// HALBORN

Sacred Finance

Smart Contract and Circom Circuit Security Audit

Prepared by: **Halborn**

Date of Engagement: **June 6th, 2022 - July 25th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	06/24/2022	Alessandro Cara
0.2	Document Amended	07/18/2022	Alessandro Cara
0.3	Draft Review	07/18/2022	Kubilay Onur Gungor
0.4	Draft Review	07/18/2022	Gabi Urrutia
1.0	Remediation Plan	08/12/2022	István Böhm
1.1	Remediation Plan Review	08/23/2022	Kubilay Onur Gungor
1.2	Remediation Plan Review	08/23/2022	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com

Alessandro Cara	Halborn	Alessandro.Cara@halborn.com
Kubilay Onur Gungor	Halborn	kubilay.gungor@halborn.com
István Böhm	Halborn	Istvan.Bohm@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

engaged Halborn to conduct a security audit on their smart contracts beginning on June 6th, 2022 and ending on July 25th, 2022 . The security assessment was scoped to the smart contracts provided to the Halborn team.

Sacred finance is a fork of Tornado Cash with added functionality that allows their users to anonymously leverage DeFi protocols like Aave to earn yield on their deposits. By using ZK-SNARKS, Sacred Finance is able to achieve privacy for both withdrawals and yield farming.

1.2 AUDIT SUMMARY

The team at Halborn was provided six weeks for the engagement and assigned two full-time security engineers to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

Halborn reviewed Solidity's smart contracts, technical documentation and the SNARK circuits written in the Circom language. The Circuits and the Solidity contracts share many similarities with the Tornado Cash's code, which has been previously audited and battle-tested in a real environment.

Due to the nature of zero knowledge proofs, the secure circuit code cannot be tampered and this resulted in the prevention of unauthorized access to user funds. However, users' privacy revolves around their actions and therefore they must follow the privacy guidelines provided by the Sacred Finance team. It was understood that the backend that generates

the deposit note and the ZK proof of a deposit, would only record pending event deposits; therefore it is not possible to register deposit events that did not actually go through the entire on-chain flow. During the smart contract audit, Halborn made use of backend test JavaScript code, both to test and better understand the protocol as a whole. However, a full evaluation of the backend is recommended. This will ensure that there are no vulnerabilities present in these components, which are heavily relied upon by the entire system.

Sacred Finance allows users to withdraw funds through a relayer, which would charge a fee of the deposit amount. This makes it possible to fund new wallets directly with Sacred Finance, and while anyone can become a relayer, there are protections in place to ensure that the relayer does not change the fee because it is included in the SNARK proof. Note that the security of relayers should also be considered, as their details are public and could be subject to attacks such as denial of service, as a way of damaging the protocol and its users.

Within Sacred Finance, double-spend attacks are prevented using a nullified hash, which is the hash of the note nullified, stored on-chain. All withdraw operations check that the nullified hash was not stored in the smart contract.

To summarize the findings, Halborn identified that most contracts, which use an initialization pattern, could be frontrun at the deployment stage and initialized by a malicious actor. If this were identified by the Sacred Finance team, the damage would only be in terms of gas costs spent to deploy the contracts and to re-deploy. On the other hand, due to the nature of the code that does not revert to a failed initialization, if the initialization frontrun goes unnoticed, the attacker is likely to control the system and user funds could be at risk. Furthermore, one of the functions within the ETHSacred contract was publicly accessible, and users could target this to reduce the compound yield earned by the deposits on Aave. For the correct functioning of the contract, the function can be set as internal/private, since it is already called on each deposit and on each withdrawal.

Furthermore, the code was using an outdated version of Solidity that

does not offer security features such as built-in arithmetic operation safety, and certain functions lacked input validation, which could cause parameters to be set to invalid values. For instance, setting certain values to zero, would result in divisions by zero and reverting in function calls. Halborn recommends that validation be enforced on all smart contracts to prevent these edge cases.

In summary, Halborn identified some security risks that were addressed by Sacred Finance team.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this audit. While manual testing is recommended to uncover flaws in logic, process, and implementation, automated testing techniques help enhance coverage of the protocol code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the audit:

- Research into architecture and purpose
- Smart contract manual code review and walkthrough
- Graphing out functionality and contract logic/connectivity/functions (solgraph)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes
- Manual testing by custom scripts
- Static Analysis of security for scoped contract, and imported functions (Slither)
- Testnet deployment (Brownie, Remix IDE)

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident

and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

10 - CRITICAL
9 - 8 - HIGH
7 - 6 - MEDIUM
5 - 4 - LOW
3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

The assessment was scoped to the following repositories and commit IDs:

