www.securing.biz

Smart Contracts Security Review for FujiDAO Labs

Report

document version: 1.0

author: Pawel Kurylowicz (SecuRing)

test time period: 2021-06-23 – 2021-07-14

report date: 2021-07-16

TABLE OF CONTENTS

1. EXECUTIVE SUMMARY

1.1. Summary of test results

- No vulnerability with **critical** impact on risk was found during the tests.
- There is 1 vulnerability with **high** impact on risk found:
  - Invalid slippage verification (4.1). The team was informed about the issue on the day it was found.
- There are 2 vulnerabilities with **medium** impact on risk found.
  - The first one is related to excessively powerful *Owner* address (4.2).
  - The second cause denial of service by pausing Vault forever (4.3).
- There are 6 vulnerabilities with **low** impact on risk found. Their potential consequences are:
  - Potential loss of control over contracts by the *Owner* (4.4),
  - Temporal block of deposits (4.5),
  - Victim's tokens transfer using the previous allowance as well as the new allowance (4.6),
  - Lack of possibility to unlock transfers of *FujiERC1155* tokens (4.7),
  - Non-compliant *setURI* function (4.8),
  - Continuing execution of the flow that should be reverted and loss of gas (4.9).
- Additionally, 10 recommendations have been proposed that do not have any direct risk impact. However, it is suggested to implement them due to good security practices.
  - Most of the recommendations apply to code clarity, it is worth paying attention to these issues and significantly improve the readability of the code.
- Many of the SCSVS checks have not been covered. It is worth taking a look at them and working on the quality of the created solution.
- The development team was very helpful in carrying out the tests. Communication ran smoothly and had a significant impact on the end result of tests.

1.2. Compliance with Smart Contract Security Verification Standard

Smart Contracts:	**Contracts:** *AlphaWhitelist* *Controller* *Fliquidator* *FujiAdmin* *FujiMapping* *IAlphaWhiteList* *IFujiAdmin*

	Flashloans: *AaveFlashLoans* *CreamFlashLoans* *DyDxFlashLoans* *Flasher* *LibFlashLoan* **FujiERC1155:** *FujiBaseERC1155* *FujiERC1155* *IFujiERC1155* **Libraries:** *Errors* *LibUniERC20* *LibUniversalERC20* *LibVault* *MathUtils* *WadRayMath* **Providers:** *IProvider* *ProviderAave* *ProviderCompound* *ProviderDYDX* *ProviderIronBank* *ProviderLQTY* **Vaults:** *IVault* *VaultBase* *VaultETHDAI* *VaultETHUSDC* *VaultETHUSDT* *VaultHarvester*	
Security Auditors:	Damian Rusinek, Pawel Kurylowicz	
Verification date:	16.07.2021r.	
Passed:	71	
Failed:	27	
Not applicable:	16	
Total score:	72%	71/98

The categories in which some of the requirements has not been met are:

- V1: Architecture, Design and Threat Modelling
- V2: Access Control
- V7: Gas Usage & Limitations
- V8: Business Logic
- V9: Denial of Service
- V10: Token
- V11: Code Clarity
- V12: Test Coverage

- V13: Known Attacks
- V14: Decentralized Finance

Some of the failed checks currently have a high impact on the security of the system. Failure to meet certain checks results in multiple vulnerabilities, including those with a critical and high risk impact on the system.

The results of SCSVS compliance verification are attached to the report in the *SCSVS_Fuji_20210716.xlsx* file.

2. PROJECT OVERVIEW

2.1. Project description

The analyzed system is a borrowing aggregator. It aims to optimize loan expenses by monitoring borrow markets and automatically refinance the whole pool of debt to use the better rate. It distinguishes roles such as *FujiAdmin*, *Flashbot* and *User*:

- ***FujiAdmin*** - the most privileged role responsible for managing the system components, setting the liquidation bonus and creating vaults.
- ***Flashbot*** - supports flashloans required for liquidation and transfer of credit to another provider.
- ***User*** - entrusts funds as collateral to the *Vaults* and borrow assets corresponding to the *Vault* pools. *User* can also liquidate positions both their own and other users.

An example usage scenario:

1. *User* deposits collateral in ETH through *deposit* function in Vault (i.e. *VaultETHDAI*) contract.
2. *User* borrows DAI through *borrow* function in Vault (i.e. VaultETHDAI) contract. Up to ~75% of their collateral.
3. *User* uses borrowed DAI to trade without selling their ETH as long as position is not liquidated.
4. *Flashbot* checks where the most favorable credit terms are for the *User* and automatically transfers the credit to the provider with better terms.

2.2. Target in scope

The objects being analysed were selected smart contracts accessible in the following:

- GitHub repository:
  https://github.com/Fujicracy/fuji/tree/alpha/packages/hardhat/contracts
- Commit ID: *cab79a820be68f11ec00728ead74cbb9aa370692*
- Documentation: https://docs.fujidao.org/

The contracts reviewed were the following:

- Contracts:
  - *AlphaWhitelist* – used for controlling user limit number and *ethCapValue* in alpha version of the system.
  - *Controller* - identifies how much debt position it has to move to a new provider given certain parameters, and calls the *Flasher* to execute the switch.
  - *Fliquidator* – contains a logic of possible liquidation options.
  - *FujiAdmin* – contains system management functions.

   - *FujiMapping* – contains mapping management functions such as setting new addresses mapping and setting *URI* for different token types.
   - *IAlphaWhiteList* – interface to *AlphaWhiteList* contract. It contains a set of functions necessary for the proper function of the system.
   - *IFujiAdmin* - interface to *FujiAdmin* contract. It contains a set of functions necessary for the proper function of the system.
- Flashloans:
   - *AaveFlashLoans* – contains interfaces *IFlashLoanReceiver* and *ILendingPool* to handle flash loans through Aave protocol.
   - *CreamFlashLoans* – contains interfaces *IFlashLoanReceiver* and *ICTokenFlashloan* to handle flash loans through Cream protocol.
   - *DyDxFlashLoans* – contains interfaces and *DyDxFlashloanBase* contract to handle flash loans through *dYdX* protocol.
   - *Flasher* - uses *SafeMath* and *UniERC20* to handle all the parameters that are needed to execute the protocol switch.
   - *LibFlashLoan* – library which defines struct of params to be passed between functions executing flash loan logic.
- FujiERC1155:
   - *FujiBaseERC1155* – implementation of the *Base ERC1155* multi-token standard functions for *Fuji* protocol control of *User* collaterals and borrow debt positions.
   - *FujiERC1155*(*F1155Manager*, *FujiERC1155*) – implementation of the management functions for *FujiBaseERC1155* and a set of custom functions required for the proper function of the system.
   - *IFujiERC1155* – interface for *FujiERC1155*. It contains a set of functions necessary for the proper function of the system.
- Libraries:
   - *Errors* – defines the error messages emitted by the different contracts of the Fuji protocol.
   - *LibUniERC20* – old library used for transferring various tokens.
   - *LibUniversalERC20* – newer implementation of the *LibUniERC20* library for transferring various tokens. Uses *call* instead of *send* underneath.
   - *LibVault* – defines structs of values for multiplying *Factors*.
   - *MathUtils* – contract with set of functions required to calculate the interest.
   - *WadRayMath* – provides *mul* and *div* function for decimal numbers with 18 and 27 digits precision.
- Providers:
   - *IProvider* – interface for providers. It contains a set of functions necessary for the proper function of the system.
   - *ProviderAave* – contract containing Aave-compliant implementations of the functions from the *iProvider* interface.

