))`. It also contains external calls to external oracles for each registered token through `_oracleMaster.getLivePrice()` on line [**594**].

There is a chance, with all of these external calls, that one of them might revert. In that case, the entire transaction would revert.

Some potential revert scenarios might be:

- If the anchor oracle and the main oracle are reporting prices too far apart.
- If an oracle is experiencing a problem with its data source and reverts to prevent providing potentially bad data.
- If a token has a complex balance calculation system, akin to uFragments, but containing an error which causes some balance queries to revert.

Consider a situation where an arbitrary token, let's call it `RVRT`, is added to the protocol. After a few weeks of normal operation, it experiences the third issue listed above and reverts on all calls to `RVRT.balanceOf()`. Because this function is called inside `get_vault_borrowing_power()` for all vaults, even those not containing `RVRT`, all calls to borrow against a vault, or to liquidate a vault would also revert.

In the event that a vault cannot be liquidated and the asset it contains dropping heavily, it would be possible for it to become heavily insolvent. If this happened to many vaults across the entire protocol, then the entire protocol would become insolvent: the value of the USDC reserve and the assets in vaults would be significantly lower than the total USDi in circulation.

Recommendations

This issue can be partially mitigated by carefully managing and curating the list of registered tokens and oracles. However, this does not completely mitigate the issue.

To fully address this issue, more significant changes may be required. One approach would be to make all vaults single asset. That way, there is no need for the loop within `get_vault_borrowing_power()`. Another approach might be to create a copy of `get_vault_borrowing_power()` for use in transactions, and have that wrap its external calls in `try` blocks.

Resolution

This risk was accepted by the project team. No mitigations have been implemented.

INT-06	`initialize()` functions are front-runnable		
Asset	`VaultController.sol`, `USDI.sol`, `TokenDelegate.sol`, `GovernorDelegate.sol`		
Status	**Closed:** Risk accepted by the project team. See Resolution for more information.		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

There are no access control checks on `initialize()` functions used to configure the protocol during the deployment.

An attacker could front-run the deployment process and call `initialize()` functions to set their own parameters, e.g. set arbitrary implementation and token addresses or modify intended ownerships.

The affected contracts include (note, the list below should not be considered exhaustive):

- `VaultController.sol`
- `USDI.sol`
- `TokenDelegate.sol`
- `GovernorDelegate.sol`

Recommendations

Implement logic in the constructors to save address of the owner/deployer of the contract and verify it in `initialize()` functions, e.g.

```
require(msg.sender == owner);
_;
```

Resolution

The development team has risk accepted this finding and no mitigations have been implemented.

INT-07	Function `getPoints()` allows overbuying		
Asset	`slowroll.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

The `getPoints()` function does not ensure the amount of `USDC` taken from the users is equal to the value of `IPT` given from the contract.

This can occur if the balance of `IPT` in the `SlowRoll` contract is less than `_maxQuantity` at the final wave.

It is very likely that users will not buy out all the tokens at each wave, making it possible to have a wave where the value of the balance of `IPT` in the `SlowRoll` contract is less than the amount of `USDC` taken from the users.

In this situation, it is also possible to end up overbuying, even when purchasing the exact amount of `IPT` balance, when another user front-runs a transaction buying smaller amount of `IPT` with the sum not exceeding `_maxQuantity`.

An example of this scenario is described below:

1. A new wave starts with 500,000 `IPT` in balance, `_maxQuantity` is set to 1,000,000 `IPT`.
2. Alice pays for 500,000 `IPT`.
3. Bob front-runs a transaction paying for 499,999 `IPT`.
4. There is only 1 `IPT` remaining but Alice ends up paying for 500,000 `IPT`.

Recommendations

Implement checks in the `takeFrom()` function to ensure it is not allowed to transfer more `USDC` than the value of `IPT` available in the contract.

This could be achieved by validating that the `rewardAmount` variable never exceeds the balance of `IPT` in the `SlowRoll` contract.

Resolution

The development team are aware of the issue but have decided not to implement a fix.

INT-08	Incorrect sender in `Transfer` events		
Asset	`USDI.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Low	Likelihood: High

Description

The `Transfer` events in functions `withdrawTo()` and `withdrawAllTo()` have an incorrect `from` value when `target` is not the same as the message sender.