- Sacred-Deploy - Commit ID **f9d9f3d43843fb8fd1544be0aac62971970b5a79**
- Sacred Anonymity Mining - Commit ID **f4c0f9b8201674721f66dc0aed609830a3a9cfb6**
- Sacred Trees - Commit ID **9728ba27a827edf31d748fb9a08232ced0fdb871**
- Sacred Contracts Eth - Commit ID **6a54d0aa72eda2d3a600dfc9fb6bc2723db23990**
- Sacred Token - Commit ID - Commit ID **c45376cc918e6dfbad28b19269d090344abe311c**

The remediation checks were performed on the following repository and commit ID, containing the contract from the above repositories combined:

- Sacred-Deploy - Commit ID **f17a56a917bfc882bb46d1da6e443cd1dacfa62b**

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	2	2	4

LIKELIHOOD

IMPACT

	(HAL-01)			
	(HAL-03) (HAL-04)		(HAL-02)	
(HAL-05) (HAL-06) (HAL-07) (HAL-08)				

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 PUBLICLY AVAILABLE FUNCTION COULD REDUCE THE AMOUNT OF INTERESTS EARNED ON AAVE	Medium	SOLVED - 08/19/2022
HAL-02 INITIALIZATION CAN BE FRONTRUN	Medium	SOLVED - 08/19/2022
HAL-03 LACK OF TRANSFER-OWNERSHIP PATTERN	Low	SOLVED - 08/19/2022
HAL-04 MISSING ZERO VALUE CHECKS	Low	SOLVED - 08/19/2022
HAL-05 GAS OPTIMIZATIONS	Informational	SOLVED - 08/19/2022
HAL-06 MISSING REENTRANCY PROTECTION	Informational	SOLVED - 08/19/2022
HAL-07 OUTDATED SOLIDITY VERSION	Informational	SOLVED - 08/19/2022
HAL-08 REDUNDANT CODE	Informational	SOLVED - 08/19/2022

FINDINGS & TECH DETAILS

3.1 (HAL-01) PUBLICLY AVAILABLE FUNCTION COULD REDUCE THE AMOUNT OF INTERESTS EARNED ON AAVE - MEDIUM

Description:

The collectAaveInterests function in the ETHSacred contract was publicly accessible. This function allows calculating the amount of interest earned on Aave WETH deposits up to the current block, and automatically withdrawing that portion of WETH, exchanging it them with AWETH (Aave interest bearing WETH). While the amount withdrawn would be sent to the AaveInterestProxy contract, and thus stored securely, this would reduce the amount of compound interest earned on the deposit.

Code Location:

The following function code was available in the ETHSacred contract (Lines #93-101):

Listing 1

```
function collectAaveInterests() public payable {
    uint256 interests = AToken(wETHToken).balanceOf(address(this))
  - collateralAmount;
    if(interests > 0 && aaveInterestsProxy != address(0)) {
      address lendingPool = AddressesProvider(
lendingPoolAddressProvider).getPool();
      require(AToken(wETHToken).approve(wETHGateway, interests), "
aToken approval failed");
      WETHGateway(wETHGateway).withdrawETH(lendingPool, interests,
 aaveInterestsProxy);
      totalAaveInterests += interests;
    }
}
```

Recommendation:

Halborn recommends that this logic be reviewed, and the function is not made accessible by everyone. While the amount of additional interest that could have been earned from compounding might be small, attackers could still launch a griefing attack against Sacred Finance users due to the extremely cheap transaction fees on Polygon. In addition, this function was only called within this contract, so it could be done internally or privately. Finally, it is recommended to review if this function should also be called on the deposit operations, or only to withdrawals, to further maximize the time spent on Aave WETH pools.

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by setting the visibility of the `collectAaveInterests` function to private.

3.2 (HAL-02) INITIALIZATION CAN BE FRONTRUN - MEDIUM

Description:

It was identified that all the contracts that make use of an initialization function could be frontrun by an attacker during the deployment phase, who could then set critical parameters such as user roles. While Sacred Finance could re-deploy the contracts, it would incur financial costs. This was also evident from the deployment scripts which did not perform any checks whether the correct parameters were updated in the contracts by making use of an initialization function. Furthermore, if Sacred Finance deploys their contracts and does not identify that they have already been initialized, users could start using a system that was compromised from the start.

Code Location:

This applies to all contracts using an initializer function. Here is an example:

Listing 2

```
function initialize(address _miner) external {
    if(!initialized) {
      miner = _miner;
      initialized = true;
    }
}
```

The full list is provided below:

- AaveInterestsProxy.sol
- RewardSwap.sol
- SacredProxy.sol
- SacredTrees.sol

Risk Level:

Likelihood - 4
Impact - 2

Recommendation:

Halborn recommends that the initialization logic be checked to ensure that it cannot be frontrun. At a minimum, the initialize function must be reverted when called after initialization, and the logic must be included in the Constructor or a proxy pattern with deployment and initialization in the same function must be implemented instead.

