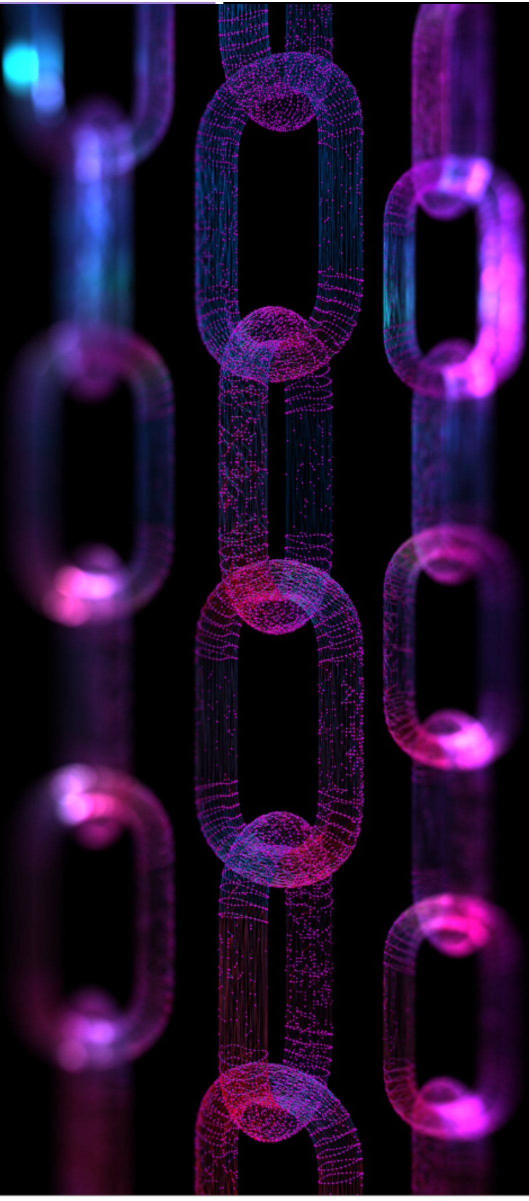


REPORT

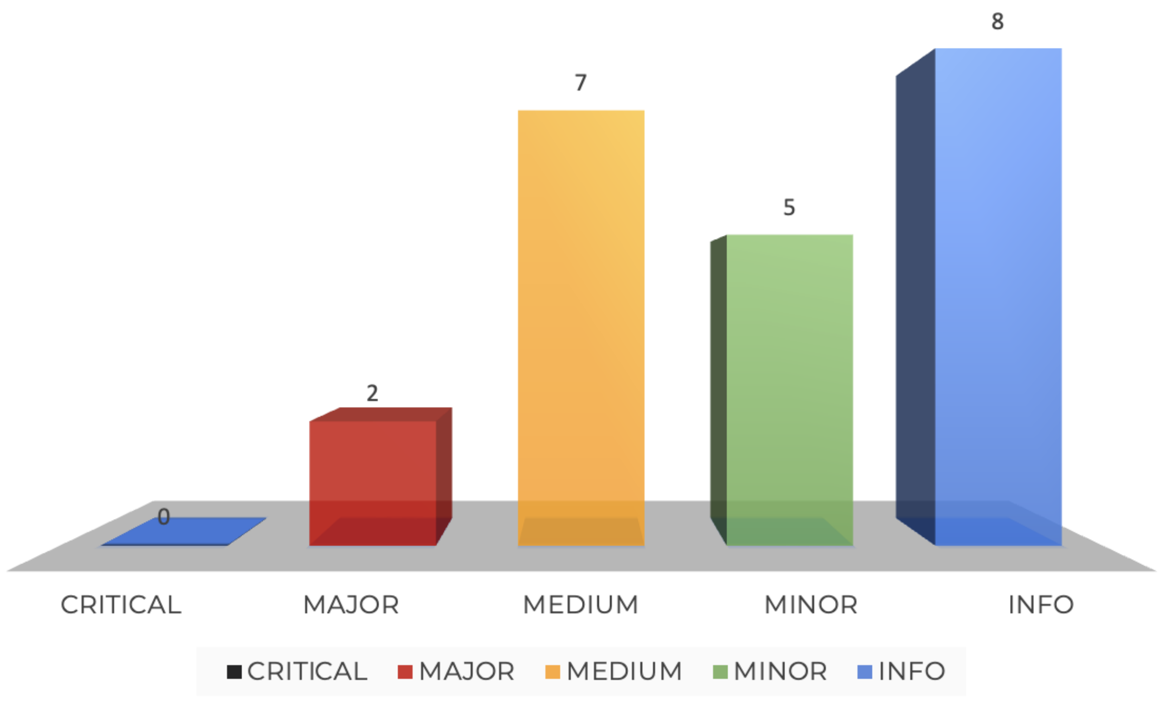
Smart contract security review for Fujidao Labs

Prepared by: Composable Security
Test time period: 2022-10-26 - 2022-11-29
Report date: 2022-12-06
Visit: composable-security.com

1. Contents

2. Executive summary

2.1. Audit results diagram (2022-12-06)



2.2. Audit results (2022-12-06)

Fujidao Labs engaged Composable Security to review the security of Cross-chain Lending Aggregator. Composable Security conducted this assessment over 4 person-week with 3 engineers.

The audit was divided into 2 consecutive rounds. The scope of the tests for both rounds included selected smart contracts (enumerated in Agreed scope of tests section) from repositories presented below.

