ETH0 Protocol

Security Review

Cantina Managed review by:

Phaze, Lead Security Researcher
Xmxanuel, Lead Security Researcher

June 9, 2025

Contents

1 Introduction **2**

1.1 About Cantina . . . 2

1.2 Disclaimer . . . 2

1.3 Risk assessment . . . 2

1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**

3.1 Medium Risk . . . 4

3.1.1 `DaoCollateral.redeem` doesn't return the corresponding fee collateral to the `redeemUser` if `CBR` is active . . . 4

3.1.2 Lido oracle creates cross-collateral arbitrage risk in multi-collateral system . . . 4

3.2 Low Risk . . . 6

3.2.1 `DaoCollateral.redeem` is not blocked in a depeg event with a zero `redeemFee` . . . 6

3.2.2 Decrease of the `eth0.mintCap` by setting to a lower amount than `eth0.totalSupply()` will block `DaoCollateral.redeem` . . . 7

3.2.3 `redeemDao()` function lacks pause protection allowing operations during system shutdown . . . 7

3.2.4 Fee calculation rounding favors users over protocol . . . 8

3.2.5 `eth0.mintCap` can block the minting of the protocol surplus . . . 8

3.3 Informational . . . 9

3.3.1 Missing mechanism to remove collateral tokens from token mapping . . . 9

3.3.2 Adding collateral tokens without oracle validation can halt minting operations . . . 9

3.3.3 Inconsistent amount validation between mint and burn functions . . . 10

3.3.4 Registry contract changes not immediately reflected in dependent contracts . . . 11

3.3.5 Outdated constants in `constants.sol` . . . 11

DRAFT

1 Introduction

1.1 About Cantina

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

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

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

1.3.1 Severity Classification

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

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

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

DRAFT

2 Security Review Summary

Usual is a Stablecoin DeFi protocol that redistributes control and redefines value sharing. It empowers users by aligning their interests with the platform's success.

From May 21st to May 27th the Cantina team conducted a review of eth0-protocol on commit hash d638edee. The team identified a total of **12** issues:

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	2	0	2
Low Risk	5	3	2
Gas Optimizations	0	0	0
Informational	5	1	4
Total	**12**	**4**	**8**

DRAFT

3 Findings

3.1 Medium Risk

3.1.1 `DaoCollateral.redeem` doesn't return the corresponding fee collateral to the `redeemUser` if `CBR` is active

Severity: Medium Risk

Context: DaoCollateral.sol#L481

The `_burnEth0TokenAndTransferCollateral` function called by `DaoCollateral.redeem` calculates the required amount of `USD0` to be burned in exchange for the underlying collateral. As a first step, the entire `eth0Amount` is burned, and afterwards, the `stableFee` amount is minted to the `$.treasuryYield`. Therefore, the `burnedEth0` describes the effectively burned `ETH0`.

```
// we burn the remaining ETH0 token
uint256 burnedEth0 = eth0Amount - stableFee;
// we burn all the ETH0 token
$.eth0.burnFrom(msg.sender, eth0Amount);
```

If the `CBR` mechanism is activated, users don't need to pay a fee for calling `redeem`. Therefore, only if the `CBR` mechanism is not , the `fee` gets minted.

```
// transfer the fee to the treasury if the redemption-fee is above 0 and CBR isn't turned on.
// if CBR is on fees are forfeited
if (stableFee > 0 && !$.isCBROn) {
    $.eth0.mint($.treasuryYield, stableFee);
}
```

However, in the following `_getTokenAmountForAmountInETH` call, which calculates the amount of `collateral` that should be returned to the user, always uses `burnedEth0` variable. This is correct if the `stableFee` has been moved to the `treasury` because the corresponding collateral for the `stableFee` should stay in the protocol.

DRAFT

```
// get the amount of collateral token for the amount of ETH0 burned by calling the oracle
returnedCollateral = _getTokenAmountForAmountInETH(burnedEth0, collateralToken);
```

This is incorrect, in the `stableFee> 0 && isCBROn==true` case. Since no `stableFee` needs to be paid, the entire collateral can be returned to the user in that case.

```
- returnedCollateral = _getTokenAmountForAmountInETH(burnedEth0, collateralToken);
+ returnedCollateral = _getTokenAmountForAmountInETH(eth0Amount, collateralToken);
```

Recommendation: The function should use the `eth0Amount` if the `cbr` mechanism is activated for the `_getTokenAmountForAmountInETH` calculation. This would also return the corresponding fee collateral to the user with the outcome that the user doesn't pay a fee.

