DERIVE

Atomic Matching and Basis Vault

Security Assessment Report

Version: 2.0

March, 2025

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Derive components. The review focused solely on the security aspects of the Solidity implementation of the contracts, 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 components in scope. 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 Derive components 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*.

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Derive components in scope.

Overview

Derive is a decentralized protocol that creates unique and programmable onchain options, perpetuals, and structured products.

Atomic matching functionality was introduced to allow multiple actions with signatures to be submitted to the matching system, and be verified and executed in a single and convenient transaction.

Security Assessment Summary

Scope

The review was conducted on the files hosted on the Derive V2 Core and Derive V2 Matching repositories.

The scope of this time-boxed review was strictly limited to changes in the pull requests PR-358 and PR-64.

Note: third party libraries and dependencies were excluded from the scope of this assessment.

Approach

The security assessment covered components written in Solidity.

The manual review focused on identifying issues 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 identifying vulnerabilities related to known Solidity anti-patterns and attack vectors, such as re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers.

For a more detailed, but non-exhaustive list of examined vectors, see [1, 2].

To support the Solidity components of the review, the testing team also utilised the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Aderyn: `https://github.com/Cyfrin/aderyn`

Output for these automated tools is available upon request.

Coverage Limitations

Due to the time-boxed nature of this review, all documented vulnerabilities reflect best effort within the allotted, limited engagement time. As such, Sigma Prime recommends to further investigate areas of the code, and any related functionality, where majority of critical and high risk vulnerabilities were identified.

Findings Summary

The testing team identified a total of 5 issues during this assessment. Categorised by their severity:

- Medium: 1 issue.

- Low: 2 issues.
- Informational: 2 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Derive components in scope. 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 components, including 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
AMV-01	Signatures Can Be Reused To Undo Revocation	Medium	Resolved
AMV-02	Verification Checks Performed At Action Signing Can Become Stale	Low	Closed
AMV-03	Potential For Setting `lbtsaParams.leverageCeil` Equal To `lbtsaParams.leverageFloor`	Low	Resolved
AMV-04	Possible Index Out Of Bounds Error During Loop Iteration	Informational	Resolved
AMV-05	Miscellaneous General Comments	Informational	Resolved

AMV-01	Signatures Can Be Reused To Undo Revocation		
Asset	`BaseOnChainSigningTSA.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

A valid signature submitted to the function `signActionViaPermit()` has the effect of setting `signedData[actionHhash]` to `true`, allowing the action to be executed. The functions for revoking signed status for an action operate by setting `signedData[actionHhash]` to `false`. It is then possible for any user to resubmit the previously used signature to `signActionViaPermit()` to undo this revocation.

After verifying the signature, the function `signActionViaPermit()` calls this function:

```
function _signActionData(IMatching.Action memory action, bytes memory extraData) internal {
  bytes32 hash = getActionTypedDataHash(action);

  if (action.signer != address(this)) {
    revert BOCST_InvalidAction();
  }
  _verifyAction(action, hash, extraData);
  _getBaseSigningTSAStorage().signedData[hash] = true;

  emit ActionSigned(action.signer, hash, action);
}
```

All revocation functions call this internal function:

```
function _revokeSignature(bytes32 hash) internal virtual {
  _getBaseSigningTSAStorage().signedData[hash] = false;

  emit SignatureRevoked(msg.sender, hash);
}
```

There are many factors mitigating this issue. The signatures have expiry dates and action nonces, so can become unusable quickly. The sign by permit system is intended for use in an atomic manner where the signatures are immediately followed by execution, and so are unlikely to be revoked.

Nevertheless, that does not guarantee that no users will call `signActionViaPermit()` in a non-atomic manner and then wish to revoke the action.

Recommendations

Consider implementing a nonce system for each signer, and incrementing a signer's nonce every time that a signature is validated. If this nonce is included in the action hash, any revoked action hash will need a fresh signature to grant it signed status again.

Resolution

The development team has addressed this issue by restricting the callers of the `signActionViaPermit()` function to only whitelisted addresses.

This issue has been resolved in commit cf80d28.

AMV-02	Verification Checks Performed At Action Signing Can Become Stale		
Asset	`LevBasisTSA.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

Action verification checks are performed when an action is signed to confirm that the action is compatible with the TSA's financial strategic goals. However, they are not performed at action execution. For some checks this is not an issue as they are not time dependent. However, there are some checks that fetch prices that could become stale.