First round

GitHub repository: https://github.com/Fujicracy/fuji-v2
CommitID: *ab02d2308797577973ac358af8c7aadf973bcec7*

During the first round the following issues were identified:

- No **critical** or **major** risk impact vulnerability was identified.
- **4** vulnerabilities with a **medium** impact on risk were identified. Their potential consequences are:
  - Lack of possibility to decrease *withdrawAllowance* value.

    - Attackers bypass the timelock update and control key parameters of the protocol, including: mappings in *AddrMapping* contract, oracle feeds.
    - The vault cannot get more than one deposit. Additionally, the vault is non compliant with ERC4626, because *maxMint* MUST NOT revert.
    - The users are not able to withdraw all their shares. For the record, they are allowed to withdraw less.
- **5** vulnerabilities with a **minor** impact on risk were identified.
- **4 recommendations** have been proposed that can improve overall security and help implement best practice.
- The most important issues detected concerns compliance with ERC4626 and incorrect conversion of shares and assets.

Second round

GitHub repository: https://github.com/Fujicracy/fuji-v2
CommitID: *bffa427797ad8d6df63671868ee8823574e044ca*

The second round added the following issues:

- **2** vulnerabilities with a **major** impact on risk were identified. Their potential consequences are:
    - Theft of users' assets.
    - Bypass of WithdrawETH action protection.
- **3** vulnerabilities with a **medium** impact on risk were identified. Their potential consequences are:
    - The router contract is temporarily blocked (until a user with a *HOUSE_KEEPER_ROLE* role sweeps Ether or anyone uses the Ether left on the router to deposit).
    - Users are not protected from overpaying for token swaps.
    - Attackers steal accidentally stuck assets.
- **4 recommendations** have been proposed that can improve overall security and help implement best practice.
- There were two potential issues identified that were out of scope of the security review. The *Fujidao labs* team has been informed about them.

Composable Security recommends that *Fujidao Labs* complete the following:

- Address all reported issues.
- Extend unit tests with scenarios that cover detected vulnerabilities where possible.

- Consider whether the detected vulnerabilities may exist in other places (or ongoing projects) that have not been detected during engagement.

3. Project details

3.1. Projects goal

The Composable Security team focused during this audit on the following:

- Perform a tailored threat analysis.
- Ensure that smart contract code is written according to security best practices.
- Identify security issues and potential threats both for *Fujidao Labs* and their users.

The secondary goal is to improve code clarity and optimize code where possible.

3.2. Agreed scope of tests

The subjects of the first test round were selected contracts from the *Fujidao Labs* repository.

First round

GitHub repository: https://github.com/Fujicracy/fuji-v2
CommitID: *ab02d2308797577973ac358af8c7aadf973bcec7*

Files in scope:

```
.
├── abstracts
│   ├── BaseRouter.sol
│   ├── BaseVault.sol
│   ├── PausableVault.sol
│   └── VaultDeployer.sol
├── access
│   ├── CoreRoles.sol
│   └── SystemAccessControl.sol
├── Chief.sol
├── FujiOracle.sol
├── helpers
│   └── AddrMapper.sol
├── providers
│   └── mainnet
│       ├── AaveV2.sol
│       ├── CompoundV2.sol
│       ├── CompoundV3.sol
└── vaults
    ├── borrowing
    │   ├── BorrowingVaultFactory.sol
    │   └── BorrowingVault.sol
    ├── VaultPermissions.sol
    └── yield
        ├──YieldVaultFactory.sol
        └── YieldVault.sol
```

Second round

GitHub repository: https://github.com/Fujicracy/fuji-v2
CommitID: *bffa427797ad8d6df63671868ee8823574e044ca*
Files in scope :

```
.
├── RebalancerManager.sol
├── abstracts
│   ├── BaseFlasher.sol
│   ├── BaseRouter.sol
│   ├── EIP712.sol
├── flashloans
│   └── FlasherAaveV3.sol
├── libraries
│   ├── LibActionBundler.sol
│   ├── LibConnextBundler.sol
├── routers
│   ├── ConnextRouter.sol
│   └── SimpleRouter.sol
└── swappers
    └── UniswapV2Swapper.sol
```

Documentation: Is not public, it was delivered as a Notion article.

3.3. Threat analysis

This section summarizes the potential threats that were identified during initial threat modeling performed before the audit. The tests were focused, but not limited to, finding security issues that could be exploited to achieve these threats.

Potential attackers goals:
- Theft of user's funds.
- Lock users' funds in the contract.
- Making the vault positions undercollateralized.
- Block the contract, so that others cannot use it.

Potential scenarios to achieve the indicated attacker's goals:
- Influence or bypass the business logic of the system.
- Take advantage of arithmetic errors.
- Privilege escalation through incorrect access control to functions or badly written modifiers.
- Existence of known vulnerabilities (e.g., front-running, re-entrancy).
- Design issues.
- Excessive power, too much in relation to the declared one.
- Unintentional loss of the ability to govern the system.
- Poor security against taking over the managing account.
- Private key compromise, rug-pull.
- Withdrawal of more funds than expected.
- Oracle price manipulation.
- Impersonating other users.

- Modifying or executing submitted transactions.

3.4. Testing methodology

Smart contract security review was performed using the following methods:

- Q&A sessions with the *Fujidao Labs* development team to thoroughly understand intentions and assumptions of the project.
- Initial threat modeling to identify key areas and focus on covering the most relevant scenarios based on real threats.
- Automatic tests using *slither*.
- Custom scripts (e.g. unit tests) to verify scenarios from initial threat modeling.
- **Manual review of the code.**

3.5. Disclaimer

Smart contract security review **IS NOT A SECURITY WARRANTY**.