Usual: As a part of the ETH0 design, If the CBR is on, the fees that are usually supposed to be minted for the Usual Treasury are forfeited to increase recollateralization through redemptions instead. Added a comment in PR 3.

Cantina Managed: Acknowledged. A code comment is now included to clarify this behavior.

3.1.2 Lido oracle creates cross-collateral arbitrage risk in multi-collateral system

Severity: Medium Risk

Context: LidoWstEthOracle.sol#L66

Summary: The `LidoWstEthOracle` assumes a 1:1 peg between stETH and ETH, which creates arbitrage vulnerabilities when the protocol introduces additional collateral types that use market-based pricing. This design inconsistency could allow attackers to systematically drain higher-valued collateral during stETH depeg events.

Description: The current Lido oracle implementation uses the internal stETH per token rate rather than secondary market pricing:

```
answer = int256(IWstETH(WST_ETH_CONTRACT).stEthPerToken());
```

While this provides accurate wstETH:stETH conversion rates, it assumes stETH maintains parity with ETH regardless of secondary market conditions. This assumption becomes problematic when combined with other collateral types. Consider ETH0 backed by multiple collateral types:

- 50% wstETH (valued using Lido's 1:1 assumption).
- 50% WETH (valued using market rates).

When stETH depegs to 0.8 ETH on secondary markets:

1. Protocol maintains 1:1 rate: Users can still swap stETH for ETH0 at par through the protocol.
2. Market reflects true value: ETH0's actual backing is worth `0.5 × 0.8 + 0.5 × 1.0 = 0.9 ETH`.
3. Arbitrage opportunity emerges:
   - Buy stETH at 0.8 ETH on secondary markets.
   - Swap for ETH0 at 1:1 through the protocol.
   - Redeem ETH0 for WETH at 1:1 through the protocol.
   - Net profit: 0.2 ETH per cycle.

This attack systematically drains the higher-valued collateral (WETH) while flooding the protocol with lower-valued collateral (stETH), potentially continuing until WETH reserves are exhausted.

For temporary price fluctuations, this might not pose a fundamental risk to the protocol, as backing value would be restored when the peg recovers. However, ETH0 faces greater liquidity risk than stETH itself due to instant redemption capabilities - while stETH holders must wait 1-5 days to unstake, ETH0 holders can immediately exit to any supported collateral type, amplifying the speed and impact of potential bank runs during depeg events.

The problem is worsened by stETH's withdrawal mechanics:

- Normal stETH unstaking: 1-5 day queue, subject to delays.
- Protocol conversion: Instant stETH → ETH0 → other assets.

DRAFT

This makes the protocol an attractive "fast lane" for stETH liquidity during market stress, potentially at significant cost to the protocol.

Impact Explanation: The impact is high as this design flaw could lead to systematic drainage of protocol reserves during stETH depeg events. Even if the peg is restored and backing value recovers, ETH0 will always be exposed to increased systemic risk - any depeg of a collateral token creates potential for cross-collateral arbitrage that could destabilize the entire protocol. The attack becomes more profitable and damaging as the price deviation increases and as more collateral types are added to the system.

Likelihood Explanation: The likelihood is low. While historical data shows stETH has experienced periods of depeg, for the protocol to suffer significant damage the depeg would need to persist long enough for attackers to systematically drain reserves. Most stETH depegs have been temporary, and the protocol's CBR mechanisms could potentially be activated to mitigate extended arbitrage attacks.

Recommendation:

- Primary Solution: Implement Depeg Protection for stETH: Treat stETH with the same depeg protection mechanisms used for other collateral.
  1. Add oracle validation: Use a Chainlink stETH/ETH oracle to monitor secondary market pricing.
  2. Apply depeg thresholds: Configure the existing `_checkDepegPrice()` mechanism to pause operations when stETH deviates beyond acceptable limits.
  3. Maintain internal rate precision: Continue using Lido's `stEthPerToken()` for accurate wstETH conversions, but only when stETH is within acceptable peg range.