The following check is performed in the function `_verifyTradeAction()`. It fetches live price data with the call to `_getTradeHelperVars()` and then uses them in the function `_verifyCollateralTradeFee()` to check that the highest allowed fee, `tradeData.worstFee`, is not too high a proportion of the fetched price. A similar check is performed for the `tradeHelperVars.perpPrice`.

```
// either base or perp
 TradeHelperVars memory tradeHelperVars =
   _getTradeHelperVars(tradeData.asset, address(tsaAddresses.wrappedDepositAsset), address($.perpAsset));

 // Fees
 if (tradeHelperVars.isBaseTrade) {
   _verifyCollateralTradeFee(tradeData.worstFee, tradeHelperVars.basePrice);
 } else {
   _verifyPerpTradeFee(tradeData.worstFee, tradeHelperVars.perpPrice);
 }
```

If this check is performed on signing and finds that the fee is very close to the maximum and then the action is not executed, then over time the asset price could decrease, which would have previously caused the action to fail verification:

```
function _verifyCollateralTradeFee(uint worstFee, uint basePrice) internal view {
  CollateralManagementParams storage baseParams = _getCollateralManagementParams();

  if (worstFee > basePrice.multiplyDecimal(baseParams.feeFactor)) {
    revert CMTSA_FeeTooHigh();
  }
}
```

However, because the action is signed, it can still be executed.

The intended usage pattern of the system would be unlikely to give rise to this issue as an action of this nature would likely have a short expiry time.

Recommendations

If the development team is confident that actions will be managed correctly to avoid this issue, they may determine that no action is warranted. Alternatively, live prices could be checked on execution.

Resolution

The development team has acknowledged this issue with the following note:

> *"Acknowledged but will not be changed (downside of the signing system is that only the hash is passed back to the contract to validate "signature", so we can't run the logic unless we keep a record of what hash is what data)".*

AMV-03	Potential For Setting `lbtsaParams.leverageCeil` Equal To `lbtsaParams.leverageFloor`		
Asset	`LevBasisTSA.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `setLBTSAParams()` function on line [**148**] verifies the `lbtsaParams.leverageCeil` and `lbtsaParams.leverageFloor` parameters. However, the check only ensures that `lbtsaParams.leverageCeil` is not less than `lbtsaParams.leverageFloor`. This means that it is possible to set `lbtsaParams.leverageCeil` equal to `lbtsaParams.leverageFloor`, which contradicts the requirement that `lbtsaParams.leverageCeil` must be greater than `lbtsaParams.leverageFloor`.

If `lbtsaParams.leverageCeil` is allowed to be set equal to `btsaParams.leverageFloor` the following issues could occur:

No Risk Management Flexibility

- Users can't reduce or increase leverage to manage their exposure during market volatility.
- Typically, protocols set a range for leverage to allow traders to respond to price changes. Without this range, there's a higher risk of liquidation.

Increased Liquidation Risk

- If the market moves unfavourably, the lack of flexibility means users cannot adjust their leverage to prevent liquidation.
- This can lead to forced liquidations more easily compared to systems with adjustable leverage.

Recommendations

Update the check to ensure that `lbtsaParams.leverageCeil` cannot be less than or equal to `lbtsaParams.leverageFloor`. This will guarantee that leverage is adjustable which could potentially reduce liquidation risk.

Resolution

The development team has addressed this issue by updating the check to ensure that `lbtsaParams.leverageCeil` is always greater than `lbtsaParams.leverageFloor`.

This issue has been resolved in commit cf80d28.

AMV-04	Possible Index Out Of Bounds Error During Loop Iteration
Asset	`AtomicSigningExecutor.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `atomicVerifyAndMatch()` function iterates over the three arrays `actions`, `signatures` and `atomicActionData` reading from each index. However, there is no check to ensure that all three arrays are the same length.

This could lead to a possible index out of bounds error when accessing the same index in all three arrays.

Recommendations

Consider adding a check to ensure that all three arrays are the same size.

Resolution

The development team has addressed this issue by adding a check to ensure that all three arrays are the same size.

This issue has been resolved in commit cf80d28.

AMV-05	Miscellaneous General Comments
Asset	All contracts
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Typos**
   Related Asset(s): AtomicSigningExecutor.sol
   On line [**8**] of `AtomicSigningExecutor.sol`, "singer" should be "signer".
   Correct the typos.
2. **Mismatch Between Comment And Variable Name**
   Related Asset(s): CollateralManagementTSA.sol
   The `CollateralManagementParams` `struct` has the following on line [**29**] and line [**30**]:

   ```
   /// @dev Percentage of spot price that the TSA will sell baseAsset at in the worst case (e.g. 0.98e18)
       uint worstSpotBuyPrice;
   ```

   Here the comment mentions "sell" while the variable name contains "buy".
   Consider updating the comment to match the purpose of the variable.
3. **Possible Gas Savings By Using `calldata` Instead Of `memory` In `atomicVerifyAndMatch()`**
   Related Asset(s): AtomicSigningExecutor.sol
   The `atomicVerifyAndMatch()` function passes all of its parameters from `memory` and uses the following line which copies data from a `memory` parameter into a `memory` variable:

   ```
   AtomicAction memory atomicAction = atomicActionData[i];
   ```

   To save gas, consider changing the function parameters from `memory` to `calldata` and instead of creating a `memory` `atomicAction` variable, simply access the value directly from `calldata` using `atomicActionData[i]`.

Recommendations

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

Resolution

The development team has addressed issues 1 and 2. These issue have been resolved in commit cf80d28.

Issue 3 has been acknowledged by the development team with the following note:

> *"Left alone for consistency across codebase and readability."*

Appendix A 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.

Impact			
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium
	Low	Medium	High
		Likelihood	

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].