Three Sigma Labs

Code Audit

Syrup Lending Protocol

Disclaimer

Code Audit

Syrup Lending Protocol

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

Syrup Lending Protocol

Table of Contents

Summary

Code Audit

Syrup Lending Protocol

Summary

Three Sigma Labs audited Syrup in a 2 days engagement. The audit was conducted from 21-05-2024 to 22-05-2024.

Protocol Description

The Syrup platform, built by Maple Labs, enables users permissionless access to secured, institutional lending for the first time. By depositing USDC into the platform, users receive LP tokens (syrupUSDC) and begin earning yield immediately. All of the yield generated by Syrup is sourced from secured loans to the largest institutions in crypto, fully collateralized with digital assets.

Scope

Code Audit

Syrup Lending Protocol

Scope

```
SyrupRouter.sol
SyrupRateProvider.sol
```

Assumptions

The scope of the audit was carefully defined to include the contracts at the lowest level of the inheritance hierarchy, as these are the ones that will be deployed to the mainnet. The libraries used in the implementation of these contracts are internal contracts that have been previously audited and shown to be secure.

Methodology

Code Audit

Syrup Lending Protocol

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and likelihood of the exploit. The following table summarizes the general expected classification according to severity and likelihood; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Likelihood	LOW	MEDIUM	HIGH
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

Syrup Lending Protocol

Project Dashboard

Application Summary

Name	Syrup
Commit	bd74ca9 and 4aab188
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	21 to 22 May, 2024
№ of Auditors	2
Review Time	2 days

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	0	0	0
High	0	0	0
Medium	0	0	0
Low	2	2	0
None	3	1	2

Category Breakdown

Suggestion	1
Documentation	0
Bug	2
Optimization	0
Good Code Practices	2

Code Maturity Evaluation

Code Audit

Syrup Lending Protocol

Code Maturity Evaluation

Code Maturity Evaluation Guidelines

Category	Evaluation
Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system.
Arithmetic	The proper use of mathematical operations and semantics.
Centralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Code Stability	The extent to which the code was altered during the audit.
Upgradeability	The presence of parameterizations of the system that allow modifications after deployment.
Function Composition	The functions are generally small and have clear purposes.
Front-Running	The system's resistance to front-running attacks.
Monitoring	All operations that change the state of the system emit events, making it simple to monitor the state of the system. These events need to be correctly emitted.
Specification	The presence of comprehensive and readable codebase documentation.
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage.

Code Maturity Evaluation Results

Category	Evaluation
Access Controls	Satisfactory. The codebase has a strong access control mechanism.
Arithmetic	Satisfactory. The codebase uses Solidity version >0.8.0 as well as takes the correct measures in rounding the results of arithmetic operations.
Centralization	Weak. Users of the Syrup Router may be DoSed at any time by the permissions admin.
Code Stability	Satisfactory. The code was stable during the audit.
Upgradeability	Moderate. Certain smart contract implementations can be modified after deployment, albeit with proper timelocks and functional upgradeability patterns.
Function Composition	Satisfactory. Certain components are similar, and the codebase would benefit from increased code reuse.
Front-Running	Moderate. Some front-running issues were found with permits, but they are not significant.
Monitoring	Satisfactory. Events are correctly emitted and the Syrup Router keeps track of on chain activity.
Specification	Satisfactory. In-depth and well structured high-level specification as well as codebase documentation.
Testing and Verification	Satisfactory. Extensive test code coverage as well as usage of tools and different test methods.

Findings

Code Audit

Syrup Lending Protocol

Findings

3S-Sy-L01

Attacker may frontrun **`SyrupRouter::depositWithPermit()`** call and use a different **`depositData_`** as it is not signed

Id	3S-Sy-L01
Classification	Low
Severity	Low
Likelihood	Low
Category	Bug
Status	Addressed in #cb70312

Description

SyrupRouter::depositWithPermit() accepts a permit signature and a **`depositData_`** as extra data to be emitted. An attacker may spot the signature and frontrun the transaction, but with a differerent **`depositData_`**, griefing the user.

Recommendation

A simple fix is not possible as the permit function does not accept any extra data. It should be possible to send the transaction through flashbots to mitigate this issue if needed.

3S-Sy-L02

`SyrupRouter::depositWithPermit()` may be DoSed by frontrunning `ERC20::permit()`

Id	3S-Sy-L02
Classification	Low
Severity	Medium
Likelihood	Low
Category	Suggestion
Status	Addressed in #cb70312

Description

SyrupRouter::depositWithPermit() uses the permit functionality of the **`asset`** by calling **`IERC20Like(asset).permit(owner_, address(this), amount_, deadline_, v_, r_, s_);`**. Some griefer may spot the **`SyrupRouter::depositWithPermit()`** transaction, frontrun it and call the **`IERC20Like::permit()`** function directly, spending the nonce of the signature and DoSing the router.

Recommendation

An attacker has no profit from executing so it is not expected to happen, but some measures could still be taken. Some examples are:

1. Get the allowance and only call **`IERC20Like::permit()`** if it is smaller than **`amount_`**.

2. Use **`try/catch`**.

3. Flashbots.

3S-Sy-N01

Signatures in the **`SyrupRouter`** are not fully compatible with EIP712

Id	3S-Sy-N01
Classification	None
Category	Bug
Status	Acknowledged

Description

EIP712 defines the digest as **`keccak256("\x19\x01" || domainSeparator || hashStruct(message))`**.

`domainSeparator` is **`keccak256(abi.encode(TYPE_HASH, _hashedName, _hashedVersion, block.chainid, address(this)))`**.

`TYPE_HASH` is **`keccak256("EIP712Domain(string name,string version,uint256 chainId,address verifyingContract)");`**

`hashStruct(message))` is dependent on the application, **`ERC20Permit`** is a good example. In this case it would be something like:

```
AUTH_TYPEHASH = keccak256("Auth(address lender,uint256 nonce,uint256 bitmap,uint256 deadline)");
bytes32 structHash = keccak256(abi.encode(AUTH_TYPEHASH, msg.sender, nonces[msg.sender]++, bitmap, deadline);
```

And the final digest is **`keccak256(abi.encodePacked("\x19\x01", domainSeparator, structHash)`**.

Recommendation

To be strictly compliant, the structure above should be applied. However, signatures are signed by Maple so it has no impact on users.

3S-Sy-N02

Hardcoded `1e18` in `SyrupRateProvider`

Id	3S-Sy-N02
Classification	None
Category	Good Code Practices
Status	Addressed in 32b7bc3

Description

`SyrupRateProvider` hardcodes the `shares` when calling `IPoolLike(pool).convertToExitAssets(1e18);`.

Recommendation

Consider using a constant state value instead along with a comment for better readability.

3S-Sy-N03

`owner` in **`SyrupRouter`** also requires **`transfer`** permission

Id	3S-Sy-N03
Classification	None
Category	Good Code Practices
Status	Acknowledged

Description

`SyrupRouter::_deposit()` checks for **`owner`** permission to deposit, but due to transferring the shares later in **`ERC20Helper::transfer()`**, it also requires **`P:transfer`** permission.

Recommendation

According to the specifications, **`P:transfer`** and **`P:transferFrom`** should be permissionless, so this is not a problem. However, consider adding a comment in the code.