  Implementation approach:

```
// In oracle price checks
function getPrice(address token) public view override returns (uint256) {
    if (token == WSTETH_ADDRESS) {
        // Check stETH peg using Chainlink oracle
        uint256 marketPrice = getChainlinkStETHPrice();
        _checkDepegPrice(STETH_ADDRESS, marketPrice); // Revert if depegged

        // Use precise Lido rate only when peg is maintained
        return getLidoStETHPerToken();
    }
    // ... other token logic
}
```

- Additional Safeguard: Collateral Composition Limits. Consider implementing caps on collateral backing percentages or maximum daily changes in collateral composition. This would prevent complete drainage of one collateral type in favor of another:
  - Percentage caps: Limit any single collateral type to a maximum percentage of total backing.
  - Rate limiting: Restrict how quickly the collateral composition can shift between types.
  - Minimum reserves: Maintain minimum amounts of each collateral type.

  This approach preserves the accuracy of Lido's internal rates while protecting against cross-collateral arbitrage by pausing operations during significant depeg events, similar to how other stablecoins are protected in the existing `AbstractOracle` implementation.

Usual: We use the on-chain wstETH → stETH → ETH rate (Lido oracle) because it is the source of truth for backing, immune to market noise, cheaper in gas, and follows Aave's precedent.

If ETH0 ever trades off that rate, arbitrage closes the gap and the redemption fee keeps it uneconomical to drain funds.

Additionally, a Chainlink stETH/ETH feed Watcher Bot watches for extreme deviations and auto-pauses DaoCollateral if Lido is ever hacked, so we get black-swan protection without daily oracle freezes.

If any of these conditions change, we can additionally also swap to a different oracle at any time on the Usual Protocol itself.

Cantina Managed: Acknowledged.

3.2 Low Risk

3.2.1 `DaoCollateral.redeem` is not blocked in a depeg event with a zero `redeemFee`

Severity: Low Risk

Context: DaoCollateral.sol#L481

Description: The `eth0.mint` function includes a `collateralBackingInETH` check. If not sucessful the `mint` call will revert. Since the `redeem` function requires minting the `eth0` `stableFee` into the treasury, a revert in `eth0.mint` due to the `collateralBackingInETH` check would revert the entire `redeem` transaction.

```
// transfer the fee to the treasury if the redemption-fee is above 0 and CBR isn't turned on.
// if CBR is on fees are forfeited
if (stableFee > 0 && !$.isCBROn) {
    $.eth0.mint($.treasuryYield, stableFee);
}
```

While this behavior is intended by the protocol to block the `redeem` calls until the `CBR` mechanism gets activated. There is still the edge case, that the `redeemFee` could be set to zero basis points. This would result in a `stableFee=0` and no follow-up `eth0.mint` call, which would allow to sucessful `redeem` if the protocol has depegged.

Recommendation: To ensure the `redeem` function is consistently blocked during a depeg befor the `CBR` get's activated, consider requiring the `redeemFee > 0`. Alternatively, introduce a public function in `eth0` for the `collateralBackingInETH` and ensure it is called in all scenarios.

Usual: Fixed in commit 1f715f87.

DRAFT

Cantina Managed: Fixed as recommended.

3.2.2 Decrease of the `eth0.mintCap` by setting to a lower amount than `eth0.totalSupply()` will block `DaoCollateral.redeem`

Severity: Low Risk

Context: Eth0.sol#L175

Description: In case the mint cap for `ETH0` is decreased by calling `eth0.setMintCap` and a big `swap` transaction happens before, it can result in a state where the `totalSupply() > mintCap`. The `DaoCollateral.redeem` first burns all eth0 tokens and afterwards `eth0.mint` to `mint` the fees into the treasury.

However, if `redeem` amount is smaller than the difference between `totalSupply() - mintCap` the `eth0.mint` call would revert because of the `mintCap` constraint. Even after burning the `eth0` tokens the `totalSupply()` would be above the `mintCap` and `eth0.mint` would revert. This would result in a state where it is not possible to call `DaoCallateral.swap` since the `mintCap` has been reached, but it would block the reward of smaller amounts.

Recommendation: Don't allow setting the `mintCap` below the current `totalSupply` by adding the following check to the `eth0.setMintCap` function:

```
if (newMintCap < totalSupply()) {
    revert MintCapTooSmall();
}
```

Usual: Fixed in commit 1f715f87.