  - *ProviderCompound* – contract containing Compound-compliant implementations of the functions from the *iProvider* interface.
  - *ProviderDYDX* – contract containing dYdX-compliant implementations of the functions from the *iProvider* interface.
  - *ProviderIronBank* –contract containing IronBank-compliant implementations of the functions from the *iProvider* interface.
  - *ProviderLQTY* – contract containing LQTY-compliant implementations of the functions from the *iProvider* interface.
- Vaults:
  - *IVault* - interface for *Vaults*. It contains a set of functions necessary for the proper function of the system.
  - *VaultBase* – *Vault* base contract used as contract from which *Vaults* with specified asset pairs can inherit.
  - *VaultETHDAI* – *Vault* with specified asset pair as ETH/DAI. Used for housing the collateral (in ETH) which allows anyone to borrow DAI.
  - *VaultETHUSDC*– *Vault* with specified asset pair as ETH/USDC. Used for housing the collateral (in ETH) which allows anyone to borrow USDC.
  - *VaultETHUSDT*– *Vault* with specified asset pair as ETH/USDT. Used for housing the collateral (in ETH) which allows anyone to borrow USDT.
  - *VaultHarvester* – Called by the *Vault* to harvest farmed tokens at base layer protocols. <u>The full functionality of the contract has not been implemented yet.</u>

2.3. Threat analysis

This section summarized the potential threats that were identified during initial threat modeling. The audit was mainly focused on, but not limited to, finding security issues that be exploited to achieve these threats.

The key threats were identified from particular perspectives as follows:

1. General (apply to all of the roles mentioned)
   - Bypassing the business logic of the system.
   - Theft of users' funds.
   - Errors in arithmetic operation.
   - Possibility to block the contract.
   - Possibility to lock users' funds in the contract.
   - Incorrect access control for roles used by the contracts.
   - Existing of known vulnerabilities (e.g. front-running, re-entrancy, overflows).
   - Cause denial of service.
   - Problems resulting from the solution architecture.
2. *FujiAdmin*

- Too much power in relation to the declared one.
- Unintentional loss of the ability to governance the system.
- Malicious influence on fee and users' positions.
- Front-running users to set higher fees than they expect.
- Potential rug pulls.
- Theft of entrusted funds.

3. *Flashbot*
    - Too much power in relation to the declared one.
    - Business logic abuse.
    - Malicious liquidation strategies.
4. *User*
    - Theft of users' funds.
    - Withdrawal of more funds than expected by users.
    - Earning a higher reward than according to the system's assumptions.
    - Oracle price manipulation.
    - Economic attacks through large flash loans and flash swaps.
    - Liquidation without repaying the debt.
    - Withdrawing higher collateral than the deposited one.
    - Taking multiple loans with the same collateral.
    - Act against the protocol.

2.4. Testing overview

The security review of the *FujiDAO Labs* smart contracts was meant to verify whether the proper security mechanisms were in place to prevent users from abusing the contracts' business logic and to detect the vulnerabilities which could cause financial losses to the client or its customers.

Security tests were performed using the following methods:

- Source code review,
- Automated tests through various tools (static analysis),
- Custom scripts (e.g. unit tests) for test scenarios based on key threats,
- Verification of compliance with Smart Contracts Security Verification Standard (SCSVS) in a form of code review,
- Q&A sessions with the client's representatives which allowed to gain knowledge about the project and the technical details behind the platform.

2.5. Basic information

Testing team	Damian Rusinek Paweł Kuryłowicz

Testing time period	2021-06-23 – 2021-07-15
Report date	2021-07-14 – 2021-07-16
Version	1.0

2.6. Disclaimer

The security review described in this report is not a security warranty.
It does not guarantee the security or correctness of reviewed smart contracts. Securing smart contract platforms is a multi-stage process, starting from threat modeling, through development based on best security practices, security reviews and formal verification, ending with constant monitoring and incident response.

Therefore, we strongly recommend implementing security mechanisms at all stages of development and maintenance.

3. SUMMARY OF IDENTIFIED VULNERABILITIES

3.1. Risk classification

Vulnerabilities	
Risk impact	**Description**
Critical	Vulnerabilities that affect the security of the entire system, all users or can lead to a significant financial loss (e.g. tokens). It is recommended to take immediate mitigating actions or limit the possibility of vulnerability exploitation.
High	Vulnerabilities that can lead to escalation of privileges, a denial of service or significant business logic abuse. It is recommended to take mitigating actions as soon as possible.
Medium	Vulnerabilities that affect the security of more than one user or the system, but might require specific privileges, or custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical and high risk impact.
Low	Vulnerabilities that affect the security of individual users or require strong custom prerequisites. The mitigating actions should be taken after eliminating the vulnerabilities with critical, high, and medium risk impact.
Recommendations	
Methods of increasing the security of the system by implementing good security practices or eliminating weaknesses. No direct risk impact has been identified. The decision whether to take mitigating actions should be made by the client.	

3.2. Identified vulnerabilities

Vulnerability	Risk impact
4.1 Invalid slippage verification	**High**
4.2 Excessively powerful Owner address	**Medium**
4.3 Denial of service	**Medium**
4.4 Instant ownership transfer	**Low**
4.5 Unauthorized block of deposits	**Low**
4.6 Front-runnable approve function	**Low**
4.7 FujiERC1155 transfers locked forever	**Low**
4.8 Function non-compliant with ERC1155	**Low**
4.9 Function does not check the *return* value	**Low**

Recommendations
5.1 Use specific Solidity compiler version
5.2 Extend the list of abuser stories in unit tests
5.3 Do not duplicate the code of modifiers and follow the code clarity rules
5.4 Define variables explicitly
5.5 Remove deprecated functions
5.6 Correct comments
5.7 Make functions external to optimise gas costs
5.8 Improve code clarity
5.9 Use constant for hardcoded addresses
5.10 Consider mocking the external contracts

4. VULNERABILITIES

4.1. Invalid slippage verification

Risk impact	High	SCSVS V14
Vulnerable contracts	*Fliquidator*	
Exploitation conditions	Existing positions to be liquidated.	
Exploitation results	Making liquidator doing swap on imbalanced pool with invalid swap rate and stealing some collateral.	
Remediation	Use function from Uniswap V3 to get the price in a defined time or use decentralized price oracle.	