During the tests, the Composable Security team makes every effort to detect any occurring problems and help to address them. However, it is not allowed to treat the report as a security certificate and assume that the project does not contain any vulnerabilities. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we encourage the implementation of security mechanisms at all stages of development and maintenance.

4. Findings overview

ID	Severity	Vulnerability
[R1] ROUND 1		
Fuji_ab02d230_5.1	MEDIUM	Incorrect allowance value used in calculations
Fuji_ab02d230_5.2	MEDIUM	Timelock bypass
Fuji_ab02d230_5.3	MEDIUM	Arithmetic operation leading to Denial of Service
Fuji_ab02d230_5.4	MEDIUM	Inability to withdraw all assets and all shares
Fuji_ab02d230_5.5	MINOR	Incorrect user’s borrow balance
Fuji_ab02d230_5.6	MINOR	Incorrect calculation of deposit balance
Fuji_ab02d230_5.7	MINOR	Lack of decimals verification
Fuji_ab02d230_5.8	MINOR	Out-of-date provider
Fuji_ab02d230_5.9	MINOR	Lack of slippage protection
[R2] ROUND 2		
Fuji_ab02d230_5.10	MAJOR	Insecure implementation of approvals and permits
Fuji_ab02d230_5.11	MAJOR	Invalid WithdrawETH protection

Fuji_ab02d230_5.12	MEDIUM	Router DoS
Fuji_ab02d230_5.13	MEDIUM	Swapper slippageTooHigh mechanism does not work correctly
Fuji_ab02d230_5.14	MEDIUM	Possibility to deposit accidentally stuck tokens

ID	Severity	Recommendation
[R1] ROUND 1		
Fuji_ab02d230_6.1	INFO	Do not set unnecessary roles
Fuji_ab02d230_6.2	INFO	Consider using the latest solidity version
Fuji_ab02d230_6.3	INFO	Improve code clarity
Fuji_ab02d230_6.4	INFO	Optimize gas consumption
[R2] ROUND 2		
Fuji_ab02d230_6.5	INFO	Use current version of Uniswap router
Fuji_ab02d230_6.6	INFO	Lack of support for all non-WETH9 pools
Fuji_ab02d230_6.7	INFO	Update custom libraries
Fuji_ab02d230_6.8	INFO	Improve test coverage

5. Vulnerabilities

5.1. [R1] Incorrect allowance value used in calculations

Severity

MEDIUM

Affected smart contracts

VaultPermissions.sol#L100

Description

The *VaultPermissions* contract distinguishes the quantity of borrow and withdrawal allowances by keeping those into separate mappings. However, the *decreaseWithdrawAllowance* function incorrectly assigns value from *_withdrawAllowance* mapping into the *currentAllowance* variable.

Attack scenario

The malicious scenario might take the following steps in turn:

- User wants to decrease the **borrow** allowance given to another user, to whom the **withdrawal** allowance has not been given at all.
- The *decreaseBorrowAllowance* function reverts, because it cannot meet the required condition (function tries to decrease the **withdrawal** allowance which is not set).

Result of the attack: Lack of possibility to decrease *withdrawAllowance* value.

Recommendation
Change the *decreaseWithdrawAllowance* function, so as to assign the appropriate value into the *currentAllowance* variable. ● VaultPermissions.sol#L100

References

SCSVS G4: Business Logic
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x104-G4-Business-Logic.md

5.2. [R1] Timelock bypass

Severity

MEDIUM

Affected smart contracts

Chief.sol#L53

Description

The address with *DEFAULT_ADMIN_ROLE* role is able to instantly update the timelock contract address and control key parameters of the protocol.

Attack scenario

The attackers require access to an address that has *DEFAULT_ADMIN_ROLE* role and do the following steps:

- Call function *setTimelock* on the *Chief* contract and set the timelock to a malicious contract.
- Call any function on a new, malicious timelock contract that can instantly make changes in the protocol, e.g. update mapping using *setMapping* function.

Result of the attack: Attackers bypass the timelock update and control key parameters of the protocol, including: mappings in *AddrMapping* contract, oracle feeds.

Recommendation
Consider using a timelock mechanism to update the timelock contract address and set the initial timelock via constructor or one-time use function.

References

G5: Access control
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x105-G5-Access-Control.md

5.3. [R1] Arithmetic operation leading to Denial of Service

Severity

MEDIUM

Affected smart contracts

BaseVault.sol#L151

Description

When more than one user deposits to the vault, the *maxMint* function calculates how much shares can be minted based on the maximum amount of assets to be deposited and total supply of the shares.

However, the maximum amount of assets to be deposited is set to the biggest value that *uint256* can get and the multiplication overflows, not allowing to make a second deposit.

Note: Tested using the *CompoundV2* provider.

The Composable Security team has created a PoC for the bug and created a PR to Client's repo.

Attack scenario

The attackers might take the following steps in turn:

- Make the first deposit for any amount greater than 0.
- New deposits by any users are reverted.

Result of the attack: The vault cannot get more than one deposit. Additionally, the vault is non compliant with *ERC4626*, because *maxMint* MUST NOT revert.

Recommendation
• Use the constant value as the return value in the *maxMint* function. • Use the maximum value of 2**128 for *depositCap* to protect from multiplication overflow.

