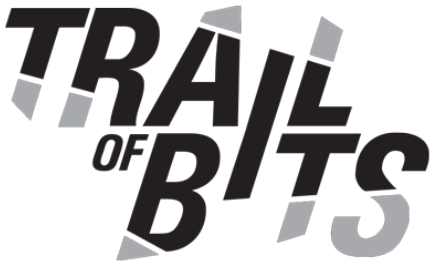

CREAM

Summary Report

January 28th, 2021

Prepared For:
Leo Cheng | *CREAM Finance*
leo@machix.com

Jeremy Yang | *CREAM Finance*
jeremy@cream.finance

Eason Wu | *CREAM Finance*
eason@cream.finance

Prepared By:
Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Maximilian Krüger | *Trail of Bits*
max.kruger@trailofbits.com

Review Summary

From January 25 to January 27, 2021, Trail of Bits performed an assessment of the CREAM smart contracts with two engineers, working from commit `2e83fc3` from `CreamFi/compound-protocol` as well as commit `8c44071` from the `cream-v2` branch of the same repository. CREAM is a fork of the Compound lending protocol with additional features. Due to the short length of the engagement, we focused our review on changes introduced by the fork.

Throughout this assessment, we sought to answer various questions about the security of CREAM. We focused on flaws that would allow an attacker to:

- Manipulate asset prices returned by the price oracles.
- Subvert the imposed caps on borrowing or supplying.
- Bypass access controls to modify contract state.

This review resulted in three findings ranging from medium to informational in severity. The medium-severity issue describes how the price oracle acts as a single point of failure for the system. The remaining informational issues highlight the lack of documentation accompanying the system as well as the use of `ABIEncoderV2` in some of the contracts. Additionally, we reported several code quality suggestions.

On the following page, we review the maturity of the codebase and the likelihood of future issues. In each area of control, we rate the maturity from strong to weak, or missing, and give a brief explanation of our reasoning. Appendix B provides a list of recommendations to consult when considering adding support for new assets. Appendix C includes guidance on handling sensitive key material.

Additionally, CREAM Finance should consider these steps to improve their security maturity:

- Improve the documentation of the system, especially the differences from the original Compound protocol.
- Integrate fuzzing or symbolic execution to test the correctness of contract functionality.
- Follow best practices for privileged accounts, e.g., use a multisig wallet for the owner, and consider the use of an HSM (see our HSM recommendations).
- Follow best practices when using price oracles.
- Conduct further in-depth review focused on the off-chain price oracle infrastructure.

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** Adequate access controls were in place for all privileged operations.
Arithmetic	**Satisfactory.** All relevant arithmetic was checked for errors using the custom CarefulMath library.
Assembly Use	**Satisfactory.** The usage of assembly was minimal and limited to areas where it was necessary.
Centralization	**Weak.** An oracle operated by CREAM finance was used as a fallback for certain assets. Additionally, the Comptroller admin address had the authority to replace the oracle at any time.
Contract Upgradeability	**Satisfactory.** The system used the delegatecall proxy pattern for upgradeability and no issues were identified.
Function Composition	**Satisfactory.** Functions were organized and scoped appropriately. Code that was added or modified appeared to be consistent with the existing code style.
Front-Running	**Satisfactory.** We did not identify any issues related to front-running.
Monitoring	**Satisfactory.** All functions that made important state modifications emitted events.
Specification	**Missing.** Official documentation was very minimal. As the project was a fork of Compound, much of the relevant documentation already existed. However, the specific differences from Compound were not documented clearly.
Testing & Verification	**Satisfactory.** The repository included tests for a variety of scenarios.

Appendix A. Code Maturity Classifications

Code Maturity Classes	
Category Name	**Description**
Access Controls	Related to the authentication and authorization of components.
Arithmetic	Related to the proper use of mathematical operations and semantics.
Assembly Use	Related to the use of inline assembly.
Centralization	Related to the existence of a single point of failure.
Upgradeability	Related to contract upgradeability.
Function Composition	Related to separation of the logic into functions with clear purpose.
Front-Running	Related to resilience against front-running.
Key Management	Related to the existence of proper procedures for key generation, distribution, and access.
Monitoring	Related to use of events and monitoring procedures.
Specification	Related to the expected codebase documentation.
Testing & Verification	Related to the use of testing techniques (unit tests, fuzzing, symbolic execution, etc.).

Rating Criteria	
Rating	**Description**
Strong	The component was reviewed and no concerns were found.
Satisfactory	The component had only minor issues.
Moderate	The component had some issues.

Weak	The component led to multiple issues; more issues might be present.
Missing	The component was missing.
Not Applicable	The component is not applicable.
Not Considered	The component was not reviewed.
Further Investigation Required	The component requires further investigation.

Appendix B. Token Integration Checklist

The following checklist provides recommendations when interacting with arbitrary tokens. Every unchecked item should be justified and its associated risks understood. An up to date version of the checklist can be found in crytic/building-secure-contracts.

For convenience, all Slither utilities can be run directly on a token address, such as:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken
```

To follow this checklist, you will want to have this output from Slither for the token:

```
- slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
- slither [target] --print human-summary
- slither [target] --print contract-summary
- slither-prop . --contract ContractName # requires configuration, and use of
Echidna and Manticore
```

General Security Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (aka "level of effort"), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on blockchain-security-contacts.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

ERC Conformity

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use slither-check-erc to review that:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **`The name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and might not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. If this is the

case, ensure the value returned is below 255.

- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.
- ❏ **The token is not an ERC777 token and has no external function call in `transfer` and `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use slither-prop to review that:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests, then check the properties with Echidna and Manticore.

Finally, there are certain characteristics that are difficult to identify automatically. Review for these conditions by hand:

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest might be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unneeded complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non–token-related functions.** Non–token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.
- ❏ **The token only has one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g. balances[token_address][msg.sender] might not reflect the actual balance).

Owner privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts might change their rules over time. Use Slither's `human-summary` printer to determine if the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pauseable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams, or that reside in legal shelters should require a higher standard of review.

Token Scarcity

Reviews for issues of token scarcity requires manual review. Check for these conditions:

- ❏ **No user owns most of the supply.** If a few users own most of the tokens, they can influence operations based on the token's repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange can compromise the contract relying on the token.
- ❏ **Users understand the associated risks of large funds or flash loans.** Contracts relying on the token balance must carefully take in consideration attackers with large funds or attacks through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

Appendix C. Handling Key Material

The safety of key material is important in any system, but particularly so in Ethereum; keys dictate access to money and resources. Theft of keys could mean a complete loss of funds or trust in the market. The current configuration uses an environment variable in production to relay key material to applications that use these keys for interacting with on-chain components. However, attackers with local access to the machine may be able to extract these environment variables and steal key material, even without privileged positions. Therefore, we recommend the following:

- Move key material from environment variables to a dedicated secret management system with trusted computing capabilities. The two best options for this are Google Cloud Key Management System (GCKMS) or Hashicorp Vault with Hardware Security Module (HSM) backing.
- Restrict access to GCKMS or Hashicorp Vault to only those applications and administrators that must have access to the credential store.
- Local key material, such as keys used by fund administrators, may be stored in local HSMs, such as YubiHSM2.
- Limit the number of staff members and applications with access to this machine.
- Segment the machine away from all other hosts on the network.
- Ensure strict host logging, patching, and auditing policies are in place for any machine or application that handles said material.
- Determine the business risk of a lost or stolen key, and what the Disaster Recovery and Business Continuity (DR/BC) policies are in the event of a stolen or lost key.