Vulnerability description:

The *Fliquidator* contract computes how much collateral needs to be swapped (Fliquidator.sol#L168) and stores it in *globalCollateralInPlay* variable, which is later used in the *_swap* function (Fliquidator.sol#L178). The swapped collateral is computed using the *getAmountsIn* function from Uniswap pool

The attacker can imbalance the pool (perform a big swap before calling the *batchLiquidate function*) and make the liquidator contract calculate collateral using invalid swap rate. This allows the attacker to steal some collateral from the vault.

Test case:

Below is the scenario of the attack:

1. [ATTACKER] Takes a flashloan to imbalance the Uniswap pool.
2. [ATTACKER] Performs the huge (unfavourable) swap to cause imbalance in the Uniswap pool and make unfavourable conditions for buying borrow asset.
3. [ATTACKER] Uses the batchLiquidate function to liquidate large debt.
4. [FUJI] Calculates *globalCollateralInPlay* based on unfavourable exchange rate (selling more ETH for borrow asset).
5. [FUJI] Performs *_swap* converting more ETH collateral than should (because of very unfavourable conditions and to high value of *globalCollateralInPlay*).
6. [FUJI] Transfer reward to Liquidator.
7. [ATTACKER] Performs the second huge (very favorable) swap to make the Uniswap pool balanced again and gets more ETH because of Fuji's collateral.
8. [ATTACKER] Earns swap profit including Fuji's collateral minus swap and flash loan fees.

The simulation has been made for the ETH/DAI Uniswap pool and ETH/DAI Vault with the following initial conditions:

- Uniswap pool DAI balance: 40 000 000
- Uniswap pool ETH balance: 20 000

- Fuji's credit to be repaid (equal to Fuji's swapped amount): 200 000 DAI
- Attacker flash loan amount: 15 000 ETH

The cost of the attack are following:

- Flash floan fee (0.09%) – 13.5 ETH
- Swap fees (0.3% x 2) – 90 ETH
- **Total: 103.5 ETH**

Profit:

- Stolen ETH: 208.45 ETH
- Cost: 103.5 ETH
- **Profit nett: 104.95 ETH**

References:
SCSVS V14: Decentralized Finance
https://github.com/securing/SCSVS/blob/master/1.1/0x23-V14-Decentralized-Finance.md

4.2. Excessively powerful Owner address

Risk impact	Medium	SCSVS V2
Vulnerable contracts	*Flasher, FujiERC1155, VaultETHDAI (and other vaults), VaultBase, Controller, Fliquidator, FujiAdmin, FujiMapping*	
Exploitation conditions	Access to the Owner's wallet.	
Exploitation results	Taking control over whole protocol, including user's collateral theft in many different ways described in test cases.	
Remediation	The remediation is divided into 3 categories: • Remove unnecessary function accessible by the owner or revoke ownership after the protocol is set up. • In short term we recommend to implement timelocks to delay the updates and allow users to react. • In long term we recommend to implement the governance with upgradability process. The remediation is explained in details below.	

Vulnerability description:
Audited contracts contain a lot of functions that can be called by the Owner only, which is an Externally Owned Account (not a contract). These functions include updating addresses of other contracts (within the scope of the audit) and business logic parameters. Some of the function calls can have significant consequences leading to the theft of users' collateral. Recent rug pulls have reduced trust in DeFi projects and too powerful functionalities may deter potential users.

The proposed remediations should not be introduced for the pausing functionality.

Test case:

Switching to a malicious provider

The *initiateFlashloan* function (Flasher.sol#L77) can be called by an authorized contract (address) which is controllable by owner. It allows to start the process of switching providers and new provider is not validated. As the consequence, changing provider to a malicious one allows to steal collateral.

Liquidating collateralized positions

The *initiateFlashloan* function (Flasher.sol#L77) can be called by an authorized contract (address) which is controllable by owner. It allows to start the process of flash batch liquidation which bypasses the check for the under-collateralized positions not present in *executeFlashBatchLiquidation* function (Fliquidator.sol#L405).

FujiERC1155 state update

The *updateState* function (FujiERC1155.sol#L66) allows permitted contracts (controllable by owner via *setPermit* function (FujiERC1155.sol#L25)) to increase total supply of any asset and indirectly increase the deposits of users. This could be used in one transaction with withdrawal call to steal collateral.

FujiERC1155 minting and burning

The following functions in FujiERC1155 contract allow owner and other permitted addresses to mint and burn any asset:

- mint (FujiERC1155.sol#L198),
- mintBatch (FujiERC1155.sol#L229),
- burn (FujiERC1155.sol#L268),
- burnBatch (FujiERC1155.sol#L295).

Malicious oracle

The *setOracle* function (VaultETHDAI.sol#L360) allows authorized contracts (controllable by owner via *setController* function (FujiAdmin.sol#L69)) to change the price oracle contract and manipulate the amount of required collateral.

Malicious flash close fee

The owner can define any value for *bonusL* parameter (FujiAdmin.sol#L97) when liquidating a position (in the same transaction) and potentially steal whole collateral of the user via liquidation bonus.

Malicious bonus factors

The owner can define any value for *flashCloseF* parameter (Fliquidator.sol#L543) when liquidating a position using flash (in the same transaction) and potentially steal whole collateral of the user.

Malicious Fliquidator

The owner can change the address of _fliquidator variable (FujiAdmin.sol#L52) which is allowed to withdraw whole collateral from the vaults (VaultETHDAI.sol#L171).

Malicious mapping

The owner can change mappings using *setMapping* function (FujiMapping.sol#L20). The mapping is used to get the address of the prividers' tokens for the underlying tokens. When its changed to a malicious contract all deposits are transferred to this contract and controlled by its owner.

Pausing whole protocol

All operations on the provider contract use *_execute* function with *whenNotPaused* modifier (VaultBase.sol#L93). It allows the owner not only to pause deposits but also withdrawals, thus lock users' funds. It is recommended not to pause the withdrawal functionality or introduce an emergency withdrawal with basic business logic.

Detailed remediation:

Timelocks

The timelock contract (Compound's example) allows to delay the updates in time (in terms of blocks) and thus it does not allow the Owner to get their transactions accepted before the users' transactions due to the transaction order dependency.

Governance

In the long term, a governance contract should be implemented (on top of the timelock contract). All updates should be voted and if they are accepted they should be queued in the timelock contract to be later activated.
The only exception is the pausing functionality which could be activated without voting, after a small number of signers accept it (e.g. 2 signers).

References:
SCSVS V2: Access control
https://securing.github.io/SCSVS/1.1/0x11-V2-Access-Control.html

4.3. Denial of service

Risk impact	Medium	SCSVS V9
Vulnerable contracts	*VaultControl*	
Exploitation conditions	Pausing the vault.	
Exploitation results	Vault is paused forever.	
Remediation	Fix the function call. The *unpause* function should call *_unpause*.	

Vulnerability description:
There is a typo in the *unpause* function (VaultBase.sol#L38). It calls *_pause* instead of *_unpause* function. When the vault is once paused it cannot be unpaused.