When a withdrawal is performed by any of these functions, USDC are sent from the protocol to the `target` and USDi are sent from the message sender to the protocol.

Recommendations

Ensure that `Transfer` events from functions `_withdraw()` and `_withdrawAll()` specify `_msgSender()` as the sender.

Be aware that emitted `Transfer` events prior to the fix of this issue specify an incorrect sender and shouldn't be used as correct data.

Resolution

The recommendation has been implemented in commit b447c19.

INT-09	Proposal censorship by delegators		
Asset	`GovernorDelegate.sol`		
Status	**Closed:** Risk accepted by the project team. See Resolution for more information.		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Approved proposals scheduled for execution can be cancelled at any time by vote delegators.

Normally, proposals can only be cancelled by the original proposer, or by anyone when proposer votes drop below proposal threshold.

If a proposer (a delegatee) had all of their votes delegated by someone else (a delegator), at any time, delegator may 'un-delegate' their votes (e.g. by simply assigning them to someone else), resulting in the proposer's votes dropping below required proposal threshold. Delegator may then call `cancel()` on the proposal and successfully terminate it, even if it has already successfully passed governance vote.

Whilst it may be an intended functionality, it appears that delegatees do not have the same absolute power as IPT token holders as their privileges can be taken away at any time, and effectively have their proposals censored, even if they were already accepted by the governance and scheduled for execution.

Recommendations

Consider implementing logic to disallow cancellation of proposals after reaching a specific state (e.g. queued and surpassed a timelock).

Resolution

The development team has advised:

This is intended behavior, flipsiding someone is the right of the delegator.

INT-10	Unsafe casting in `initialize()` function of `TokenDelegate.sol`		
Asset	`TokenDelegate.sol`		
Status	**Resolved:** Addressed at commit b447c190. See Resolution for more information.		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Unsafe casting may lead to a loss of significant portion of the initial supply of IPT tokens.

Consider the code snippet below from `contracts/governance/token/TokenDelegate.sol`. If a value provided in `initialSupply_` parameter is larger than `uint96` maximum, the result will be 'wrapped around', producing a significantly smaller value.

Subsequently, balance of the `account_` address intended to receive all of the initial suppply will be a lot lower than an intended value.

```
function initialize(address account_, uint256 initialSupply_) public override {
  require(totalSupply == 0, "initialize: can only do once");
  require(account_ != address(0), "initialize: invalid address");
  require(initialSupply_ > 0, "invalid initial supply");

  totalSupply = initialSupply_;

  balances[account_] = uint96(totalSupply);
  emit Transfer(address(0), account_, totalSupply);
}
```

Recommendations

Implement bounds checks on `initialSupply_` parameter to ensure it is within an expected range.

Resolution

The development team has resolved this issue by performing an additional check to ensure the `initialSupply_` parameter is within `uint96` boundries:

```
require(initialSupply_ < 2**96, "initialSupply_ overflow uint96");
```

INT-11	Calls to `updateRegisteredErc20()` could render vaults instantly insolvent		
Asset	`VaultController.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

`updateRegisteredErc20()` has the ability to modify the LTV ratio of a token across the protocol to any arbitrary, potentially dangerously low value.

If this value is lowered so that the allowed loan amount for a given deposit value drops, then, in a single transaction, any vault that has borrowed an amount that is within the old LTV but above the new LTV would become instantly insolvent. This would then cause them to be liquidated, damaging the protocol.

This is mitigated by the presumed procedure of such a change, which would work through governance and so there would be time for vault owners to react during the proposal and the timelock implementation period.

Recommendations

Be aware of this issue and inform the governance community of the dangers of proposals to lower LTV.

Resolution

The development team has acknowledged the issue and stated their intention to advise governance to manage the protocol with consideration for the points raised above.

INT-12	Contracts holding USDi may not be adapted to the gradually changing balances		
Asset	`USDI.sol`, `UFragments.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Gradually increasing balance of USDi token may cause unexpected issues when used with Uniswap style liquidity pools or other staking smart contracts.

The USDi token automatically pays interest to its holders through a gradually increasing balance. This represents the redistributed interest payments made those borrowing USDi in the protocol's vaults.

The development team expressed a desire to avoid wrapping the USDi token and use it in its current form with liquidity pools and other smart contracts, which may result in an unexpected behaviour.