References

G7: Arithmetic
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x107-G7-Arithmetic.md
Added max mint overflow exploit. #112
https://github.com/Fujicracy/fuji-v2/pull/112

5.4. [R1] Inability to withdraw all assets and all shares

Severity

MEDIUM

Affected smart contracts

BaseVault.sol#L161, #L332, #L177

Description

The vault compliant to *ERC4626* uses *maxRedeem* and *maxWithdraw* functions to calculate the maximum shares and deposits to be redeemed and withdrawn, respectively.

However, these functions use the conversion functions that cause the arithmetic errors and when the user tries to withdraw all assets or redeem all shares (returned by *maxWithdraw* and *balanceOf* functions, respectively) the transaction reverts.

This bug happens only when more than one user deposits to the vault and is caused by the fact that when withdrawing all available assets or shares, function *maxRedeem* is called and it does the rounding down (BaseVault.sol#L177) when converting to the redeemable shares and returning an amount smaller by 1 than the actual share balance.

When a user tries to redeem all shares using the *maxRedeem* function, the transaction is not reverted but the user is left with 1 share in the vault.

The severity of this issue has been increased to *Medium* due to the fact that protocol must allow the withdrawal of the amount of assets returned by *maxWithdraw* function.

Note: Tested using the *CompoundV2* provider.

The Composable Security team has created a PoC for the bug and created a PR to Client's repo.

Attack scenario

The following scenario leads to locked shares:

- A user deposits any amount to the vault.
- Another user deposits any amount to the vault.
- None of the users is able to withdraw assets equal to their *maxWithdraw* amount.
- None of the users is able to redeem shares equal to their balance.

Result of the attack: The users are not able to withdraw all their shares. For the record, they are allowed to withdraw less.

Recommendation
● The bug is caused by the rounding in conversion functions. Consider using one type of rounding for all operations. ● Also, add tests for cases when multiple users use vaults.

References

G7: Arithmetic
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x107-G7-Arithmetic.md
Added 3 security bug scenarios. #114
https://github.com/Fujicracy/fuji-v2/pull/114

5.5. [R1] Incorrect user's borrow balance

Severity

MINOR

Affected smart contracts

CompoundV2.sol#L197

Description

The *getBorrowBalance* function returns the borrowed balance for the vault without including the current interests.

Fuji V2 protocol uses this value to calculate the amount of debt assets to be paid back by users and it returns a lower number than the actual one.

The severity of this issue has been decreased because the exploit requires the provider's token to never accrue interests which is very unlikely to happen as many users interact with the token on a daily basis.

Note: Tested using the *CompoundV2* provider.

The Composable Security team has created a PoC for the bug and created a PR to Client's repo.

Attack scenario

The attackers might take the following steps in turn:

- Deposit collateral and take a loan.
- Wait for some time (the longer the better for the attacker).
- Pay back the initial loan, without the borrow rate interests.

Result of the attack: The user is able to get a free loan.

Recommendation
Use the current borrow balance for the user. This can be achieved using the *libcompound* library (see references).

References

libcompound library
https://github.com/transmissions11/libcompound
Added 3 security bug scenarios. #114
https://github.com/Fujicracy/fuji-v2/pull/114

5.6. [R1] Incorrect calculation of deposit balance

Severity

MINOR

Affected smart contracts

BaseVault.sol#L161
CompoundV2.sol#L181
Euler.sol#L93

Description

The amount of total assets in the vault is retrieved with a call to the *getDepositBalance* on the provider contract.

However, the *getDepositBalance* returns a value that is not the same as the user has deposited and transferred to the vault in the first deposit.

Note: Tested using the *CompoundV2* provider.
Note (round 2): Tested using *Euler* provider.

Attack scenario

The faulty scenario might take the following steps in turn:

CompoundV2

- The user deposits 0.5 WETH to the *CompundV2* via *Fuji's* vault.
- The user calls the *totalAssets* function on the vault and it returns a bit less than 0.5 WETH.

Euler

- The user deposits 0.5 WETH (500000000000000000 Wei) to the *Euler* via *Fuji's* vault.
- The user calls the *totalAssets* function on the vault and it returns 499999999999999999 Wei.

Result of the attack: Users are credited less assets than they have deposited.

Recommendation
Fix the calculation formula for the *totalAssets* function.

References

G7: Arithmetic
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x107-G7-Arithmetic.md
Added invalid share verification test. #113
https://github.com/Fujicracy/fuji-v2/pull/113

5.7. [R1] Lack of decimals verification

Severity

MINOR

Affected smart contracts

FujiOracle.sol#L31

Description

The price feed that is added to the oracle is assumed to have 8 decimals, but it is never verified.

Result of the attack: The feed that returns a different value for decimals than 8 could lead to invalid price values (different by orders of magnitudes).

Recommendation
Verify that price feed returns value 8 from decimals function.

References

C3: Oracle
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x200-Components/0x203-C3-Oracle.md

5.8. [R1] Out-of-date provider

Severity

MINOR

Affected smart contracts

BaseVault.sol#L487

Description

The *BaseVault* has the *setActiveProvider* functions, callable by rebalancer, that allows to change the current provider. The function is called during rebalancing.

It is crucial to call this function within one transaction to make the rebalancing process an atomic process. Otherwise, the vault might be used by users after the provider is changed and before the funds are moved, which would result in transactions being reverted (e.g. borrowing from a new provider before it gets collateral).

Attack scenario

The faulty scenario might take the following steps in turn:

- User deposits collateral to the vault.
- Rebalancer changes the active provider (without moving funds).

- Users cannot borrow from a new provider, indirectly, because there is no collateral yet.

Result of the attack: Some operations are blocked for users (e.g. borrowing, withdrawing).