Cantina Managed: Fixed as recommended.

3.2.3 `redeemDao()` function lacks pause protection allowing operations during system shutdown

Severity: Low Risk

Context: DaoCollateral.sol#L560

Description: The `redeemDao()` function in the DaoCollateral contract is not protected by the `whenNotPaused` modifier, unlike the regular `redeem()` and `swap()` functions. This creates an inconsistency in the contract's pause mechanism and potentially allows DAO redemptions to continue even during system-wide emergency shutdowns.

The current pause structure includes:

- `redeem()`: Protected by both `whenRedeemNotPaused` and `whenNotPaused`.
- `swap()`: Protected by both `whenSwapNotPaused` and `whenNotPaused`.
- `redeemDao()`: Only protected by `nonReentrant`, no pause modifiers.

This design means that when the contract is globally paused via `pause()`, regular user operations are halted but DAO redemptions can continue unimpeded. While this might be intentional to allow DAO operations during emergencies, it creates a potential gap in emergency response capabilities.

If there's a critical issue requiring a complete system shutdown (such as a security vulnerability or oracle failure), administrators currently have no way to prevent DAO redemptions from occurring, which could potentially worsen the situation or interfere with emergency response procedures.

Recommendation: Consider adding the `whenNotPaused` modifier to `redeemDao()` to enable complete system shutdown when necessary:

```
  function redeemDao(address collateralToken, uint256 amount)
      external
      nonReentrant
+     whenNotPaused
  {
      // ... function implementation
  }
```

This change would establish a clear hierarchy:

DRAFT

- Individual function pauses (`whenSwapNotPaused`, `whenRedeemNotPaused`) for targeted operational control.
- Global pause (`whenNotPaused`) for system-wide emergency shutdown that affects all operations including DAO redemptions.

If DAO operations need to remain available during specific emergencies, this can be achieved by using only the individual pause functions rather than the global pause.

Usual: Fixed in commit 1f715f87.

Cantina Managed: Fixed as recommended.

3.2.4 Fee calculation rounding favors users over protocol

Severity: Low Risk

Context: DaoCollateral.sol#L450-L459

Description: In the `_calculateFee()` function of the DaoCollateral contract, the fee calculation and normalization steps consistently round in favor of users rather than the protocol. This occurs in two places:

1. Initial fee calculation: Uses `Math.Rounding.Floor` which rounds down.
2. Normalization process: The `wadAmountToDecimals()` and `tokenAmountToWad()` functions in the Normalize library use floor rounding by default.

```
stableFee = Math.mulDiv(eth0Amount, $.redeemFee, BASIS_POINT_BASE, Math.Rounding.Floor);
// ...
if (tokenDecimals < 18) {
    stableFee = Normalize.tokenAmountToWad(
        Normalize.wadAmountToDecimals(stableFee, tokenDecimals), tokenDecimals
    );
}
```

Both operations consistently round down, meaning users pay slightly less in fees than the intended percentage, and the protocol collects less revenue than designed.

While the team has indicated this behavior is intentional (carried over from the USD0 protocol), it represents a design choice where precision loss consistently favors users over the protocol's fee collection.

Recommendation: This appears to be an intentional design decision. However, if the protocol wishes to ensure it collects the full intended fee amount, consider using ceiling rounding for fee calculations:

```
- stableFee = Math.mulDiv(eth0Amount, $.redeemFee, BASIS_POINT_BASE, Math.Rounding.Floor);
+ stableFee = Math.mulDiv(eth0Amount, $.redeemFee, BASIS_POINT_BASE, Math.Rounding.Ceil);
```

The current implementation prioritizes user-friendly rounding at the expense of protocol fee collection accuracy.

Usual: We acknowledge this and will keep this in favor for user, impact should be marginal.

Cantina Managed: Acknowledged.

DRAFT

3.2.5 `eth0.mintCap` can block the minting of the protocol surplus

Severity: Low Risk

Context: Eth0.sol#L139

Description: The underlying collateral of `eth0` will increase in value. In the case of `wstETH`, this will happen because of Lido staking rewards. `Eth0` holders can only redeem the equivalent of `1 ether` for `1 eth0` but not the rewards. The surplus is captured by the protocol in the form of newly issued `eth0`. Inside the `eth0.mint` function, there is a check to ensure that the overall `totalSupply` can never be higher than the `mintCap`.

