DeFi SDK

Security Assessment

December 18, 2020

Prepared For:
Evgeny Yurtaev | *Zerion*
evgeny@zerion.io

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

Will Song | *Trail of Bits*
will.song@trailofbits.com

Jim Miller | *Trail of Bits*
james.miller@trailofbits.com

Changelog:
November 16, 2020: Initial report delivered
December 18, 2020: Added Appendix D. Fix Log

Executive Summary

From November 2 through November 13, 2020, Zerion engaged Trail of Bits to review the security of their DeFi SDK. Trail of Bits conducted this assessment over the course of three person-weeks with three engineers working from commit `ddff636` of the `interactive` branch from the `zeriontech/defi-sdk` repository.

In the first week, we focused on understanding the codebase. We reviewed the contracts for common Solidity flaws, both manually and through static analysis with Slither. In the final week, we completed our manual review with a more in-depth focus on the interactions between contracts as well as any logic issues specific to DeFi SDK.

Our review resulted in four informational-severity findings. Two of these findings related to signature construction and handling, and the remaining two issues highlighted the use of Solidity compiler features that are not enabled by default and so may not be as thoroughly tested as the underlying compiler. We also make several code quality recommendations in Appendix C.

Overall, the code follows Solidity development best practices. It has a suitable architecture and is properly documented. The system architecture, specifically the logical separation between the Router and Core contracts, minimizes the attack surface of a malicious or malfunctioning adapter. The interactions between components are well defined. The functions are small, and have a clear purpose.

Trail of Bits recommends addressing the signature-related issues and keeping an eye on the development of the optional compiler features used by the DeFi SDK. We also recommend reviewing our hardware wallet guidance for the owner address of the contracts, and reviewing our token integration checklist when considering support for new tokens or protocols in the DeFi SDK.

Update: On December 8, 2020, Trail of Bits reviewed fixes by Zerion for the issues presented in this report. See a detailed review of the current status of each issue in Appendix D.

Project Dashboard

Application Summary

Name	DeFi SDK
Version	ddff636
Type	Solidity
Platforms	Ethereum

Engagement Summary

Dates	November 2–November 13, 2020
Method	Whitebox
Consultants Engaged	3
Level of Effort	3 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	0	
Total Low-Severity Issues	0	
Total Informational-Severity Issues	4	■■■■
Total Undetermined-Severity Issues	0	
Total	4	

Category Breakdown

Authentication	1	■
Cryptography	1	■
Undefined Behavior	2	■■
Total	4	

Code Maturity Evaluation

Category Name	Description
Access Controls	**Satisfactory.** Appropriate access controls were in place for performing privileged operations.
Arithmetic	**Satisfactory.** The arithmetic used by the contracts is minimal and any instances that could overflow include safe arithmetic functions.
Assembly Use	**Satisfactory.** The contracts made minimal use of assembly for checking `delegatecall` return values and for the OpenZeppelin implementation of `ecrecover`.
Decentralization	**Strong.** As the system is non-custodial, there is no concern over the owner's ability to remove trapped tokens. The only additional capability of the owner is to add or remove adapters. Additionally, as the contracts are open-source and only act as an interface to existing contracts, a distrusting user could deploy their own fully functional DeFi SDK.
Contract Upgradeability	**Satisfactory.** The core components of the protocol are not upgradeable. New token and protocol adapter contracts can be registered or deregistered with the system after deployment.
Function Composition	**Strong.** Functions and contracts were organized and scoped appropriately.
Front-Running	**Satisfactory.** Although some functionality could have been affected by front-running attacks, the impact was low.
Monitoring	**Satisfactory.** The events produced by the smart contract code were sufficient to monitor on-chain activity.
Specification	**Moderate.** The contract source code included NatSpec comments for all contracts and externally exposed functions, but internal function readability would be improved by comments as well.
Testing & Verification	**Satisfactory.** The core of the SDK has a broad suite of unit tests that execute via Github Actions.

Engagement Goals

The engagement was scoped to provide a security assessment of the `interactive` branch of the [`zeriontech/defi-sdk`] repository at commit [`ddff636`].

Specifically, we sought to answer the following questions:

- Can user approvals be spent unintentionally?
- Are signatures well formed and properly handled?
- What considerations must be made when developing new adapters?

Coverage

We carried out the review through in-depth manual review and static analysis with Slither, our Solidity static analyzer. In addition to the specific areas mentioned below, we reviewed all in-scope contracts for common Solidity issues such as arithmetic overflow, access control issues, improper use of `delegatecall`, and re-entrancy.

`Router`. The `Router` is the main entrypoint for users and is the contract approved by users to transfer tokens that are forwarded to the `Core` to be used with the adapters. We reviewed this contract to ensure these allowances cannot be accessed by malicious third parties. The logical separation between the `Router` and `Core` contracts help minimize the attack surface and will prevent a malicious adapter from accessing the allowances users grant to the DeFi SDK.