Recommendation
Make the function internal and call it during the rebalancing process.

References

G4: Business logic
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x104-G4-Business-Logic.md

5.9. [R1] Lack of slippage protection

Severity

MINOR

Affected smart contracts

BaseVault.sol

Description

The *EIP-4626* standard is not meant to be used directly by EAOs. The deposit, mint, redeem, withdraw function can be subject to slippage and have no specific protection. Users should be allowed to verify if the received amount of shares or assets is as expected.

Result of the attack: The lack of slippage checks could cause arbitrage opportunities for attackers, hurting the end-users of the vaults.

Recommendation
• Extend the vaults' logic with functions described in *EIP-5143* standard. • Implement the *ERC4626Router* to interact with the vaults through.

References

EIP-5143: Slippage protection for Tokenized Vault

https://eips.ethereum.org/EIPS/eip-5143
ERC4625Router
https://github.com/fei-protocol/ERC4626#erc4626router-and-base

5.10. [R2] Insecure implementation of approvals and permits

Severity

MAJOR

Affected smart contracts

BaseRouter.sol#L78

Description

The *BaseRouter* contract allows bundling multiple actions in one call to execute them in one transaction. Some of these actions (namely Deposit, Payback, Borrow, Withdraw) can abuse the permits and approvals given by the victim to the router.

That approach is not an exploit per se as it is required to execute specific actions. For example, if a user wants to deposit via router, they must approve the router to transfer the assets. On the other hand, if a user wants to deposit and borrow in a bundle, they must approve the router to transfer the assets but also permit the router to borrow on the user's behalf.

However, special care should be taken when handling bundles that include actions using approvals and permits. More information can be found in the Recommendation section. Additionally, the protocol supports unlimited approvals that increases the risk.

It is worth mentioning that there are two cases of approvals:

1. withdrawal/borrow approvals that are in control of *Fujidao Labs* developers, and
2. token (e.g. USDC) approvals that are out of control of *Fujidao Labs* developers.

Attack scenario

Deposit and Payback actions

1. The victim approves a collateral token to the router.
2. The victim calls a bundle with a *Deposit* action.

3. Attacker front-runs the second transaction and sends a call with a *Deposit* or *Payback* action with victim as sender and attacker as receiver.

Withdraw action (approval case)

1. The victim increases withdrawal allowance for the router.
2. The victim wants to call a bundle with a *Withdraw* action and, for example, *WithdrawETH* or *XTransfer* action.
3. Attacker front-runs the second transaction and sends a call with a *Withdraw* action with victim as sender and router as receiver and another action *Deposit* that pulls assets from router and deposits on attacker's behalf.

Withdraw action (permit case)

1. The victim signs a permit to increase withdrawal allowance for the router.
2. The victim calls a bundle with a *PermitWithdraw* action (that permits router), Withdraw action and, for example, *WithdrawETH* or *XTransfer* action, or
   The victim's permit is published.
3. Attacker sends a call with a *PermitWithdraw* action, *Withdraw* action with victim as sender and router as receiver and another action *Deposit* that pulls assets from router and deposits on attacker's behalf.

Borrow action

1. The victim signs a permit to increase borrow allowance for the router.
2. The victim calls a bundle with a *PermitBorrow* action (that permits router), and *Borrow* action,
   or
   The victim's permit is published.
3. Attacker sends a call with a *PermitBorrow* action and *Borrow* action with victim as sender and router as receiver and another action *Deposit* that pulls assets from router and deposits on attacker's behalf.

Result of the attack: Theft of users' assets.

Recommendation
As this vulnerability is a special case of intended feature, it is hard to propose a solution that will not affect the feature itself. The Composable Security team has been discussing it with *Fujidao Labs* team and the current recommendation is to require the receiver to be the owner of tokens (sender).

References

G5: Access control
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x105-G5-Access-Control.md

5.11. [R2] Invalid WithdrawETH protection

Severity

MAJOR

Affected smart contracts

BaseRouter.sol#L212

Description

The execution of *WithdrawETH* action is secured in a way that it checks whether the previous action was *Borrow* or *Withdraw* action and the *WithdrawETH* receiver is the same as the owner of assets from previous action.
There are two issues with this construction:

1. There is a typo in the expression. It should be
   `actions[i-1] != Action.Borrow`
   instead of
   `actions[i] != Action.Borrow`
2. The mentioned mechanism can be bypassed as presented in the Attack Scenario.

Attack scenario

1. The attacker prepares the following bundle:
   a. *PermitWithdraw* of leaked victim's permit for a vault with WETH as collateral,
   b. Withdraw of victim's collateral to router,
   c. *Withdraw* of attacker's collateral to router (this action bypasses the protection mechanism),
   d. *WithdrawETH* of all Ether gathered on the router.

Result of the attack: Bypass of *WithdrawETH* action protection.

Recommendation
This vulnerability can be avoided by removing the mentioned protection mechanism and implementing the one from "Insecure implementation of approvals and permits" as the root cause is the same.

References

G5: Access control
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x105-G5-Access-Control.md

5.12. [R2] Router DoS

Severity

MEDIUM

Affected smart contracts

BaseRouter.sol#L280

Description

The *_checkNoNativeBalance* function is used to make sure that there are no Ethers left on the router after a bundle is executed. It is used to protect users from executing invalid bundles that do not withdraw Ethers from the router and leave it locked on the router. The function reverts whenever the Ether balance is greater than 0.

