OPTIMUM

BLOCKCHAIN SECURITY

Defi Saver

Smart contract
Security Assessment

April, 2024

Table Of Contents

Disclaimer

This report should not be considered as a security guarantee, investment advice, endorsement or disapproval of any specific project or team. The report makes no claim that the code being reviewed is completely free of vulnerabilities, bugs or potential exploits. Additionally, the report does not assess the financial risk of any asset. Therefore, it is not intended for any third party to make any decisions to buy or sell any asset or product based on this report.

It is important to note that ensuring the security of code is an ongoing process that requires multiple measures. Therefore, it is highly recommended that best coding practices, comprehensive testing, internal audits and bug bounty programs be implemented in addition to this report.

It is the responsibility of the project team to ensure that the code being reviewed is functioning as intended, and that the recommendations provided in this report are thoroughly tested before deployment.

Overview Page

Summary

Project name	Defi Saver
URL	https://defisaver.com
Code	https://etherscan.deth.net/address/0xb3fe6f712c8b8c64cd2780ce714a36e7640ddf0f#code https://github.com/defisaver/defisaver-v3-contracts
Commit hash	8126fc0cf2928d13e0241cfcb51508f253701d34
Mitigations commit hash	
Language	Solidity

Contracts Assessed

Contract name	SHA-1
BytesTransientStorage.sol	
LlamaLendBoost.sol	
LlamaLendSwapper.sol	

We were engaged by the Defi Saver team to conduct a review of the *BytesTransientStorage* contract deployed on the Ethereum mainnet at 0xb3fe6f712c8b8c64cd2780ce714a36e7640ddf0f. This contract utilizes the innovative *TLOAD* and *TSTORE* opcodes introduced in the Cancun hard fork. The Defi Saver team sought assurance regarding the security implications of these opcodes and wanted to gain confidence in their usage.

The security assessment was performed by a single researcher over a half-day period. Specifically, our evaluation focused on the integration aspects of the *LlamaLendBoost* and *LlamaLendSwapper* contracts with the *BytesTransientStorage* contract. The remainder of the codebase was not included in this assessment.

Findings Summary

Severity	Found	Resolved	Partially resolved	Acknowledged
Critical	0	0	0	0
High	0	0	0	0
Medium	0	0	0	0
Low	0	0	0	0
Informational	3	0	0	0
Total	3	0	0	0

Classification of issues

Severity	
Critical	Issues that may directly result in loss of funds, and thus require an urgent fix.
High	Issues that may not be directly exploitable, or with a limited impact, are still required to be fixed.
Medium	Issues that are not necessarily security vulnerabilities, that are required to be fixed unless there is a clear reason not to.
Low	Subjective issues with a negligible impact.
Informational	Subjective issues or observations with negligible or no impact.

Findings

Issue #01	***LlamaLendSwapper*: Consider adding authentication checks to callback functions**
Severity	**Informational**
Description	*LlamaLendSwapper* is a smart contract that implements the callback functions that the *LlamaLend* protocol will invoke upon interaction. Although we could not find any concrete issue in the current version of the code, it is recommended to add authentication checks to these functions that will make sure only the *LlamaLendController* can invoke these functions.
Resolution	

Issue #02	***LlamaLendSwapper.takeAutomationFee*: Consider adding an “_” prefix to the function name**
Severity	**Informational**
Location	LlamaLendSwapper.sol#L122
Description	The use of the "_" prefix for function names is a widely recognized practice that enhances code readability for developers and security researchers alike. To align with this convention, we recommend applying the "_" prefix to the *takeAutomationFee* function within the *LlamaLendSwapper* contract. This adjustment will not only contribute to clearer code organization but also promote consistency within the codebase.
Resolution	

Issue #03	**General potential caveats around Transient Storage**
Severity	**Informational**
Description	Transient storage, a recent addition to the Ethereum Virtual Machine (EVM), introduces new functionalities but also comes with potential challenges. The fundamental issue with transient storage is its hybrid nature, straddling between the characteristics of traditional storage and memory—both widely familiar to developers. Unlike memory, which only persists within a single external call, transient storage maintains its state across external calls within the same transaction. However, unlike storage, transient storage does not retain its values between separate transactions. One critical concern is how transient storage affects the composability of external calls within the EVM. When a sequence of external calls is grouped into a single transaction, the outcome might differ from the same sequence spread across multiple transactions. This variation can pose vulnerabilities, especially for smart contracts that engage with external contracts. In the case of *DefiSaver*, where the system interacts extensively with external contracts, the *BytesTransientStorage* component appears **secure** due to the callback functions within *LlamaLendSwapper*. These

	functions are invoked externally only once after *setBytesTransiently* is called, subsequently overwriting the most recent value. Moreover, *getBytesTransiently* solely retrieves the last written value, ensuring predictable behavior.

General Recommendations

- Ensure that state-changing functions emit events that accurately reflect the changes made to the contract state.
- Remove any unused or test code before deploying the smart contract to minimize security risks and improve its performance.
- Consider adding reentrancy guards to state changing functions that contain external calls.
- While dealing with external token contracts, conduct a thorough due diligence before adding external token contracts to the platform. Utilize secure wrapper libraries such as SafeERC20 for added protection.
- While dealing with upgradeable contracts, consider using a plugin to verify implementation contracts are upgrade safe. openzeppelin-upgrades is highly recommended for this.
- Implement a thorough testing process in a testnet before deploying code to mainnet to minimize risks and ensure stability.
- Consider introducing a contingency plan. The plan should include steps to mitigate risks and address potential issues in the event of unforeseen circumstances. It may include the following elements:
    - Regular code audits and security assessments to identify potential vulnerabilities and improve the robustness of the smart contract code.
    - Monitoring the smart contract's deployment and usage, with automated alerts in place to identify and respond to potential incidents quickly.
    - Establishing a clear process for updating and upgrading the smart contract code, in case of a critical bug.
    - Having a plan in place to limit the impact of an attack, such as implementing a fail-safe mechanism or shutting down the smart contract temporarily until the issue can be resolved.

- Implementing a bug bounty program to incentivize security researchers to report any security issues.
- Establishing clear communications with users and stakeholders, including regular updates on the performance of the smart contract and any potential risks.

OPTIMUM

BLOCKCHAIN SECURITY