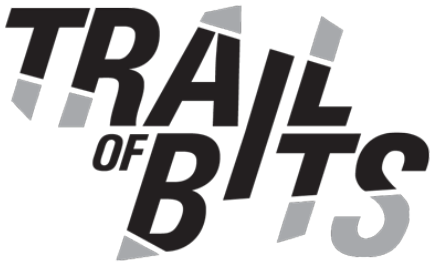

dToken

Security Assessment

July 13, 2020

Prepared For:
Mindao Yang | *dForce*
hy@dforce.network

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

Sam Sun | *Trail of Bits*
sam.sun@trailofbits.com

Review Summary

From June 29 to July 10, 2020, Trail of Bits performed an assessment of dForce's dToken smart contracts with two engineers over two person-weeks. We reported 14 issues ranging from medium to informational severity and made several code quality suggestions.

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

- Gain unauthorized access to user funds.
- Bypass access controls to modify contract state.
- Interfere with interactions between dToken components.

The two medium-severity issues concerned 1) the heavy centralization of the system, which could allow a malicious insider to drain user funds, and 2) the system's inability to natively handle airdropped tokens such as COMP. Two similar low-severity issues describe how adding duplicate handlers could move the system into an inconsistent state, while a third low-severity issue describes a method of manipulating the Aave interest rate. Several of the informational issues concern external interactions with common tokens that do not strictly implement the ERC20 standard.

The dForce team began fixing the issues as they were reported. See updated versions of the codebase in the Project Dashboard.

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. dForce should consider these steps to improve their security maturity:

- Integrate fuzzing or symbolic execution to test the correctness of contract functionality.
- Use crytic.io for any new code development.
- Follow best practices for privileged accounts, e.g., use a multi-sig wallet for authorized users, and consider using an HSM (see our HSM recommendations).

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** Appropriate access controls were in place for performing privileged operations. Slither identified one function missing a `whenNotPaused` modifier but this omission did not have serious security implications.
Arithmetic	**Satisfactory.** The contracts made consistent use of safe arithmetic library functions to prevent overflow.
Assembly Use	**Not Applicable.** The contracts did not include any assembly outside of the vendored OpenZeppelin libraries.
Centralization	**Moderate.** Authorized users of the system were able to make significant changes to the system such that a malicious insider could trivially drain funds from the contracts. The authorization system could eventually be migrated to a decentralized governance model.
Contract Upgradeability	**Satisfactory.** The contracts made use of the OpenZeppelin proxy upgradeability implementation.
Function Composition	**Satisfactory.** Most functions were organized and scoped appropriately. We suggested more consistent names along with some shared code deduplication for the dToken redemption functions.
Front-Running	**Satisfactory.** dToken included the common `increaseAllowance` and `decreaseAllowance` functions to help mitigate the ERC20 race condition.
Monitoring	**Satisfactory.** No functions were identified that would benefit from additional events. Some events did not have indexed parameters, and we noted that one event had parameters reversed relative to similar events.
Specification	**Moderate.** The code had adequate comment coverage, but the project documentation and specification outside of source files was minimal.
Testing & Verification	**Satisfactory.** The repositories included tests for a variety of scenarios.

Project Dashboard

Versions 1.0 and 1.1 formed the basis of the review. The subsequent versions were reviewed to verify that the changes made correctly remedied the issues and did not introduce new vulnerabilities.

Commit hashes of the reviewed versions from the `dforce-network/dToken` repository:

- Audit Version 1.0: `9adc11f`
- Audit Version 1.1: `e8492c4`
- Audit Version 1.2: `06e34e4`
- Audit Version 1.3: `c9b874a`
- Audit Version 1.4: `00a02f2`

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.

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

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

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 calls 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.

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.