On one hand, the *fallback* function (BaseRouter.sol#L299) reverts on any call and *receive* function (BaseRouter.sol#L290) is allowed only for WETH contract. On the other hand, anyone can increase the balance of the router contract with a self-destruct function or using the xTransfer with a call that transfers any amount of Ether and reverts.

Note: The transferred amount can be swept by users with a *HOUSE_KEEPER_ROLE* role or used by anyone to deposit, however it becomes economically unprofitable when the transferred amount is 1 Wei.

Attack scenario

The malicious scenario might take the following steps in turn:

- The attacker self-destructs a contract that has 1 Wei and which is transferred to the router contract,
  or
  The attacker makes a xTransfer of 1 Wei with a call that reverts on the destination chain.
- A user is calling a bundle call which reverts until the Ether is not transferred out.

Result of the attack: The router contract is temporarily blocked (until a user with a *HOUSE_KEEPER_ROLE* role sweeps Ether or anyone uses the Ether left on the router to deposit).

Recommendation
In order to keep the protection mechanism that does not allow leaving new Ether on the router contract we recommend to check the balance before and after the bundle is executed and revert if the balances differ.

References

G8: Denial of service
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x108-G8-Denial-of-Service.md

5.13. [R2] Swapper *slippageTooHigh* mechanism does not work correctly

Severity

MEDIUM

Affected smart contracts

UniswapV2Swapper.sol#L81-L83
LibActionBundler.sol#L65

Description

Assumptions behind this mechanism is to protect users from overpaying for token swaps. The *minSweepOut* parameter is a minimum required amount that users expect to get back from the initial *amountIn*, in order for the swap to succeed.

For example:

- the user provides 1ETH and wants to swap for only 500 USDC,
- they want min 0.49 ETH back (1ETH = 1000$), however the leftover is 0.47ETH,
- the swap should revert.

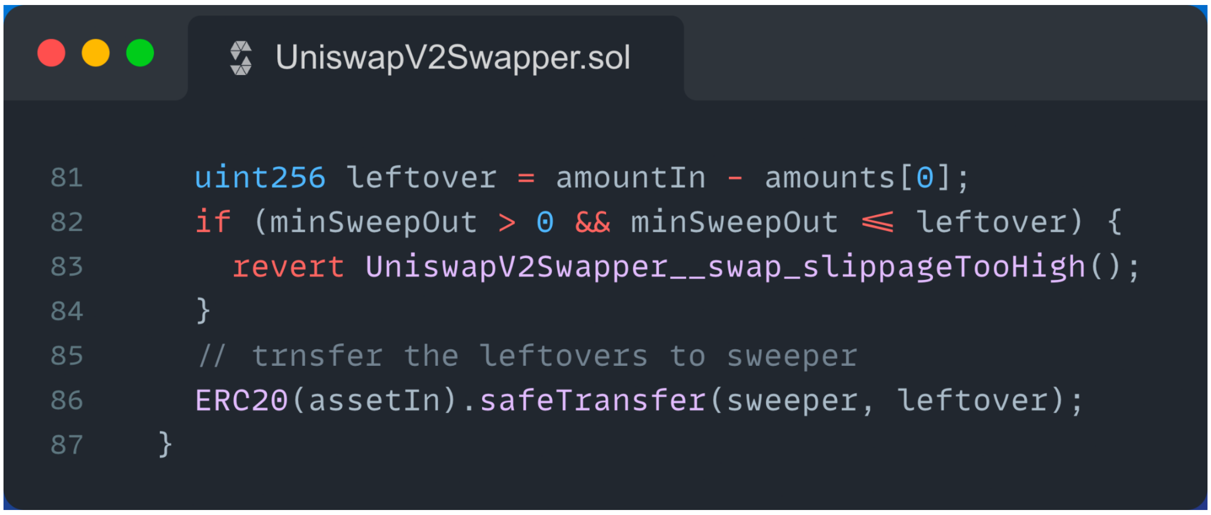
There are two issues with this mechanism:

- typo in condition,

- improper use of the mechanism in the library.

Typo in condition

Current mechanism would accept unwanted cases where leftovers are lower than *minSweepOut (*[UniswapV2Swapper.sol#L81-L83]).

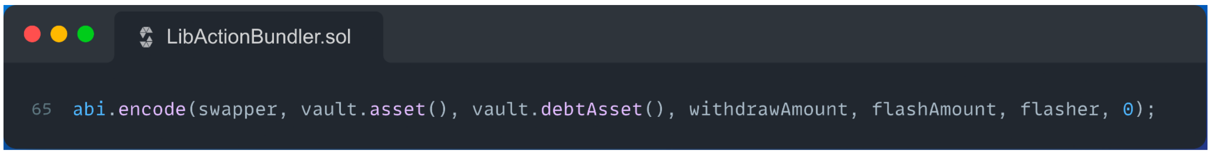


UniswapV2Swapper.sol

```
81        uint256 leftover = amountIn - amounts[0];
82        if (minSweepOut > 0 && minSweepOut <= leftover) {
83          revert UniswapV2Swapper__swap_slippageTooHigh();
84        }
85        // trnsfer the leftovers to sweeper
86        ERC20(assetIn).safeTransfer(sweeper, leftover);
87      }
```

Improper use of the mechanism

The *minSweepOut* it set to 0 in [LibActionBundler.sol#L65] which means that the protection mechanism is not used.



LibActionBundler.sol

```
65  abi.encode(swapper, vault.asset(), vault.debtAsset(), withdrawAmount, flashAmount, flasher, 0);
```

Note: The *Fujidao Labs* team plans to remove the libraries.

