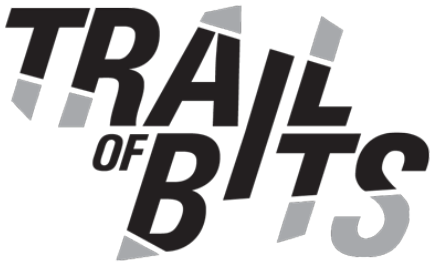

Orbit Dai

Rapid Code Review

April 23, 2020

Prepared For:
TK Park | *Ozys*
tk@ozys.net

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

Review Summary

From April 21 to April 22, 2020, Trail of Bits performed an assessment of Orbit Dai with one engineer (two person-days), and reported five issues ranging from medium to informational severity. *On June 16, Trail of Bits reviewed and verified the fixes to three of the five reported issues and one of the code quality concerns.*

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

- improperly mint `ODAI` or `EDAI`,
- access other users' funds, or
- gain ownership of the Proxy contract.

The medium-severity issue pertained to the lack of mitigations for the known ERC20 race condition. The low-severity issue was due to the lack of any tests accompanying the code. The informational issues related to some areas where events would be useful, the overly permissive version pragma, and the fact that the infinite allowance in the `transferFrom` function is not documented. Finally, we provided code quality recommendations to improve the readability of the code, such as improving the lack of comment coverage and using an enum to represent the contract state. Ozys should address all five reported issues.

We also strongly recommend writing unit tests and integration tests with a copy of the Dai contracts before deployment. Testing the contracts for correct interactions with Dai is crucial. Furthermore, missing events should be added, and effective monitoring of the deployed contracts should be in place.

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

- Use fuzzing or symbolic execution to test the correctness of the arithmetic functions, in particular the `muldiv` function.
- Use crytic.io for any new code development.
- Locate and document all privileged accounts.
- Follow best practices for privileged accounts, e.g., use a multi-sig wallet for the owner, and consider the use of an HSM (see our HSM recommendations).
- Write an incident response plan that details how funds will be managed if `killProxy` is called, among other scenarios.
- Describe the properties and responsibilities of the reserve contract (e.g., who, if anyone, will control it, whether that entity will have access to the funds held in reserve, etc.).

Code Maturity Evaluation

Category Name	Description
Access Controls	**Strong**. All critical functions had appropriate access control modifiers.
Arithmetics	**Satisfactory**. The contracts incorporated their own arithmetic functions throughout to prevent overflow. However, the `muldiv` function was not intuitive and needed more documentation.
Assembly Use	**Not Applicable.** The codebase did not use inline assembly.
Centralization	**Moderate.** The proxy owner was able to change the reserve address or migrate the proxy contract's balance unilaterally.
Contract Upgradeability	**Strong.** The proxy contract had built-in functionality for the owner to properly migrate balances to a new proxy version.
Function Composition	**Strong.** Contracts and functions were organized and scoped appropriately.
Front-Running	**Satisfactory.** Most functionality would not have been negatively impacted by attempts at front-running. However, the token contracts did not include a mitigation for the ERC20 `approve` race condition.
Monitoring	**Weak**. Several key operations lacked events.
Specification	**Moderate.** The high-level documentation for the code was comprehensive; however, the contracts had poor comment coverage.
Testing & Verification	**Missing.** No unit or integration tests were provided.

Categories

Category Name	Description
Access Controls	Related to the authentication and authorization of components.
Arithmetics	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.

SCRATCH

Control Rating	Description
Strong	Controls were well implemented, centrally located, non-bypassable, and robustly designed.
Satisfactory	Controls were well implemented, but may be weakened by vulnerabilities or are diffuse in location.
Adequate	Controls were implemented to industry-standard best practice guidelines.
Weak	Controls were either partially unimplemented or applied, or contained flaws in their design or location.
Missing	An entire family of control was missing from a component.
Not Applicable	This control family is not needed for protecting the current component.
Further Investigation Required	This control family requires further investigation. Controls were not sufficiently evaluated to receive a control rating.