The permissioned `eth0.mint` call is intended to be used by governance to issue the protocol surplus. However, if the `mintCap` is reached, it won't be possible to issue the surplus for the protocol. It would be required to increase the `mintCap` again only to mint the `protocol` surplus. The `eth0.setMintCap` and `minting` of the protocol surplus are required to be in the same transaction. Otherwise, other users could use the increased mintCap to swap again.

Recommendation: Consider adding another permissioned function for minting the surplus, which can increase the mintCap if needed. A higher totalSupply() than the mintCap can block the redeem functionality as described in another issue. Therefore, we would recommend increasing it instead of not performing the check for surplus minting.

Usual: We can raise mintcap when price of wstETH raised significantly over time. No need to do anything.

Cantina Managed: Acknowledged.

3.3 Informational

3.3.1 Missing mechanism to remove collateral tokens from token mapping

Severity: Informational

Context: TokenMapping.sol#L77-L78

Description: The TokenMapping contract only provides functionality to add collateral tokens via addEth0CollateralToken() but lacks a corresponding mechanism to remove tokens once they have been added. This design limitation can lead to operational issues in several scenarios.

When a collateral token becomes problematic (faulty, insolvent, or deprecated), it cannot be removed from the system. This creates potential operational risks:

- A faulty token could cause the entire Eth0.mint() function to halt when it attempts to iterate through all collateral tokens to calculate backing.
- If a token becomes insolvent, the protocol may need to pause operations, but resuming would require removing the problematic token first.
- The current implementation permanently locks tokens into the mapping with no administrative override.

While such issues could potentially be resolved through contract upgrades in worst-case scenarios, this approach introduces unnecessary complexity and delay in emergency situations.

Recommendation: Consider adding an administrative function to remove collateral tokens from the mapping. This would provide administrators with the flexibility to respond to problematic tokens without requiring contract upgrades.

Usual: It was decided not to add this functionality to be consistent with pegasus repo.

Cantina Managed: Acknowledged.

DRAFT

3.3.2 Adding collateral tokens without oracle validation can halt minting operations

Severity: Informational

Context: TokenMapping.sol#L77-L78

Description: The addEth0CollateralToken() function in the TokenMapping contract does not verify that an oracle has been initialized for the collateral token being added. This oversight can cause all ETH0 minting operations to fail when the system attempts to calculate collateral backing.

When Eth0.mint() is called, it iterates through all registered collateral tokens and calls oracle.getPrice(collateralToken) to calculate the total backing. If any token lacks an initialized oracle, this call will revert with "OracleNotInitialized", effectively halting all minting operations until the issue is resolved.

The current flow allows for a problematic sequence:

1. Admin adds a collateral token via addEth0CollateralToken().
2. Token is successfully added to the mapping.
3. Oracle initialization is delayed or forgotten.
4. Any attempt to mint ETH0 fails when calculating collateral backing.

This creates an operational vulnerability where adding tokens and initializing their oracles are not atomic operations, potentially disrupting core protocol functionality.

Recommendation: Consider adding oracle validation to the `addEth0CollateralToken()` function to ensure the token has a properly initialized oracle before being added to the mapping:

```
  function addEth0CollateralToken(address collateral) external returns (bool) {
      if (collateral == address(0)) {
          revert NullAddress();
      }
      // check if there is a decimals function at the address
      // and if there is at least 1 decimal
      // if not, revert
      if (IERC20Metadata(collateral).decimals() == 0) {
          revert Invalid();
      }

      TokenMappingStorageV0 storage $ = _tokenMappingStorageV0();
      $._registryAccess.onlyMatchingRole(DEFAULT_ADMIN_ROLE);

+     // Verify that oracle is initialized for this token
+     IOracle oracle = IOracle($.registryContract.getContract(CONTRACT_ORACLE));
+     try oracle.getPrice(collateral) returns (uint256) {
+         // Oracle exists and is working
+     } catch {
+         revert OracleNotInitialized();
+     }

      // is the collateral token already registered as a ETH0 collateral
      if ($.isEth0Collateral[collateral]) revert SameValue();
      // ... rest of function
  }
```

Alternatively, consider making the token addition and oracle initialization atomic by combining both operations into a single administrative function.