In the case of Uniswap style liquidity pools, the pool smart contract tracks its own expected balance of the two tokens in the pool. Any excess balance can be removed with the `skim()` function. If USDi is held in one of these pools, its balance would slowly increase over time and any user, even one not connected with the protocol, could call `skim()` and receive the generated interest USDi tokens.

Other staking smart contracts might make calculations based on the token balances they hold, or even assess the value of USDi on the basis of their balance.

Recommendations

Consider deploying the `WUSDi` contract and using it for liquidity pools and other smart contract interactions ouside of Interest Protocol.

Resolution

The development team has acknowledged the issue and pointed out that they do not control the usage of USDi outside of their own systems, but have made the `WUSDi` contract publicly available for any who wish to use it.

INT-13	Whitelisting can expire mid way through a proposal		
Asset	`GovernorDelegate.sol`		
Status	**Closed:** Risk accepted by the project team. See Resolution for more information.		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Whitelisted proposals can expire during the assessment period, making a user, and a proposal, lose its status and benefits.

Different procedures and privileges are enacted for proposers based on whether they are whitelisted or not. The assessment of whether an account is whitelisted is based on the test performed in `isWhitelisted()`, where `whitelistAccountExpirations[account]` is compared to the current time.

Based on a comment on line [**256**], a proposal by a whitelisted proposer should not be cancelled for falling below proposal threshold.

Consider a situation in which a whitelisted proposer makes a proposal and then their whitelisting expires during the assessment period. The entire proposal would then lose its whitelisted status, notably allowing cancellation should the proposer drop below proposal threshold.

Recommendations

When a proposal is created by a whitelisted proposer, consider marking it as whitelisted, and then assess the proposals status according to this value, rather than continuously assessing the status of the account that proposed it.

Resolution

The development team has risk accepted this finding and no mitigations have been implemented.

INT-14	Inaccurate `votingPeriod` and `votingDelay` time measure		
Asset	`GovernorDelegate.sol`		
Status	**Closed:** Risk accepted by the project team. See Resolution for more information.		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

As `votingPeriod` and `votingDelay` times are measured in blocks, they may produce inconsistent results.

```
votingPeriod = 40320;
votingDelay = 13140;

// (...)

Proposal memory newProposal = Proposal({
  // (...)
  startBlock: block.number + votingDelay,
  endBlock: block.number + votingDelay + votingPeriod,
  // (...)
```

With Ethereum's move to proof-of-stake (PoS) after The Merge, `block.number` is now considered an inaccurate method of measuring time as the block times are no longer consistent.

Recommendations

Use `block.timestamp` for time measurement to ensure best accuracy and consistency.

Resolution

The development team has advised:

We understand that they are inaccurate - the idea is that you have one parameter which is blocks, and one which is time. They are only intended to be approximately accurate.

INT-15	Hardcoded voting and proposal thresholds		
Asset	`GovernorDelegate.sol`		
Status	**Closed:** Risk accepted by the project team. See Resolution for more information.		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Voting and proposal thresholds are hardcoded in the `initialize()` function of `GovernorCharlieDelegate`.

Whilst it is assumed the IPT total supply should not change, if it ever does (e.g. through an owner address minting additional tokens), the hardcoded thresholds will represent smaller percentage of total supply.

Currently, to put forward a new proposal, an address is required to hold at least 1% of total supply of IPT tokens (1,000,000 tokens, assuming total supply is fixed at 100,000,000 tokens). If additional 50,000,000 tokens are minted, with proposal threshold remaining hardcoded at the same value, the percentage of total supply ownership required to submit a new proposal drops to 0.06%.

In such case, to bump the requirement back up to 1% of total supply, governance would have to submit and vote on a new proposal to modify `proposalThreshold` value.

Recommendations

To maintain stable voting and proposal thresholds based on total supply percentages, use `ipt.totalSupply()` with a relevant percentage multipliers, instead of hardcoded values.

Resolution

The development team has advised:

Since there aren't plans to reinitialize new governance contract, it would just be adding more complexity to an upgrade (mores tests), so we chose not to.