References:
SCSVS V9: Denial of service
https://securing.github.io/SCSVS/1.1/0x18-V9-Denial-Of-Service.html

4.4. Instant ownership transfer

Risk impact	Low	SCSVS V1
Vulnerable contracts	*Ownable (imported)*	
Exploitation conditions	Contract uses incorrect address of new owner.	
Exploitation results	The owner loses control over contracts.	
Remediation	Make the process of owner update a two-step process where the new owner must accept it (see references for details). If there is a need to remove owner, add new function that explicitly assign zero address to owner.	

Vulnerability description:
In case of a mistake in the address, the management of the contract will be irretrievably lost. It is suggested to extend the ownership delegation in such a way that the address taking over the ownership has to confirm owner address change.

Test case:
The *Ownable* contract (imported from OpenZeppelin) instantly transfers the owner in *transferOwnership* function:

```
/**
    * @dev Transfers ownership of the contract to a new account (`newOwner`).
    * Can only be called by the current owner.
```

```
    */
function transferOwnership(address newOwner) public virtual onlyOwner {
    require(newOwner != address(0), "Ownable: new owner is the zero address");
    _setOwner(newOwner);
}

function _setOwner(address newOwner) private {
    address oldOwner = _owner;
    _owner = newOwner;
    emit OwnershipTransferred(oldOwner, newOwner);
}
```

References:
SCSVS V1: Architecture, design and threat modelling
https://securing.github.io/SCSVS/1.1/0x10-V1-Architecture-Design-Threat-modelling.html
[Ownable] Grant and claim ownership
https://github.com/OpenZeppelin/openzeppelin-contracts/issues/2369

4.5. Unauthorized block of deposits

Risk impact	Low	SCSVS V2
Vulnerable contracts	*AlphaWhitelist*	
Exploitation conditions	Calling *whiteListRoutine* function as many times to make the counter variable greater than *limitUsers*.	
Exploitation results	Temporal block of deposits (until owner increases the *limitUsers* variable).	
Remediation	Allow to call *whiteListRoutine* function only by vaults.	

Vulnerability description:
The *AlphaWhitelist* contract is used to control the number of users and the maximum capacity. The *whiteListRoutine* function (AlphaWhitelist.sol#L35) can be called by anyone and increases the users counter. Malicious user can call it *limitUsers* times and block next calls until the owner set greater value of *limitUsers* parameter.

Test case:

```
  function whiteListRoutine(
    address _usrAddr,
    uint64 _assetID,
    uint256 _amount,
    address _erc1155
  ) external override returns (bool letgo) {
    uint256 currentBalance = IFujiERC1155(_erc1155).balanceOf(_usrAddr,
_assetID);
    if (currentBalance == 0) {
      counter = counter.add(1);
```

```
        letgo = _amount <= ethCapValue && counter <= limitUsers;
      } else {
        letgo = currentBalance.add(_amount) <= ethCapValue.mul(2);
      }
    }
```

References:
SCSVS V2: Access control
https://securing.github.io/SCSVS/1.1/0x11-V2-Access-Control.html

4.6. Front-runnable approve function

Risk impact	Low	SCSVS V8
Vulnerable contracts	*UniERC20*, *LibUniversalERC20*	
Exploitation conditions	Front-running victim's transaction that changes existing victim's allowance for attacker as spender (by using *approve* function).	
Exploitation results	Transferring victim's tokens using the previous allowance as well as the new allowance.	
Remediation	As there is not general remediation for this vulnerability, encourage users to not use this functionality. It is also recommended to allow allowance change from value greater that zero to zero only and from zero to greater than zero (does not allow to change from non-zero to non-zero value). Also, do not use the *approve* function in a DApp. Use the *increaseAllowance* and *decreaseAllowance* instead.	

Vulnerability description:
The *UniERC20* and *LibUniversalERC20* libraries implement ERC20 *uniApprove*, *uniTransfer*, *univApprove* and *univTransfer* functions which call *approve* function on ERC20 token. The function allows to set a given allowance of sender's tokens to a defined spender.
Front-running the transaction that changes existing allowance makes it possible for the malicious spender to transfer victim's tokens before the *approve* function call is added to block and again, using another transaction, to transfer the newly set amount of tokens. As the consequence, the attacker has transferred the sum of both allowances and not the second allowance only, as it was desired by the victim.
Note: The *SafeERC20* library used by the mentioned libraries correctly mitigates this risk but the libraries introduce it again.

Test case:
The vulnerable implementation of *uniApprove* function in *UniERC20* library:

```
function uniApprove(
    IERC20 token,
    address to,
```

```
    uint256 amount
  ) internal {
    require(!isETH(token), "Approve called on ETH");

    if (amount == 0) {
      token.safeApprove(to, 0);
    } else {
      uint256 allowance = token.allowance(address(this), to);
      if (allowance < amount) {
        if (allowance > 0) {
          token.safeApprove(to, 0);
        }
        token.safeApprove(to, amount);
      }
    }
  }
```

The vulnerable implementation of *univApprove* function in *LibUniversalERC20* library:

```
function univApprove(
    IERC20 token,
    address to,
    uint256 amount
  ) internal {
    require(!isETH(token), "Approve called on ETH");

    if (amount == 0) {
      token.safeApprove(to, 0);
    } else {
      uint256 allowance = token.allowance(address(this), to);
      if (allowance < amount) {
        if (allowance > 0) {
          token.safeApprove(to, 0);
        }
        token.safeApprove(to, amount);
      }
    }
  }
```

References:
SCSVS V8: Business logic
https://securing.github.io/SCSVS/1.1/0x17-V8-Business-Logic.html

4.7. FujiERC1155 transfers locked forever

Risk impact	Low	SCSVS V8
Vulnerable contracts	*FujiBaseERC1155*	
Exploitation conditions	None	
Exploitation results	Transfers of FujiERC1155 tokens cannot be unlocked.	
Remediation	Add function to activate transfers of ERC1155 tokens in the future.	

Vulnerability description:

Functions *safeTransferFrom* and *safeBatchTransferFrom* have *isTransferActive* modifier that blocks them until the *transferActive* variable is set to true. However, there is no way to change its value.

Test case:

The *safeTransferFrom* implementation (FujiBaseERC1155.sol#L139):

```
/**
* @dev See {IERC1155-safeTransferFrom}.
*/
function safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
) public virtual override isTransferActive {
    require(to != address(0), Errors.VL_ZERO_ADDR_1155);
    require(
        from == _msgSender() || isApprovedForAll(from, _msgSender()),
        Errors.VL_MISSING_ERC1155_APPROVAL
    );
    (...)
```

The *safeBatchTransferFrom* implementation (FujiBaseERC1155.sol#L177):

```
/**
* @dev See {IERC1155-safeBatchTransferFrom}.
*/
function safeBatchTransferFrom(
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
) public virtual override isTransferActive {
    require(ids.length == amounts.length, Errors.VL_INPUT_ERROR);
    require(to != address(0), Errors.VL_ZERO_ADDR_1155);
    require(
        from == _msgSender() || isApprovedForAll(from, _msgSender()),
```

```
        Errors.VL_MISSING_ERC1155_APPROVAL
    );
    (...)
```

References:
SCSVS V8: Business logic
https://securing.github.io/SCSVS/1.1/0x17-V8-Business-Logic.html

4.8. Function non-compliant with ERC1155

Risk impact	Low	SCSVS V8
Vulnerable contracts	*FujiERC1155*	
Exploitation conditions	None	
Exploitation results	Non-compliant *setURI* function.	
Remediation	Make the function ERC1155-compliant. See references for details.	

