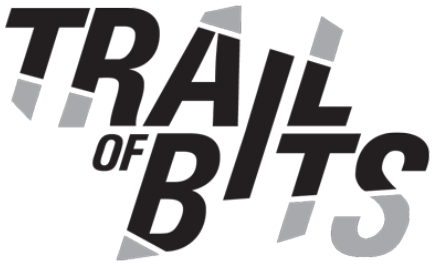

Warp v2

Summary Report

June 10, 2021

Prepared For:
Michael Geike | *Advanced Blockchain*
geike@advancedblockchain.com

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

Review Summary

Code Maturity Evaluation

Appendix A. Code Maturity Classifications

Appendix B. Token Integration Checklist
- General Security Considerations
- ERC Conformity
- Contract Composition
- Owner Privileges
- Token Scarcity

Review Summary

From June 7 to June 9, 2021, Trail of Bits performed an assessment of the Warp v2 smart contracts with one engineer working from commit hash `937259c` from the `warpfinance/blacksmith` repository. Trail of Bits previously reviewed this codebase as part of a larger assessment in April 2021. This review targeted fixes to issues reported during the previous assessment as well as changes to the codebase since commit hash `b682d99`. In combination with manual review, we also used Slither, our Solidity static analyzer, to provide broad coverage of the codebase.

One new high-severity issue was identified involving the use of constructors in combination with `delegatecall`-based proxies. This issue has been fixed. The previous assessment identified 12 issues that affected Warp v2. Four high-severity, two low-severity, and two informational-severity issues have been fixed. The one remaining low-severity issue highlights development toolchain dependencies identified by `yarn audit` as containing known vulnerabilities. The three remaining informational-severity issues describe potential risks and edge cases introduced by EIP-2612 support, highlight that the actual number of blocks per year will stray from the hardcoded value used by the codebase, and cautioning against the use of the new ABI encoder until it has been more rigorously tested.

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 also includes guidance for interacting with arbitrary tokens.

Additionally, the Warp Finance team should consider these steps to improve their security maturity:

- Use the `slither-check-upgradeability` tool to help detect common issues when developing upgradeable or proxied contracts.
- 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.

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** All privileged functionality had adequate access controls in place.
Arithmetic	**Satisfactory.** The contracts used version 0.8.1 of the Solidity compiler that includes checked math which will revert in case of overflow.
Assembly Use	**Satisfactory.** The contracts used minimal assembly code for proxy purposes or to retrieve the chain ID.
Centralization	**Moderate.** Lending pairs rely on the `PriceOracleAggregator` as a price feed. The admin of that contract can update the address of an oracle for any asset. All other contracts' owners have less ability to influence the system. The owner of the factory contracts only has the ability to upgrade implementations for newly deployed vaults and lending pairs. The owner of a lending pair is only able to pause deposits and borrows, but not withdrawals.
Code Stability	**Strong.** The code did not change during the assessment.
Upgradeability	**Moderate.** One issue was identified through the use of constructors in some proxied contracts. This would have prevented the `PriceOracleAggregator` admin role from being set and also would have resulted in vaults using an incorrect domain separator.
Function Composition	**Satisfactory.** Functions and contracts were organized and scoped appropriately.
Front-Running	**Satisfactory.** No new front-running issues were identified.
Monitoring	**Satisfactory.** All functions that made important state modifications emitted events.
Specification	**Satisfactory.** The contracts were thoroughly commented and were accompanied by high-level documentation.
Testing &	**Satisfactory.** The codebase had test coverage across all of the

Verification	system components and included a variety of test 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 purposes
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 the 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 for 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 the following:

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

To follow this checklist, use the below 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 (i.e., the 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 the following:

- ❏ **`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 may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such

cases, ensure that 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` or `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 the following:

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

Finally, there are certain characteristics that are difficult to identify automatically. Conduct a manual review of the following conditions:

- ❏ **`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 may be trapped in the contract if not taken into account.

Contract Composition

- ❏ **The contract avoids unnecessary 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 has only 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] may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may 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 pausable 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 teams that reside in legal shelters require a higher standard of review.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' 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 could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed 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.