INT-16	Insufficient checks for zero values and addresses		
Asset	`GovernorDelegate.sol`, `TokenDelegate.sol`		
Status	**Closed:** Risk accepted by the project team. See Resolution for more information.		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The following functions in `GovernorDelegate.sol` allow zero addresses to be set:

- `_setNewToken()`
- `_setWhitelistGuardian()`

And the following allow zero values:

- `setMaxWhitelistPeriod()`
- `_setDelay()`
- `_setEmergencyDelay()`
- `_setVotingDelay()`
- `_setVotingPeriod()`
- `_setEmergencyVotingPeriod()`
- `_setProposalThreshold()`
- `_setQuorumVotes()` - note, setting this to 0 will disable ability to create new proposals via `propose()`
- `_setEmergencyQuorumVotes()`
- `_setWhitelistAccountExpiration()`
- `_setOptimisticDelay()`
- `_setOptimisticQuorumVotes()`

Also, in `approve()` function of `TokenDelegate.sol`, `spender` can be set to zero address.

Whilst for some of these variables zero value may be valid, for others (e.g. token addresses), it may result in an unexpected, potentially malicious behaviour.

Recommendations

Implement bounds checking, particularly for zero values and addresses. Reject zero values where deemed relevant.

Resolution

The development team has risk accepted this finding and no mitigations have been implemented.

INT-17	Malicious modification of `_usdi` or `_oracleMaster` would allow the removal of all user funds
Asset	`VaultController.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`VaultController` has two functions `registerUSDi()` and `registerOracleMaster()` which allow modification of the key variables that the contract uses. Either of these functions, if ever used mistakenly or maliciously, could have widespread effects on all funds held in the protocol.

If `_usdi` were set to a malicious value, an attacker could zero all balances except their own, mint themselves endless tokens, and then liquidate all vaults and empty the reserve.

If `_oracleMaster` were set to a malicious value, an attacker could cause all vaults to become insolvent by delivering false asset prices and then liquidate all the vaults.

Note, these threats are heavily mitigated by the fact that the functions are modified with `onlyOwner`, so the chances of them being called by an external, malicious actor are very low.

Nevertheless, if these functions are only needed in setup, their continued existence creates an unnecessarily powerful tool for accessing all the protocol's funds in the event that an attack on governance did succeed.

Recommendations

Consider whether `registerUSDi()` and `registerOracleMaster()` provide useful ongoing utility. If not, remove them and make `_usdi` or `_oracleMaster` immutable variables defined in the constructor.

Resolution

The development team has acknowledged the issue and accepted the related risk.

INT-18	Recovery of lost USDi
Asset	`USDI.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

If the protocol is exploited and funds belonging to the protocol or its users have been lost, there is no implemented mechanism to recover them.

Funds can also get locked due to a third-party contract issue without the possibility of being recovered. In this situation, there is no implemented mechanism to recover the locked funds.

Recommendations

Ensure that the `USDI` contract implements a function to recover USDi by burning USDi from a specific address and minting USDi to the owner. This function can only be called by the DAO.

Resolution

The development team are aware of the issue but have decided not to implement it.

INT-19	Direct usage of `ecrecover()` allows signature malleability
Asset	`UFragments.sol`, `GovernorDelegate.sol`, `TokenDelegate.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `permit()` and `delegateBySig()` functions call `ecrecover()` directly to verify the given signature. The `ecrecover` precompiled contract allows for malleable (non-unique) signatures and thus is susceptible to replay attacks.

Note, although a replay attack on this contract is not possible since each user's nonce is used only once, rejecting malleable signatures is considered a best practice.

Recommendations

Reject malleable signatures (i.e. require the `s` value to be in the lower half order, and the `v` value to be either `27` or `28`).

Consider using the following OpenZeppelin's ECDSA.sol library that has those checks built in.

Resolution

The development team has resolved this issue as per commit 8c5cae0.

INT-20	Sudden price drop renders vaults insolvent
Asset	Lending and oracle contracts
Status	**Closed:** See Resolution
Rating	Informational

Description

Sudden change in the price of an underlying collateral token may render the vault insolvent.

Consider a situation where an asset drops in price signifcantly without liquidations occuring. Some potential scenarios for this might be:

- The price drop is extremely rapid.
- Network delays or downtime make liquidations impossible.
- Oracle downtime or other issues prevent liquidations.
- Gas fees peak so high during a time of extreme volatility that most liquidations are not profitable.

In this situation, if the asset's LTV and price drop is high enough, it is possible for vaults that have borrowed against that asset to become insolvent.