Vulnerability description:
Function *setURI* (FujiERC1155.sol#L354) does not emit URI event and does not update it for a specific token ID.

Test case:
The *setURI* implementation (FujiERC1155.sol#L354):

```
/**
* @dev Sets a new URI for all token types, by relying on the token type ID
*/
function setURI(string memory _newUri) public onlyOwner {
    _uri = _newUri;
}
```

References:
SCSVS V10: Token
https://securing.github.io/SCSVS/1.1/0x19-V10-Token.html
EIP-1155: ERC-1155 Multi Token Standard
https://eips.ethereum.org/EIPS/eip-1155

4.9. Function does not check the *return* value

Risk impact	Low	SCSVS V8
Vulnerable contracts	*HelperFunc*	
Exploitation conditions	Function *enterMarkets* or *exitMarket* return errors.	
Exploitation results	Continuing execution of the flow that should be reverted and loss of gas.	
Remediation	Check the return value of *enterMarkets* and *exitMarket* functions and control the flow depending on their value (e.g. revert if there are any erros).	

Vulnerability description:
Functions doesn't check the return value of function that return potential errors and continue execution of transaction while it should be reverted.

Test case:

- *HelperFunct (ProviderCompound)*
  - ProviderCompound.sol#L105
  - ProviderCompound.sol#L116
- *HelperFunct* (ProviderIronBank)
  - ProviderIronBank.sol#L103
  - ProviderIronBank.sol#L114

References:
SCSVS V8: Business logic
https://securing.github.io/SCSVS/1.1/0x17-V8-Business-Logic.html

5. RECOMMENDATIONS

5.1. Use specific Solidity compiler version

Description:
Audited contracts use the following pragma:

```
pragma solidity >=0.6.12;
```

It allows to compile contracts with old versions of compiler and to recent versions. Also, use the latest stable version (0.6.x) for testing.

How to implement:
Use a specific version of Solidity compiler (latest stable):

```
pragma solidity 0.6.12;
```

References:
Floating Pragma
https://swcregistry.io/docs/SWC-103
SCSVS V1: Architecture, design and threat modelling
https://securing.github.io/SCSVS/1.1/0x10-V1-Architecture-Design-Threat-modelling.html

5.2. Extend the list of abuser stories in unit tests

Description:
The source code covers a wide list of scenarios in unit tests. However, some are missing. It is recommended to cover more abuser stories.

How to implement:
We recommend to add the e.g. following abuser stories to unit tests:

- Position liquidation process in various scenarios.
- Access to management functions by no t authorized or permitted users.

References:
SCSVS V12: Test coverage
https://securing.github.io/SCSVS/1.1/0x21-V12-Test-Coverage.html

5.3. Do not duplicate the code of modifiers and follow the code clarity rules

Description:
Modifier *onlyPermit* (FujiERC1155#L20) should be renamed to *isAuthorized*.

```
  modifier onlyPermit() {
    require(addrPermit[_msgSender()] || msg.sender == owner(),
Errors.VL_NOT_AUTHORIZED);
```

```
    _;
  }
```

Modifier *isAuthorized* (Fliquidator#L67) should be renamed to *onlyOwner*.

```
  modifier isAuthorized() {
    require(msg.sender == owner(), Errors.VL_NOT_AUTHORIZED);
    _;
  }
```

How to implement:

General approach:

- Use separate contract to define all required roles i.e., *AccessControl* contract.
- Use *isAuthorized* if you have 2 and more possible options.
- Use precise naming if you have just one requirement.

References:

SCSVS V11: Code clarity

https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

5.4. Define variables explicitly

Description:

Variables are not defined explicitly what reduces code clarity.

- Variable *modes* in *_initiateAaveFlashLoan* function (Flasher#L181):

```
//var modes[0] = 0;
```

- Storage variable *counter* (AlphaWhitelist#L14):

```
contract AlphaWhitelist is IAlphaWhiteList, Ownable {
  using SafeMath for uint256;

  uint256 public ethCapValue;
  uint256 public limitUsers;
  uint256 public counter;

  // Log Limit Users Changed
```

- Variable *debtBalanceTotal* in *batchLiquidate* function (Fliquidator#L126):

```
    uint256[] memory usrsBals = f1155.balanceOfBatch(formattedUserAddrs,
formattedIds);

    uint256 neededCollateral;
    uint256 debtBalanceTotal;
```

How to implement:

Initialize variables explicitly by defining their default values. See example below.

```
        uint256 public counter;
```

should be changed to

```
        uint256 public counter = 0;
```

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

5.5. Remove deprecated functions

Description:
The *LibUniversalERC20* library is a newer version of the *UniERC20* library. In case of creating a new function responsible for a specific business logic, all places in the system should be updated.
LibUniversalERC20 uses *call* instead of *send* in the *univTransfer* function. However, there are still places where *uniTransfer* is still in use:
Flasher

```
136,24: IERC20(info.asset).uniTransfer(payable(info.vault), info.amount);
210,23: IERC20(assets[0]).uniTransfer(payable(info.vault), amounts[0]);
278,24: IERC20(underlying).uniTransfer(payable(info.vault), amount);
297,24: IERC20(underlying).uniTransfer(payable(crToken), amountOwing);
```

VaultETHDAI

```
166,39: IERC20(vAssets.collateralAsset).uniTransfer(msg.sender,
amountToWithdraw);
172,39: IERC20(vAssets.collateralAsset).uniTransfer(msg.sender,
uint256(_withdrawAmount));
207,33: IERC20(vAssets.borrowAsset).uniTransfer(msg.sender, _borrowAmount);
289,33: IERC20(vAssets.borrowAsset).uniTransfer(msg.sender,
_flashLoanAmount.add(_fee));
473,27: IERC20(tokenReturned).uniTransfer(payable(_fujiAdmin.getTreasury()),
tokenBal);
```

VaultETHUSDC

```
166,39: IERC20(vAssets.collateralAsset).uniTransfer(msg.sender,
amountToWithdraw);
172,39: IERC20(vAssets.collateralAsset).uniTransfer(msg.sender,
uint256(_withdrawAmount));
207,33: IERC20(vAssets.borrowAsset).uniTransfer(msg.sender, _borrowAmount);
289,33: IERC20(vAssets.borrowAsset).uniTransfer(msg.sender,
_flashLoanAmount.add(_fee));
473,27: IERC20(tokenReturned).uniTransfer(payable(_fujiAdmin.getTreasury()),
tokenBal);
```