Usual: Acknowledged, shall be no risk involved as we validate everything before launch and test on Tenderly.

DRAFT

Cantina Managed: Acknowledged.

3.3.3 Inconsistent amount validation between mint and burn functions

Severity: Informational

Context: Eth0.sol#L167-L169

Description: The ETH0 contract has inconsistent input validation between its `mint()` and burn functions. The `mint()` function includes a check to revert when `amount == 0`, but the `burnFrom()` and `burn()` functions lack this validation, creating inconsistency in the contract's input handling.

Current validation patterns:

- `mint()`: Includes `if (amount == 0) revert AmountIsZero();`.
- `burnFrom()`: No zero amount validation.
- `burn()`: No zero amount validation.

While burning zero tokens is technically a no-op that doesn't cause harm, the inconsistency could lead to confusion and unexpected behavior differences between similar operations. More importantly, in the context of the broader protocol, burn operations are often associated with releasing collateral or other state changes, so it's important to prevent scenarios where collateral could be released while burning zero tokens.

Recommendation: Consider adding zero amount validation to the burn functions for consistency:

```
  function burnFrom(address account, uint256 amount) public {
+     if (amount == 0) {
+         revert AmountIsZero();
+     }
      Eth0StorageV0 storage $ = _eth0StorageV0();
      $.registryAccess.onlyMatchingRole(ETH0_BURN);
      _burn(account, amount);
  }

  function burn(uint256 amount) public {
+     if (amount == 0) {
+         revert AmountIsZero();
+     }
      Eth0StorageV0 storage $ = _eth0StorageV0();
      $.registryAccess.onlyMatchingRole(ETH0_BURN);
      _burn(msg.sender, amount);
  }
```

This change would ensure consistent input validation across all token operations and provide clearer error messages for invalid inputs.

Usual: Fixed in commit 1f715f87.

Cantina Managed: Fixed as recommended.

3.3.4 Registry contract changes not immediately reflected in dependent contracts

Severity: Informational

Context: DaoCollateral.sol#L218-L225

Description: The DaoCollateral contract (and other contracts in the protocol) cache contract addresses from the registry during initialization but do not update these cached addresses when the registry is modified. This creates a potential lag between registry updates and their reflection in dependent contracts. During initialization, the contract fetches and stores contract addresses:

```
IRegistryContract registryContract = IRegistryContract(_registryContract);
$.registryAccess = IRegistryAccess(registryContract.getContract(CONTRACT_REGISTRY_ACCESS));
$.treasury = address(registryContract.getContract(CONTRACT_TREASURY));
$.tokenMapping = ITokenMapping(registryContract.getContract(CONTRACT_TOKEN_MAPPING));
$.eth0 = IEth0(registryContract.getContract(CONTRACT_ETH0));
$.oracle = IOracle(registryContract.getContract(CONTRACT_ORACLE));
$.treasuryYield = registryContract.getContract(CONTRACT_YIELD_TREASURY);
```

If any of these contract addresses are updated in the registry after initialization, the DaoCollateral contract will continue using the old cached addresses until it is upgraded or redeployed. While this design pattern is common across the Usual protocol contracts, it creates a potential operational issue where registry updates don't take immediate effect.

Recommendation: This appears to be a deliberate architectural choice that prioritizes gas efficiency over immediate registry synchronization and this limitation should be documented.

Usual: Acknowledged, not going to fix.

Cantina Managed: Acknowledged.

3.3.5 Outdated constants in `constants.sol`

Severity: Informational

Context: constants.sol#L72

Description: The `constants.sol` defines constants that are for the `USD0` deployment. Like the `USUAL_MULTISIG_MAINNET`, `REGISTRY_CONTRACT_MAINNET` or the `USUAL_PROXY_ADMIN_MAINNET`. A deployed `LIDO_STETH_ORACLE_MAINNET` is not used in the codebase.

Recommendation: Remove the outdated constants from the `constants.sol` file.

Usual: Acknowledged. This will be changed/replaced during deployment phase when we will have all new multisigs ready.

DRAFT

Cantina Managed: Acknowledged.

DRAFT