Consider an asset with a LTV of 75% and a liquidation penalty of 10%. Suppose it has an initial price of $2,000. A user mints a vault and deposits one token in the vault and borrows the full allowance of 1500 USDi.

If the price now drops by 20%, to $1,600, the borrowing allowance will drop to $1,200, allowing liquidation. If we use the formula from page 11 of the protocol's whitepaper:

$$1500 - 1200/1600(1 - 0.1 - 0.75) = 300/1600 * 0.15 = 1.25$$

125% of the vault's assets can be liquidated to reach solvency: in practice, this means that the entire asset holding of the vault will be liquidated, but it will still not be solvent. There will still be USDi lent out against the vault, and there are no assets to be retrieved by repaying those USDi. Those USDi are also not balanced against USDC in the reserve, as they were not created by a deposit of USDC.

Consider a situation in which this occurs across a large number of vaults, leading to a large number of unbacked USDi which the protocol would have no mechanism to account for. As USDi is no longer matched by other assets, it is likely that there could be a run on USDi, the first stage of which would be emptying the reserve. This would create high interest rates of USDi, but this would not be advantageous to the protocol's stability as USDi is no longer backed by USDC. Some vault depositors would buy up USDi as the lower prices would allow them to liquidate their vaults cheaply, but this would simply remove the remaining assets from the protocol, ultimately leaving all those still holding USDi with an unbacked token for a protocol with no assets.

Recommendations

As this issue relates to the design of the protocol and not its Solidity implementation, the testing team raises this issue as *informational* for the reader to be aware of the behaviour outlined above.

Resolution

This risk was accepted by the project team. No mitigations have been implemented.

INT-21	Gas inefficient implementation of a loop in `get_vault_borrowing_power()`
Asset	`VaultController.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

`get_vault_borrowing_power()` may loop through a large number of array elements, calling external functions on every single one, leading to significant gas costs.

`get_vault_borrowing_power()` contains a loop, starting on line [**580**], which loops through all of the registered tokens and queries the vault for its balance through a call to `vault.tokenBalance()` on line [**589**] which in turn makes the external call `IERC20(addr).balanceOf(address(this))`. It also contains external calls to external oracles for each registered token through `_oracleMaster.getLivePrice()` on line [**594**].

In a situation where the protocol expands to support a large number of tokens, calls to `get_vault_borrowing_power()` could become gas heavy, especially as they contain external calls to code outside of the protocol's control. This could impact the cost and operation of the protocol.

Recommendations

This issue can be partially mitigated by carefully managing and curating the list of registered tokens and oracles. However, this does not completely mitigate the issue.

To fully address this issue, more significant changes may be required. One approach would be to make all vaults single asset. That way, there is no need for the loop. Another approach would be for vaults to track their token balances internally and return these internally stored values when `vault.tokenBalance()` is called, although this does not address issues with oracle calls.

Resolution

The development team has acknowledged the issue and stated their intention to advise governance to manage the protocol with consideration for the points raised above.

INT-22	Miscellaneous General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings of interest:

1. **Minting vaults when paused:** Consider adding the `whenNotPaused` modifier to `VaultController.mintVault()` as a precautionary measure. Generally, it is desirable for pause functionality to block as much interaction as reasonably possible to give maximum control when dealing with a potentially unforeseen issue.
2. **Liquidation slippage:** Consider adding a parameter to `VaultController.liquidateVault()` that reverts if the number of tokens being liquidated slips below it. `liquidateVault()` can modify the number of tokens liquidated, but the liquidator has no control over this process and might prefer not to pay gas if the value is below a certain level.
3. **Potential immutable variables:** Variables that are set only in the constructor could be declared as `immutable` to save gas.
   - `r0`, `r1`, `r2`, `s1`, `s2` in `ThreeLines0_100` could be declared as `immutable`.
   - `_owner` in `SlowRoll` could be declared as `immutable`.
4. **Potential constant variables:** Variables that are hard coded and never modified could be declared as `constant` to save gas.
   - `_pointsToken`, `_rewardToken` in `SlowRoll` could be declared as `constant`.
   - `MAX_SUPPLY` in `UFragments` could be declared as `constant`.
5. **File naming consistency:** Consider changing `slowroll.sol` to `SlowRoll.sol` to match the contract it contains.
6. **Unused functions:**
   - `VaultController.patchTBL()` has no comment header and will revert with an out of gas error once the protocol reaches a certain size. However, the development team has stated that this function will be removed.
   - `USDI.mint()` and `UFragments.rebase()` should be removed with its logic since they do not have an use case and they modify the total supply, affecting the entire protocol.
7. **Unused imports:** In `USDI.sol` is not necessary to import `Vault`, only `IVaultController` needs to be imported. Note that `hardhat/console.sol` should not be imported for production.
8. **Revocation of ownership:** Consider disabling `renounceOwnership()` functions inherited from `Ownable` and `OwnableUpgradeable` contracts and check for zero address in functions that change the ownership.
9. **Comment issues:**
   - The comment on line [**65**] of `ThreeLines0_100.sol` is a little questionable. The internal modification of `x_value` would not be perceptible to an external caller in this case and the return value is not dependant on `x_value` either. So it is unclear why this is mentioned here.