VaultETHUSDT

```
166,39: IERC20(vAssets.collateralAsset).uniTransfer(msg.sender,
amountToWithdraw);
172,39: IERC20(vAssets.collateralAsset).uniTransfer(msg.sender,
uint256(_withdrawAmount));
207,33: IERC20(vAssets.borrowAsset).uniTransfer(msg.sender, _borrowAmount);
289,33: IERC20(vAssets.borrowAsset).uniTransfer(msg.sender,
_flashLoanAmount.add(fee));
472,27: IERC20(tokenReturned).uniTransfer(payable(_fujiAdmin.getTreasury()),
tokenBal);
```

LibUniversalERC20 has *univApprove* function. However, there are still places where *uniApprove* is still in use:
Flasher.sol

```
233,23: IERC20(assets[0]).uniApprove(payable(_aaveLendingPool), amountOwing);
```

ProviderCompound.sol

```
155,18: erc20token.uniApprove(address(cTokenAddr), _amount);
220,18: erc20token.uniApprove(address(cTokenAddr), _amount);
```

ProviderIronBank.sol

```
152,16: erc20token.uniApprove(address(cyTokenAddr), _amount);
221,16: erc20token.uniApprove(address(cyTokenAddr), _amount);
```

How to implement:
Use ***univTransfer*** instead of *uniTransfer*.
Use ***univApprove*** instead of *uniApprove*.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

5.6. Correct comments

Description:
Comments in the contracts should be clear, understandable and truthful. Some of the comments however, use non-existing *CallTypes*. The following comments need to be corrected:

- *LibFlashLoan* – there is no *Liquidate* enum, mentioned in the comment.

```
/**
   * @dev Used to determine which vault's function to call post-flashloan:
   * - Switch for executeSwitch(...)
   * - Close for executeFlashClose(...)
   * - Liquidate for executeFlashLiquidation(...)
   * - BatchLiquidate for executeFlashBatchLiquidation(...)
   */
(…)
  enum CallType { Switch, Close, BatchLiquidate }
(…)
```

- *LibFlashLoan* – there is no *Liquidate* enum, mentioned in the comment.

```
/**
   * @dev Struct of params to be passed between functions executing flashloan
logic
   * @param asset: Address of asset to be borrowed with flashloan
   * @param amount: Amount of asset to be borrowed with flashloan
   * @param vault: Vault's address on which the flashloan logic to be executed
   * @param newProvider: New provider's address. Used when callType is Switch
   * @param userAddrs: User's address array Used when callType is BatchLiquidate
```

```
   * @param userBals:  Array of user's balances, Used when callType is
BatchLiquidate
   * @param userliquidator: The user's address who is performing liquidation.
Used when callType is Liquidate
   * @param fliquidator: Fujis Liquidator's address.
   */
(…)
struct Info {
    CallType callType;
    address asset;
    uint256 amount;
    address vault;
    address newProvider;
    address[] userAddrs;
    uint256[] userBalances;
    address userliquidator;
    address fliquidator;
  }
```

How to implement:
Apply a comment consistent with the current state of the implementation.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

5.7. Make functions external to optimise gas costs

Description:
Functions marked as *external* use less gas than *public*. Following functions should be described as *external*:

- Flasher.sol#L68

```
function setFujiAdmin(address _newFujiAdmin) public onlyOwner {
```

- FujiBaseERC1155.sol#L53

```
function totalSupply(uint256 id) public view virtual returns (uint256) {
```

- FujiBaseERC1155.sol#L60

```
  function supportsInterface(bytes4 interfaceId)
    public
    view
    override(ERC165, IERC165)
    returns (bool)
  {
```

- FujiBaseERC1155.sol#L77

```
function uri(uint256) public view virtual returns (string memory) {
```

- FujiBaseERC1155.sol#L96

```
function balanceOfBatch(address[] memory accounts, uint256[] memory ids)
    public
```

```
    view
    override
    returns (uint256[] memory)
  {
```

- FujiBaseERC1155.sol#L116

```
function setApprovalForAll(address operator, bool approved) public virtual
override {
```

- FujiBaseERC1155.sol#L139

```
function safeTransferFrom(
    address from,
    address to,
    uint256 id,
    uint256 amount,
    bytes memory data
  ) public virtual override isTransferActive {
```

- FujiBaseERC1155.sol#L177

```
function safeBatchTransferFrom(
    address from,
    address to,
    uint256[] memory ids,
    uint256[] memory amounts,
    bytes memory data
  ) public virtual override isTransferActive {
```

- F1155Manager.sol#L25

```
function setPermit(address _address, bool _permit) public onlyOwner {
```

- AlphaWhitelist.sol#L56

```
function updateLimitUser(uint256 _newUserLimit) public onlyOwner {
```

- AlphaWhitelist.sol#L65

```
function updateCap(uint256 _newEthCapValue) public onlyOwner {
```

- FujiMapping.sol#L20

```
function setMapping(address _addr1, address _addr2) public onlyOwner {
```

- FujiMapping.sol#L27

```
  function setURI(string memory newUri) public onlyOwner {
```

How to implement:
Describe mentioned functions as *external*.

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

5.8. Improve code clarity

Description:
There are a few things worth working on to increase code clarity. Readable and well-described code allows to detect and remove bugs easier in a shorter testing time. Currently, contracts contain following issues:

- Unused code,
- Unnecessary code,
- Unnecessary comments,
- Confusing names.

How to implement:

Remove unused code:

- Flasher.sol#L128

```
  function callFunction(
    address sender,
    Account.Info calldata account,
    bytes calldata data
  ) external override {
    require(msg.sender == _dydxSoloMargin && sender == address(this),
Errors.VL_NOT_AUTHORIZED);
    account;
```

- FujiERC1155.sol#L333

```
  function getAssetID(AssetType _type, address _addr) external view override
returns (uint256 id) {
    id = assetIDs[_type][_addr];
    require(id <= qtyOfManagedAssets, Errors.VL_INVALID_ASSETID_1155);
  }
```

- HelperFunct (ProviderCompound.sol#L112, ProviderIronBank.sol#L110) - function *exitCollatMarket*

```
  function _exitCollatMarket(address _cTokenAddress) internal {
    // Create a reference to the corresponding network Comptroller
    IComptroller comptroller = IComptroller(_getComptrollerAddress());

    comptroller.exitMarket(_cTokenAddress);
  }
```

- ProviderDYDX.sol#L171

```
  bool public donothing = true;
```

- MathUtils.sol - whole library

Remove unnecessary code:

- FujiBaseERC1155.sol#L164 (the requirement will be checked in L166 using sub function) and FujiBaseERC1155.sol#167 (*unit256* casting of *_balances*).

```
function safeTransferFrom(
(…)
```

```
    uint256 fromBalance = _balances[id][from];
    require(fromBalance >= amount, Errors.VL_NO_ERC1155_BALANCE);

    _balances[id][from] = fromBalance.sub(amount);
    _balances[id][to] = uint256(_balances[id][to]).add(amount);

    emit TransferSingle(operator, from, to, id, amount);

    _doSafeTransferAcceptanceCheck(operator, from, to, id, amount, data);
  }
```