`Core`. The `Core`, as its name suggests, is the main logic contract in the DeFi SDK. It executes actions on behalf of users via the adapters. We reviewed this contract with particular attention to issues where an incorrect amount of tokens may be transferred to or from the contract, as well as the overall interaction flow between the `Core` and the underlying adapters.

`TokenAdapterRegistry` and `ProtocolAdapterRegistry`. These contracts track the tokens and protocols that have corresponding adapters registered with the DeFi SDK. We reviewed these contracts to ensure proper access controls are in place with respect to registering and deregistering adapters, and checked that token and protocol metadata was stored properly.

Contracts in the `adapters` and `interactiveAdapters` folders were out of scope for this review, but we reviewed the adapter interfaces to solidify our understanding of the system.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Measure the gas savings from optimizations,** and carefully weigh them against the possibility of an optimization-related bug. TOB-ZRN-001

❑ **Avoid using Solidity features that are not enabled by default.** Carefully weigh the risk of an `ABIEncoderV2` bug on the DeFi SDK. TOB-ZRN-002

❑ **Add the `chainID` opcode to the signature schema.** This will prevent transactions from being replayed across chains in the event of a post-deployment chainsplit. TOB-ZRN-003

❑ **Include the account in the transaction data,** and check that the account extracted from the signature is equal to this account. TOB-ZRN-004

Long Term

❑ **Monitor the development and adoption of Solidity compiler optimizations and `ABIEncoderV2` to assess their maturity.** This will allow you to react more quickly to any new developments with these features. TOB-ZRN-001, TOB-ZRN-002

❑ **When using digital signatures for transactions, ensure that these signatures include sufficient metadata to prevent replays in a wide variety of scenarios.** TOB-ZRN-003

❑ **When using digital signatures for transactions, ensure that these signatures are properly verified.** TOB-ZRN-004

Findings Summary

#	Title	Type	Severity
1	Solidity compiler optimizations can be dangerous	Undefined Behavior	Informational
2	`ABIEncoderV2` may not be production-ready	Undefined Behavior	Informational
3	Lack of `chainID` validation allows signatures to be re-used across forks	Authentication	Informational
4	Lack of address check allows transaction forgery for random accounts	Cryptography	Informational

1. Solidity compiler optimizations can be dangerous

Severity: Informational
Difficulty: High
Type: Undefined Behavior
Finding ID: TOB-ZRN-001
Target: `truffle.js`

Description
DeFi SDK has enabled optional compiler optimizations in Solidity.

There have been several bugs with security implications related to optimizations. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug that led to incorrect bit shift results was patched in Solidity 0.5.6.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe. Moreover, the Common Subexpression Elimination (CSE) optimization procedure is "implemented in a very fragile manner, with manual access to indexes, multiple structures with almost identical behavior, and up to four levels of conditional nesting in the same function." Similar code in other large projects has resulted in bugs.

There are likely latent bugs related to optimization, and/or new bugs that will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the DeFi SDK contracts.

Recommendation
Short term, measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

2. `ABIEncoderV2` may not be production-ready

Severity: Informational
Difficulty: High
Type: Undefined Behavior
Finding ID: TOB-ZRN-002
Target: Throughout

Description
The contracts use the new Solidity ABI encoder: `ABIEncoderV2`. This encoder is no longer considered experimental by the Solidity team but is still disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

Over three percent of all GitHub issues for the Solidity compiler are related to experimental features, with `ABIEncoderV2` constituting the vast majority of these. Several issues and bug reports are still open and unresolved. `ABIEncoderV2` has been associated with over 20 bugs in the past year, and some are so recent they've not yet been included in a Solidity release.

For example, in March 2019 a severe bug was found in the encoder that was introduced in Solidity 0.5.5.

Exploit Scenario
Zerion deploys its DeFI SDK contracts. After the deployment a bug is found in the encoder. As a result, the contracts are broken and no longer function properly.

Recommendation
Short term, avoid using Solidity features that are not enabled by default. Carefully weigh the risk of an `ABIEncoderV2` bug on the DeFi SDK.

Long term, monitor the development and adoption of `ABIEncoderV2` to assess its maturity.

3. Lack of `chainID` validation allows signatures to be re-used across forks

Severity: Informational
Difficulty: High
Type: Authentication
Finding ID: TOB-ZRN-003
Target: `SignatureVerifier.sol`

Description
The `SignatureVerifier` contract implements EIP712 signatures. However, none of the type hashes used by the protocol include the `chainID` as a parameter. In the event of a contentious fork, the Ethereum community may spread across two or more networks. Without explicit `chainID` verification, it may be possible to successfully replay certain signed actions across chains.