- The comment on line [**51**] of `ThreeLines0_100.sol` should be removed. There is no reason to leave this test in production code, commented out. However it might be helpful to explain why the approach of modifying `x_value` instead of reverting has been chosen in the case of `x_value > 1e18`.
- The comment on line [**21**] of `ThreeLines0_100.sol` should use the expression `x >= s2` to more accurately reflect the logic in the code.
- Line [**122**] of `Vault.sol` "vaults" should be "vault's".
- Line [**124**] of `Vault.sol` "decerase" should be "decrease".
- The comment on line [**125**] of `Vault.sol` assumes a reduction, but an increase is also possible.
- The comment on line [**72**] of `VaultController.sol` is slightly misstated. The modifier `onlyPauser` can only be called by `_usdi.pauser()`, not paused by `_usdi.pauser()`.
- The parameter `oracle_address` of `VaultController.registerErc20()` is not clearly commented in the header. The parameter is not the address of the oracle, but rather the address of the token which should be used when querying oracles. This should be clarified.
- The comment on line [**393**] of `VaultController.sol` is slightly misstated. The USDi is burned from the sender, not the sender's vault.
- Line [**324**], line [**360**] of `VaultController.sol` should be "indicates" in place of "indicated".
- The comment on line [**569**] of `VaultController.sol` is slightly misstated. The return value is the amount of USDi the vault can borrow, not what it owes.
- Line [**236**], line [**238**], line [**240**] of `VaultController.sol` "tokens" should be "token's".
- Line [**279**], line [**323**], line [**327**], line [**329**], line [**359**], line [**363**], line [**365**], line [**391**], line [**399**], line [**408**], line [**410**], line [**588**] of `VaultController.sol` "vaults" should be "vault's".
- Line [**410**] of `VaultController.sol` "vauls" should be "vault's".
- Line [**420**] of `VaultController.sol` "the vault liquidate" should be "the vault to liquidate".
- Line [**493**] of `VaultController.sol` "require that the vault is solvent" should be "require that the vault is not solvent".
- Line [**627**] of `VaultController.sol` "refernce" should be "reference".
- The comment on line [**586**] of `VaultController.sol` is unclear. Consider an alternative phrasing, perhaps: *"note that index 0 of _enabledTokens corresponds to a vaultId of 1, so we must subtract 1 from i to get the correct index"*.
- `VotingVaultController.retrieveUnderlying()` is missing its comment header.
- The comment on line [**55**] of `VotingVault.sol` would be more informative if it explained that the parameter `_id` is the ID of the vault which this voting vault will be attached to, and that the voting vault will have the same ID itself.
- The comment on line [**49**] of `GovernorDelegate.sol` appears to be for a different modifier.
- The comment header for `GovernorCharlieDelegate.propose()` does not mention the `emergency` parameter.
- The comment on line [**256**] of `GovernorDelegate.sol` does not fully cover the functionality of the code. In particular, the `whitelistGuardian` is not explained.
- Line [**64**] of `slowroll.sol` "contracto" should be "contract".
- The comment on line [**296**] of `USDI.sol` is incorrect: the function does not burn USDi.
- Error message on line [**178**] of `TokenDelegate.sol` should say "transferFrom", not "approve".
- Error message on line [**90**] of `GovernorDelegate.sol` should say "one pending proposal per proposer", not "one live proposal per proposer".