- FujiBaseERC1155.sol#L202 (*unit256* casting of *_balances*)

```
function safeBatchTransferFrom(
(…)

      uint256 fromBalance = _balances[id][from];
      require(fromBalance >= amount, Errors.VL_NO_ERC1155_BALANCE);
      _balances[id][from] = fromBalance.sub(amount);
      _balances[id][to] = uint256(_balances[id][to]).add(amount);
    }
```

Remove unnecessary comments

- FujiERC1155.sol#L48-L54

```
  // Optimizer Fee expressed in Ray, where 1 ray = 100% APR
  //uint256 public optimizerFee;
  //uint256 public lastUpdateTimestamp;
  //uint256 public fujiIndex;

  /// @dev Ignoring leap years
  //uint256 internal constant SECONDS_PER_YEAR = 365 days;
```

- FujiERC1155.sol#*L81-95*

```
// TODO: calculate interest rate for a fujiOptimizer Fee.
      /*
      if(lastUpdateTimestamp==0){
        lastUpdateTimestamp = block.timestamp;
      }

      uint256 accrued = _calculateCompoundedInterest(
        optimizerFee,
        lastUpdateTimestamp,
        block.timestamp
      ).rayMul(fujiIndex);

      fujiIndex = accrued;
      lastUpdateTimestamp = block.timestamp;
      */
```

- FujiERC1155.sol#L138-156

```
**
   * @dev Returns the balance of User, split into owed amounts to BaseProtocol
and FujiProtocol
   * @param _account: address of the User
```

```
   * @param _assetID: ERC1155 ID of the asset which state will be updated.
   **/
  /*
  function splitBalanceOf(
    address _account,
    uint256 _assetID
  ) public view override returns (uint256,uint256) {
    uint256 scaledBalance = super.balanceOf(_account, _assetID);
    if (scaledBalance == 0) {
      return (0,0);
    } else {
    TO DO COMPUTATION
      return (baseprotocol, fuji);
    }
  }
  */
```

- FujiERC1155.sol#L172-187

```
/**
   * @dev Returns the sum of balance of the user for an AssetType.
   * This function is used for when AssetType have units of account of the same
value (e.g stablecoins)
   * @param _account: address of the User
   * @param _type: enum AssetType, 0 = Collateral asset, 1 = debt asset
   **/
  /*
  function balanceOfBatchType(address _account, AssetType _type) external view
override returns (uint256 total) {

    uint256[] memory IDs = engagedIDsOf(_account, _type);

    for(uint i; i < IDs.length; i++ ){
      total = total.add(balanceOf(_account, IDs[i]));
    }
  }
  */
```

- FujiERC1155.sol#L340-350

```
/**
   * @dev Sets the FujiProtocol Fee to be charged
   * @param _fee; Fee in Ray(1e27) to charge users for optimizerFee (1 ray = 100%
APR)
   */
  /*
  function setoptimizerFee(uint256 _fee) public onlyOwner {
    require(_fee >= WadRayMath.ray(), Errors.VL_OPTIMIZER_FEE_SMALL);
    optimizerFee = _fee;
  }
  */
```

- FujiERC1155.sol#L380-420

```
/**
   * @dev Function to calculate the interest using a compounded interest rate
formula
```

```
   * To avoid expensive exponentiation, the calculation is performed using a 
binomial approximation:
   *
   *  (1+x)^n = 1+n*x+[n/2*(n-1)]*x^2+[n/6*(n-1)*(n-2)*x^3...
   *
   * The approximation slightly underpays liquidity providers and undercharges 
borrowers, with the advantage of great gas cost reductions
   * The whitepaper contains reference to the approximation and a table showing 
the margin of error per different time periods
   *
   * @param _rate The interest rate, in ray
   * @param _lastUpdateTimestamp The timestamp of the last update of the interest
   * @return The interest rate compounded during the timeDelta, in ray
   **/
  /*
  function _calculateCompoundedInterest(
    uint256 _rate,
    uint256 _lastUpdateTimestamp,
    uint256 currentTimestamp
  ) internal pure returns (uint256) {
    //solium-disable-next-line
    uint256 exp = currentTimestamp.sub(uint256(_lastUpdateTimestamp));

    if (exp == 0) {
      return WadRayMath.ray();
    }

    uint256 expMinusOne = exp - 1;

    uint256 expMinusTwo = exp > 2 ? exp - 2 : 0;

    uint256 ratePerSecond = _rate / SECONDS_PER_YEAR;

    uint256 basePowerTwo = ratePerSecond.rayMul(ratePerSecond);
    uint256 basePowerThree = basePowerTwo.rayMul(ratePerSecond);

    uint256 secondTerm = exp.mul(expMinusOne).mul(basePowerTwo) / 2;
    uint256 thirdTerm = exp.mul(expMinusOne).mul(expMinusTwo).mul(basePowerThree) 
/ 6;

    return 
WadRayMath.ray().add(ratePerSecond.mul(exp)).add(secondTerm).add(thirdTerm);
  }
  */
```

Change confusing names:

- ProviderDYDX.sol#L190 - The *tweth* variable is not *WETH* and *IWethERC20* interface. Name it according to the actual use.

```
  function deposit(address _asset, uint256 _amount) external payable override { 
(…)
    } else {
      IWethERC20 tweth = IWethERC20(_asset);
      tweth.approve(getDydxAddress(), _amount);
(…)
```

- ProviderDYDX.sol#L254 – The *_asset* token is not *WETH*. Name it according to the actual use.

```
function payback(address _asset, uint256 _amount) external payable override {
(…)
    } else {
      IWethERC20 tweth = IWethERC20(_asset);
      tweth.approve(getDydxAddress(), _amount);
(…)
  }
```

References:
SCSVS V11: Code clarity
https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

5.9. Use constant for hardcoded addresses

Description:
Use *constant* for hardcoded addresses. In the case of updating, it is enough to assign the correct address in one place, instead of updating all where it is used. This not only prevents future errors, but also minimizes the chance of a typo.
Constant is also more gas efficient than *immutable* state variable:

- Flasher.sol#L47-L49

```
(…)
address private immutable _aaveLendingPool =
0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9;
  address private immutable _dydxSoloMargin =
0x1E0447b19BB6EcFdAe1e4AE1694b0C3659614e4e;
  IFujiMappings private immutable _crMappings =
    IFujiMappings(0x03BD587Fe413D59A20F32Fc75f31bDE1dD1CD6c9);
(…)
```

- ProviderDYDX.sol#L87

```
  function getDydxAddress() public pure returns (address addr) {
    addr = 0x1E0447b19BB6EcFdAe1e4AE1694b0C3659614e4e;
  }
```

- ProviderDYDX.sol#L94

```
  function getWETHAddr() public pure returns (address weth) {
    weth = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
  }
```