Attack scenario

The faulty scenario might look like this:

- Call *Fuji* router to swap tokens with *minSweepOut* defined as 50.
- The current exchange price has become high and leftovers are only 1.
- The transaction is not reverted because the second condition was not met (`minSweepOut <= leftover`).
- The user performed the swap regardless of poor market conditions.

Result of the attack: Users are not protected from overpaying for token swaps.

Recommendation
● Fix the condition in UniswapV2Swapper to the following: **it should be if (minSweepOut > 0 && minSweepOut > leftover)**   ● Consider percentage as a unit to handle tokens with different numbers of zeros in a uniform way.   ● In the *LibActionBundler* contract allow the *minSweepOut* parameter to be passed, rather than hard-code an option that disables the security mechanism.

References

G4: Business logic
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x104-G4-Business-Logic.md

5.14. [R2] Possibility to deposit accidentally stuck tokens

Severity

MEDIUM

Affected smart contracts

BaseRouter.sol#L85, #L242

Description

The *sweepToken* function (BaseRouter.sol#L58) allows users with the *HOUSE_KEEPER_ROLE* role to transfer accidentally stuck ERC-20 tokens (due to the failed cross chain calls). A malicious user executing a Deposit action can deposit those assets on his behalf to the vault.

Similarly, *sweepETH* function (BaseRouter.sol#L67) allows users with the *HOUSE_KEEPER_ROLE* role to transfer accidentally stuck Ether. A malicious user can withdraw a small sum of their assets using Withdraw action and all Ethers using a consecutive *WithdrawETH* action.

Attack scenario

The sweepToken function

The malicious scenario might take the following steps in turn:

- Call function *xBundle* on the *BaseRouter* contract with the parameters required for Deposit action. Set the *receiver value* as controlled address and *sender* value as a *BaseRouter* address.
- The *_safePullTokenFrom* function checks if the *sender* address is set to *address(this)* - if it is true, function pass.
- The vault contract receives a deposit on behalf of the controlled address.

The sweepETH function

The malicious scenario might take the following steps in turn:

- Call function *xBundle* on the *BaseRouter* contract with the parameters required for two actions: Withdraw and WithdrawETH. The amount in the first action is small (attacker must have some assets in the vault) and the amount in second action adds the amount of stuck ETH.
- The router withdraws attacker's assets legitimately and next withdraws Ether including both, attacker's assets and stuck Ethers.

Result of the attack: Attackers steal accidentally stuck assets.

Recommendation
Update the existing protection mechanism that checks the balances of tokens and Ether before and after executing a bundle. It should not only check whether the amounts after execution are smaller or equal, but it should require amounts to be strictly equal.

References

G5: Access control
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x105-G5-Access-Control.md

6. Recommendations

6.1. [R1] Do not set unnecessary roles

Severity

INFO

Description

The Chief contract sets *PAUSER* and *UNPAUSER* roles for itself while it contains external functions that require these roles. The deployer will still need to set the roles for external pausers and unpausers as Chief cannot call these functions itself.

The vaults cannot be paused and unpaused until the deployer assigns the roles to actual addresses.

Recommendation
Do not set roles for the Chief itself. Assign roles in the deployment script to the actual addresses. • Chief.sol#L40 • Chief.sol#L41

References

G5: Access control
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x105-G5-Access-Control.md
SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md

6.2. [R1, R2] Consider using the latest solidity version

Severity

INFO

Description:

In accordance with the best security practices, it is recommended to use the latest stable versions of major Solidity releases. Older versions may contain bugs that have been discovered and fixed in newer versions.

Moreover, it is worth remembering that the version should be clearly specified so that all tests and compilations are performed with the same version.

Recommendation
Use a specific version of Solidity compiler (latest stable): `pragma solidity 0.8.16;`

References:

SCSVS V1: Architecture, design and threat modeling
https://github.com/securing/SCSVS/blob/master/1.2/0x10-V1-Architecture-Design-Threat-modelling.md
Floating pragma SWC-103
https://swcregistry.io/docs/SWC-103

6.3. [R1,R2] Improve code clarity

Severity

INFO

Description

Due to the composable nature of the smart contract ecosystem, it is recommended to keep the code clear and highly understandable. No leftovers or unused code snippets should be left in the code. Describe your assumptions, reuse code from trustworthy sources when possible and do not complicate the implementation.

This will not only allow you to avoid unwanted issues, but also maximize the effectiveness of audits and peer reviews.

Recommendation

- Consider moving not implemented and reverting function from *YieldVault* contract to a separate interface for borrowing vaults.
  - YieldVault.sol
- Remove unused *CoreRoles* inheritance.

-
    - SystemAccessControl.sol#L14
- Rename custom error name to better suit its purpose
    - Change the custom error name from *Chief__deployVault_missingRole* to *Chief__deployVault_missingRoleOrVaultFactoryNotOpen*
        - Chief.sol#L30
- Fix typos in comments
    - UniswapV2Swapper#L7
    - UniswapV2Swapper#L40
    - UniswapV2Swapper#L84

References

SCSVS G11: Code clarity
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x111-G11-Code-Clarity.md
Solidity Style Guide 0.8.16
https://docs.soliditylang.org/en/v0.8.16/style-guide.html

6.4. [R1] Optimize gas consumption

Severity

INFO

Description

When designing projects based on smart contracts, it is particularly important to pay attention to the gas used. Consumption should be optimized as much as possible. This has a direct impact on costs for users and, in exceptional cases, on security.