10. **Gas optimisations:**

- Line [**407**], line [**409**] and line [**411**] of `VaultController.sol` all make the external call `vault.baseLiability()`. The value from the first call could be saved in a memory variable and reused.

- `GovernorCharlieDelegate.getBlockTimestamp()` and `GovernorCharlieDelegate.getChainIdInternal()` would save gas if they were replaced by the `block` variables they return.
- In `TokenDelegate.permit()` and `TokenDelegate.delegateBySig()` functions perform `require(block.timestamp <= expiry);` check at the beginning of the function to avoid performing expensive calculations for expired signatures.
- A minor gas saving can be achieved in `for` loops that use `i++` by converting it to `++i`, so long as the return value of the expression is not being used.
  - Line [**209**], line [**580**] of `VaultController.sol`.
  - Line [**93**], line [**154**], line [**201**], line [**272**] of `GovernorDelegate.sol`.

11. **Event issues:** Some of these observations are of event configurations that seemed subjectively counter intuitive. It may be the case that they are configured as the development team wishes.
    - The `BorrowUSDi` event of `VaultController.sol` does not contain the information of where the USDi was borrowed to.
    - The `BorrowUSDi` event is used even when USDC is borrowed, and with no indication of this.
    - In `VotingVaultController` consider moving the event declaration from line [**55**] to the top of the contract, before `initialize()`.
    - The `ProposalCreated` event of `GovernorDelegate.sol` does not contain the information of whether the proposal is an emergency and what the quorum is.
    - Make sure all the events include the creator of the action and that the addresses are indexed.
12. **Reentrancy guard:** Keep the require test on line [**458**] of `VaultController.sol`. This require test protects from possible reentrancy calls that might be made during the external calls contained in `_liquidationMath()`. For additional protection, consider adding a reentrancy guard to `_liquidationMath()`.
13. **Decimals in capped wrapper tokens:** Both `CappedGovToken` and `CappedFeeOnTransferToken` have hard coded decimals of 18, but their underlying tokens could have a different number of decimals. This will not affect their functionality in terms of the amounts of tokens transferred. However, a user inspecting the number of wrapper tokens minted might see numbers that are 10^{12} times lower than they expect.

    On a related point, the comment on line [**22**] of `CappedFeeOnTransferToken.sol` seems to imply that the capped wrapper token has a different unit than the underlying. This is a little misleading. The decimals may be different, but the units will be the same.
14. **Documentation does not match implementation:** Numerous cases of numbers, values and logic described in documentation were found not to be matching implementation. Ensure documentation is regularly updated to stay up to date with latest implementation.

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Resolution

The comments above have been acknowledged by the development team, and relevant changes actioned in b447c19 where appropriate.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_underlying                 PASSED  [1%]
test_initialize                 PASSED  [2%]
test_decimals                   PASSED  [3%]
test_cap                        PASSED  [4%]
test_deposit                    PASSED  [6%]
test_transferFrom               PASSED  [7%]
test_initialize                 PASSED  [8%]
test_decimals                   PASSED  [9%]
test_cap                        PASSED  [10%]
test_deposit                    PASSED  [12%]
test_transferFrom               PASSED  [13%]
test_setVaultController         PASSED  [14%]
test_vaultControllerAddress     PASSED  [15%]
test_forceSetCurve              PASSED  [16%]
test_setCurve                   PASSED  [18%]
test_getValueAt                 PASSED  [19%]
test_constructor                PASSED  [20%]
test_valueAt                    PASSED  [21%]
test_read                       PASSED  [22%]
test_deployment                 PASSED  [24%]
test_constructor                PASSED  [25%]
test_tokenBalance               PASSED  [26%]
test_withdrawErc20              PASSED  [27%]
test_modifyLiability            PASSED  [28%]
test_initialize                 PASSED  [30%]
test_mintVault                  PASSED  [31%]
test_registerUSDi               PASSED  [32%]
test_registerOracleMaster       PASSED  [33%]
test_registerCurveMaster        PASSED  [34%]
test_changeProtocolFee          PASSED  [36%]
test_registerErc20              PASSED  [37%]
test_updateRegisteredErc20      XFAIL   [38%]
test_borrowUsdi                 PASSED  [39%]
test_borrowUSDIto               PASSED  [40%]
test_borrowUSDCto               PASSED  [42%]
test_repayUSDi                  PASSED  [43%]
test_repayUSDi_overpay          PASSED  [44%]
test_repayAllUSDi               PASSED  [45%]
test_liquidateVault             PASSED  [46%]
test_getters                    PASSED  [48%]
test_pause                      PASSED  [49%]
test_constructor                PASSED  [50%]
test_delegateCompLikeTo         PASSED  [51%]
test_initialize                 PASSED  [53%]
test_registerUnderlying         PASSED  [54%]
test_mintVault                  PASSED  [55%]
test_retrieveUnderlying_gov     PASSED  [56%]
test_retrieveUnderlying_fot     PASSED  [57%]
test_read                       PASSED  [59%]
test_deposit_withdraw           PASSED  [60%]
test_approve_transferFrom       PASSED  [61%]
test_deployment                 PASSED  [62%]
test_setImplementation          PASSED  [63%]
test_setters                    PASSED  [65%]
test_propose                    PASSED  [66%]
test_proposal_round_trip        PASSED  [67%]
test_proposal_DoS               XFAIL   [68%]
test_proposal_emergency         PASSED  [69%]
test_proposal_optimistic        PASSED  [71%]
test_proposal_undelegate_cancel PASSED  [72%]
test_cast_vote                  PASSED  [73%]
```