Exploit Scenario
Bob has a wallet holding `DAI`. An EIP is included in an upcoming hard fork that splits the community. After the hard fork, a significant user base remains on the old chain. On the new chain, Bob constructs a signed message that will deposit some of his DAI through the DeFi SDK. Alice, operating on both chains, replays the signed message on the old chain and is able to move some of Bob's `DAI` against his intentions.

Recommendation
Short term, add the `chainID` opcode to the signature schema.

Long term, when using digital signatures for transactions, ensure that these signatures include sufficient metadata to prevent replays in a wide variety of scenarios.

4. Lack of address check allows transaction forgery for random accounts

Severity: Informational
Difficulty: Low
Type: Cryptography
Finding ID: TOB-ZRN-004
Target: `contracts/core/Router.sol`

Description

The `Router.sol` contract contains various `execute` functions that are responsible for executing transactions (see Figure 4.1). This function takes as input the transaction data and the ECDSA signature over that data; hashes the input data; and then calls the `getAccountFromSignature` function on this hashed data and the signature. The `getAccountFromSignature` function, in turn, calls `ecrecover`, which returns the account associated with the ECDSA public key used to produce the signature.

```
/**
    * @notice Executes actions and returns tokens to account.
    * @param data TransactionData struct with the following elements:
    *     - actions Array of actions to be executed.
    *     - inputs Array of tokens to be taken from the signer of this data.
    *     - fee Fee struct with fee details.
    *     - requiredOutputs Array of requirements for the returned tokens.
    *     - salt Number that makes this data unique.
    * @param signature EIP712-compatible signature of data.
    * @return Array of AbsoluteTokenAmount structs with the returned tokens.
    */
   function execute(TransactionData memory data, bytes memory signature)
       public
       payable
       returns (AbsoluteTokenAmount[] memory)
   {
       bytes32 hashedData = hashData(data);
       address payable account = getAccountFromSignature(hashedData, signature);

       markHashUsed(hashedData, account);

       return execute(data.actions, data.inputs, data.fee, data.requiredOutputs, account);
   }
```

Figure 4.1: The `execute` *function.*

However, the account returned from `getAccountFromSignature` is never verified. This means that maliciously formed signatures will potentially not be caught by this function (aside from a few edge cases, e.g., when `r` or `s` are zero). For example, given some input transaction data, a malicious user could generate random `r` and `s` values for their signature, and this function will extract a random account from their signature without raising an error. If the account was also included in the transaction data and checked against the account extracted from the signature, this would verify that the signature was formed properly by the account recovered.

Recommendation
Short term, include the account in the transaction data, and check that the account extracted from the signature is equal to this account.

Long term, when using digital signatures for transactions, ensure that these signatures are properly verified.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Testing	Related to test methodology or test coverage
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important

Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. 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.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

General

- **Add NatSpec comments to internal functions.** Comment coverage on external functions is complete, but due to the way functions are broken down in the DeFi SDK, it would improve readability if all internal functions were provided with NatSpec comments as well.
- **Document why the ETH address was chosen to be `0xEee...EeE`.** Though the choice of address to represent ETH transfers is arbitrary and unlikely to collide with an existing address, documenting this will improve code readability and trust in the system.

Router

- **Document how the formula in the `useCHI` was derived.** Magic numbers should always be accompanied by some form of documentation to improve readability and trust in the system.

D. Fix Log

On December 8, 2020, Trail of Bits reviewed fixes for issues identified in this report. The review was performed by one engineer, working from specific pull requests supplied by the Zerion team in the `zeriontech/defi-sdk` repository. The Zerion team addressed two issues reported in the original assessment and accepted the risk of the remaining two issues. We reviewed each of the fixes to help ensure the proposed remediation would be effective. In addition to the fixes, Zerion also submitted a pull request that improved comment coverage of the codebase.

ID	Title	Severity	Status
1	Solidity compiler optimizations can be dangerous	Informational	Risk Accepted
2	`ABIEncoderV2` may not be production-ready	Informational	Risk Accepted
3	Lack of `chainID` validation allows signatures to be re-used across forks	Informational	Fixed
4	Lack of address check allows transaction forgery for random accounts	Informational	Fixed

Detailed Fix Log

Finding 1: Solidity compiler optimizations can be dangerous
Risk Accepted.

Finding 2: `ABIEncoderV2` may not be production-ready
Risk Accepted.

Finding 3: Lack of `chainID` validation allows signatures to be re-used across forks
Fixed. The Zerion team incorporated the `chainID` into the domain separator and added a check to validate it when verifying signatures. 120, 124

Finding 4: Lack of address check allows transaction forgery for random accounts
Fixed. The Zerion team incorporated the address into the signature schema and added a check for it when verifying signatures. 122