A helpful tool for estimating gas consumption and tracking optimization quality can be gas reporters:

- https://book.getfoundry.sh/forge/gas-reports
- https://www.npmjs.com/package/hardhat-gas-reporter

Recommendation
Following increments in for-loop can be unchecked : • FujiOracle.sol#L30 • YieldVaultFactory.sol#L26

References

Solidity gas optimization tips by devanshbatham:
https://github.com/devanshbatham/Solidity-Gas-Optimization-Tips
Solidity gas optimization tips by Mudit Gupta:
https://mudit.blog/solidity-gas-optimization-tips/
Under-Optimized Smart Contracts Devour Your Money research paper:
https://arxiv.org/pdf/1703.03994.pdf

6.5. [R2] Use current version of the Uniswap router

Severity

INFO

Description

In the Uniswap documentation, the used *UniswapV2Router01* is indicated as containing a vulnerability and unusable:

"*UniswapV2Router01 should not be used any longer, because of the discovery of a low severity bug and the fact that some methods do not work with tokens that take fees on transfer. The current recommendation is to use UniswapV2Router02.*"

The low severity bug mentioned is in the *getAmountIn* function and is related to tokens that contain fees. This function is not currently used in the project and no risks related to this vulnerability have been detected during testing.

Recommendation
Consider switching to *UniswapV2Router02* to follow the best security practices and use the latest supported version.

References

Uniswap documentation:
https://docs.uniswap.org/contracts/v2/reference/smart-contracts/router-01
SCSVS G1-Architecture-Design-Threat-Modeling
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x101-G1-Architecture-Design-Threat-Modeling.md

6.6. [R2] Lack of support for all non-WETH9 pools

Severity

INFO

Description

Cases where two different tokens have their own pool and can be exchanged directly (without WETH9) are currently not supported by the protocol.

Mechanism in UniswapV2Swapper#L55-65 either allows the use of pools in which one of the tokens is WETH9 or finds a connection between two tokens with WETH9 as the intermediary one.

Recommendation
Implement a mechanism that allows to detect pools that swap two tokens directly, without using WETH9, if such a pool exists.

References

SCSVS G1-Architecture-Design-Threat-Modeling
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x101-G1-Architecture-Design-Threat-Modeling.md

6.7. [R2] Update custom libraries

Severity

INFO

Description

The *LibConnextBundler* and *LibActionBundler* libraries contain the source code that is not compliant with the current version of Connext routers implementations.

The *bridgeWithCall* function in LibConnextBundler.sol#L26 uses an incorrect function name for Connext contract - *inboundXCall*.

There is incorrect number of parameters in the *closeWithFlashloan* function in LibActionBundler.sol#L65. Should be 8 instead of 7.

Recommendation
Either fix the function name and parameters or remove the libraries and create bundles on the frontend.

References

SCSVS G1-Architecture-Design-Threat-Modeling
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x101-G1-Architecture-Design-Threat-Modeling.md

6.8. [R2] Improve test coverage

Severity

INFO

Description

The solidity code should be covered with unit tests to ensure that it is correct and reliable. Test coverage should be 100% to ensure that all lines of code are tested and that all scenarios, including both positive and negative ones, are considered. This will help ensure that the code will work as intended in all situations.

Recommendation
Increase test coverage by adding new test cases.

References

G12: Test Coverage
https://github.com/ComposableSecurity/SCSVS/blob/prerelease/SCSVSv2/2.0/0x100-General/0x112-G12-Test-Coverage.md

7. Impact on risk classification

Risk classification is based on the one developed by OWASP, however it has been adapted to the immutable and transparent code nature of smart contracts. The Web3 ecosystem forgives much less mistakes than in the case of traditional applications, the servers of which can be covered by many layers of security.

Therefore, the classification is more strict and indicates higher priorities for paying attention to security.

		Overall risk severity		
Impact on risk	HIGH	CRITICAL	MAJOR	MEDIUM
	MEDIUM	MEDIUM	MEDIUM	MINOR
	LOW	MINOR	MINOR	INFO
		LOW	MEDIUM	HIGH
		Exploitation conditions		

OWASP Risk Rating methodology:
https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

8. Long-term best practices

8.1. Use automated tools to scan your code regularly

It's a good idea to incorporate automated tools (e.g. slither) into the code writing process. This will allow basic security issues to be detected and addressed at a very early stage.

8.2. Perform threat modeling

Before implementing or introducing changes to smart contracts, perform threat modeling and think with your team about what can go wrong. Set potential targets of the attacker and possible ways to achieve them, keep it in mind during implementation to prevent bad design decisions.

8.3. Use Smart Contract Security Verification Standard

Use proven standards to maintain a high level of security for your contracts. Treat individual categories as checklists to verify the security of individual components. Expand your unit tests with selected checks from the list to be sure when introducing changes that they did not affect the security of the project.

8.4. Discuss audit reports and learn from them

The best guarantee of security is the constant development of team knowledge. To use the audit as effectively as possible, make sure that everyone in the team understands the mistakes made. Consider whether the detected vulnerabilities may exist in other places, audits always have a limited time and the developers know the code best.

8.5. Monitor your and similar contracts

Use the tools available on the market to monitor key contracts (e.g. the ones where user's tokens are kept). If you have used code from another project, monitor their contracts as well and introduce procedures to capture information about detected vulnerabilities in their code.

Contact

Damian Rusinek
Smart Contract Security Auditor
@drdr_zz
damian.rusinek@composable-security.com

Paweł Kuryłowicz
Smart Contract Security Auditor
@wh01s7
pawel.kurylowicz@composable-security.com