```
test_multi_vote_poc             PASSED  [74%]
test_deployment                 PASSED  [75%]
test_initialize                 XFAIL   [77%]
test_setters                    PASSED  [78%]
test_approve                    XFAIL   [79%]
test_permit                     PASSED  [80%]
test_mint                       PASSED  [81%]
test_transfer                   PASSED  [83%]
test_transferDelegateLoop       PASSED  [84%]
test_transferFrom               PASSED  [85%]
test_delegate                   PASSED  [86%]
test_selfDelegate               PASSED  [87%]
test_delegateBySig              PASSED  [89%]
test_delegateAndTransfer        PASSED  [90%]
test_getPriorVotes              PASSED  [91%]
test_revert_balance             PASSED  [92%]
test_sudden_drop                PASSED  [93%]
test_usdi_growth                PASSED  [95%]
test_withdraw_to                XFAIL   [96%]
test_withdraw_all_to            XFAIL   [97%]
test_get_points                 PASSED  [98%]
test_get_points_overbuying      XFAIL   [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

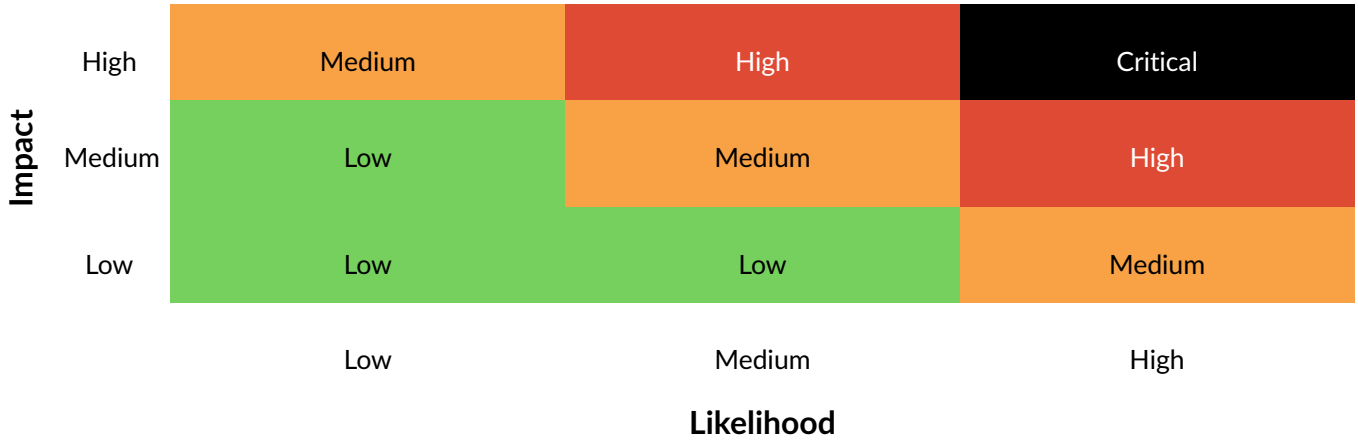

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].