- ProviderDYDX.sol#L101

```
  function _getEthAddr() internal pure returns (address) {
    return 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE; // ETH Address
  }
```

- ProviderIronBank.sol#L138

```
function deposit(address _asset, uint256 _amount) external payable override {
(…)
```

```
      // Transform ETH to WETH
      IWeth(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).deposit{ value: _amount
}();
      _asset = address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
    }
```

- ProviderIronBank.sol#L175

```
    if (_isETH(_asset)) {
      // Transform ETH to WETH
      IWeth(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).withdraw(_amount);
    }
```

  - *#L209*

```
  function payback(address _asset, uint256 _amount) external payable override {
(…)
      // Transform ETH to WETH
      IWeth(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2).deposit{ value: _amount
}();
      _asset = address(0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2);
    }
```

How to implement:

Use constant for hardcoded addresses See example below:

- Flasher.sol#L47-L49

```
(…)
address private immutable _aaveLendingPool =
0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9;
  address private immutable _dydxSoloMargin =
0x1E0447b19BB6EcFdAe1e4AE1694b0C3659614e4e;
  IFujiMappings private immutable _crMappings =
    IFujiMappings(0x03BD587Fe413D59A20F32Fc75f31bDE1dD1CD6c9);
(…)
```

should be changed to

```
(…)
address private constant _aaveLendingPool =
0x7d2768dE32b0b80b7a3454c06BdAc94A69DDc7A9;
  address private constant _dydxSoloMargin =
0x1E0447b19BB6EcFdAe1e4AE1694b0C3659614e4e;
  IFujiMappings private constant _crMappings =
    IFujiMappings(0x03BD587Fe413D59A20F32Fc75f31bDE1dD1CD6c9);
(…)
```

References:

SCSVS V11: Code clarity

https://securing.github.io/SCSVS/1.1/0x20-V11-Code-Clarity.html

5.10. Consider mocking the external contracts

Description:
Mocking external contracts have multiple benefits. Changes in the environment might cause the previously successful tests to fail. With mocked contracts, you can easily and fast check how new component/functions influence the system.
Forking the mainnet does not return the same environment each time the tests are run. Consider use of mocked contracts as additional test environment for basic verification of the system after each meaningful change.

How to implement:
We recommend to create mocks for external services (e.g. price oracle, tokens) and test on local test network with the same environment on each test run.

References:
SCSVS V1: Architecture, design and threat modelling
https://securing.github.io/SCSVS/1.1/0x10-V1-Architecture-Design-Threat-modelling.html

6. SCSVS

The application has been verified for compliance with the Smart Contracts Security Verification Standard (SCSVS), v. 1.1.

Legend:
- Passed – The application meets the requirement.
- Failed – The application does not meet the requirements.
- N/A – The requirement does not apply to system (e.g. the system does not include the feature to which the requirement applies) or is out of scope.

The results of SCSVS compliance verification are attached to the report in the *SCSVS_Fuji_20210716.xlsx* file.

6.1. Failed checks

The list of failed checks:
- V1: Architecture, design and threat modelling
    - Verify that every introduced design change is preceded by an earlier threat modelling.
    - Verify that there exists an upgrade process for contract which allows to deploy the security fixes.
    - Verify that there exists a mechanism that can temporarily stop the sensitive functionalities of the contract in case of a new attack. This mechanism should not block access to the assets (e.g. tokens) for the owners.
    - Verify that the business logic in contracts is consistent. Important changes in the logic should be allowed for all or none contract.
    - Verify that the latest version of solidity is used.
- V2: Access Control
    - Verify that the creator of the contract complies with the rule of least privilege and their rights strictly follow the documentation.
    - Verify that there is a centralized mechanism for protecting access to each type of protected resource.
    - Verify that the code of modifiers is clear and simple. The logic should not contain external calls to untrusted contracts.
    - Verify that all user and data attributes used by access controls are kept in trusted contract and cannot be manipulated by other contracts unless specifically authorized.
- V7: Gas Usage & Limitations
    - Verify that the external keyword is used for functions that can be called externally only to save gas.

- V8: Business Logic
  - Verify that contract logic implementation corresponds to the documentation.
  - Verify the contract has business limits and correctly enforces it.
  - Verify that contract uses mechanisms that mitigate transaction-ordering dependence (front-running) attacks (e.g. pre-commit scheme).
- V9: Denial of Service
  - Verify that if fallback function is not callable by anyone, it is not blocking the functionalities of contract and the contract is not vulnerable to Denial of Service attacks.
- V10: Token
  - Verify that token contract follows tested and stable token standard.
  - Use the approve function of the ERC-20 standard to change allowed amount only to 0 or from 0.
- V11: Code Clarity
  - Verify that the same rules for variable naming are followed throughout all the contracts (e.g. use the same variable name for the same object).
  - Verify that all storage variables are initialised.
  - Verify that functions which specify a return type return the value.
  - Verify that all functions are used. Unused ones should be removed.
- V12: Test Coverage
  - Verify that all functions of verified contract are covered with tests in the development phase.
  - Verify that the specification and the result of formal verification is included in the documentation.
- V13: Known attacks
  - Verify that the contract is not vulnerable to Access Control issues.
  - Verify that the contract is not vulnerable to Denial of Service attacks.
  - Verify that the contract is not vulnerable to Front-Running attacks.
- V14: Decentralized Finance
  - Verify that, when using an on-chain oracles, the smart contract is able to pause the operations based on the oracle's result (in case of oracle has been compromised).
  - Verify that the external contracts (even trusted) that are allowed to change the attributes of the smart contract (e.g. token price) have the following limitations implemented: a thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).

7. TERMINOLOGY

This section explains the terms that are related to the methodology used in this report.

Risk	=	Threat	+	Vulnerability

Threat

Any circumstance or event with the potential to adversely impact organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals through an information system via unauthorized access, destruction, disclosure, modification of information, and/or denial of service.[1]

Vulnerability

Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source.[1]

Risk

The level of impact on organizational operations (including mission, functions, image, or reputation), organizational assets, or individuals resulting from the operation of an information system given the potential impact of a threat and the likelihood of that threat occurring.[1]

The risk impact can be estimated based on the complexity of exploitation conditions (representing the likelihood) and the severity of exploitation results.

		Complexity of exploitation conditions		
		Simple	Moderate	Complex
Severity of exploitation results	Major	Critical	High	Medium
	Moderate	High	Medium	Low
	Minor	Medium	Low	Low

[1] NIST FIPS PUB 200: Minimum Security Requirements for Federal Information and Information Systems. Gaithersburg, MD: Computer Security Division, Information Technology Laboratory, National Institute of Standards and Technology.

8. CONTACT

Person responsible for providing explanations:

Damian Rusinek
e-mail: damian.rusinek@securing.pl

securing
http://www.securing.pl
e-mail: info@securing.pl
Kalwaryjska 65/6
30-504 Kraków
tel./fax.: +48 (12) 425 25