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by adding the onlyOwner modifier to the initialize functions.

3.3 (HAL-03) LACK OF TRANSFER-OWNERSHIP PATTERN - LOW

Description:

Smart contracts did not implement a two-step transfer of ownership pattern. This pattern prevents transferring ownership of a smart contract to a wrong address, and never being able to retrieve control of it. While Sacred Finance might want to renounce ownership of the contracts, this should be done via the renounceOwnership function. On the other hand, the two-step ownership transfer pattern would allow to first propose a new owner, and only effectively transfer ownership when the new owner accepts it.

If the original owner identifies an error in the proposal, they can cancel it at any time before the new owner accepts it.

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Halborn recommends that the pattern be implemented where it is deemed necessary so that roles can be safely transferred between accounts. Furthermore, when not governed by a governance, all privileged accounts must use a multi-signature account.

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by implementing the transfer-ownership pattern for the Sacred contract.

3.4 (HAL-04) MISSING ZERO VALUE CHECKS - LOW

Description:

Halborn identified that, within the code, the setter functions lacked input validation. While these functions can only be called by an administrative role, if they make a mistake an update error and set an invalid value, this could lead to calculation errors and potentially other functions reverting due to division by zero.

Code Location:

Listing 3

```
swap.poolWeight()
100000000000
>>> swap.setPoolWeight(0, {'from':miner})
Transaction sent: 0
↳ x655ca2224e0cf75fc00b6a8589e0d27ca78da9fec7fdfcee71f4adc967e129e0
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 63
  Transaction confirmed   Block: 32477231   Gas used: 14178
↳ (0.05%)

<Transaction '0
↳ x655ca2224e0cf75fc00b6a8589e0d27ca78da9fec7fdfcee71f4adc967e129e0'
↳ >
>>> swap.swap(owner, 99999999999999999999999999999, {'from': miner
↳ })
Transaction sent: 0
↳ x9be32c81470c90602ada963b6864316ac3ac382f189549fb66fb8dc11cac7f41
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 64
  Transaction confirmed (reverted)   Block: 32477232   Gas used:
↳ 25048 (0.08%)

swap.setPoolWeight(100000000000, {'from':miner})
Transaction sent: 0
↳ xbbb7f9d4a6f1cb15e6501445e1472e20a7489d44c0db4270ab797b1c4c84126c
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 65
```

```
  Transaction confirmed   Block: 32477233   Gas used: 43403 (0.14%)

<Transaction '0xbbb7f9d4a6f1cb15e6501445e1472e20a7489d44c0db4270ab797b1c4c84126c'>
>>> swap.swap(owner, 99999999999999999999999999999999, {'from': miner})
Transaction sent: 0xcd797da059f57eabc966385199e1b39647551303ca865e06e2995ab4b5652805
  Gas price: 0.0 gwei   Gas limit: 30000000   Nonce: 66
  Transaction confirmed   Block: 32477234   Gas used: 49852 (0.17%)

<Transaction '0xcd797da059f57eabc966385199e1b39647551303ca865e06e2995ab4b5652805'>
```

The full list can be found below:

- ETHSacred - setAaveInterestsProxy
- Sacred.sol - changeOperator, setFee
- Miner.sol - setMinimumInterests, setAaveInterestFee, setRates, setVerifiers, setSacredTreesContract, setPoolWeight, setAaveInterestProxyContract (The risk of mistake here is reduced due to the functions only be callable by the governance role)
- RewardSwap.sol - setPoolWeight
- SacredProxy.sol - setSacredTreesContract, updateInstance (only callable by the governance role)
- SacredTrees.sol - setSacredProxyContract, setVerifierContract (only callable by governance)

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Halborn recommends that checks be implemented to ensure that the different smart contract functions only accept the expected values.

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by adding additional null value checks where deemed necessary.

3.5 (HAL-05) GAS OPTIMIZATIONS - INFORMATIONAL

Description:

Within the “for loops” in the contract code, the variable `i` is incremented using `i++`. It is known that, in loops, using `++i` costs less gas per iteration than `i++`. This does not only apply to the iterator variable. It also applies to variables declared within the loop code block.

Code Location:

- Miner.sol Lines 172; 192; 406; 424
- SacredProxy.sol Lines 56; 114
- SacredProxyLight.sol Lines 32
- MerkleTreeWithHistory.sol Lines 33; 59
- Sacred.sol Lines 108
- SacredTrees.sol Lines 112; 160

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to use `++i` instead of `i++` to increment the value of an `uint` variable inside a loop. This applies to the iterator variable and to variables declared within the loop code block.

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by using `++i` instead of `i++`.

3.6 (HAL-06) MISSING REENTRANCY PROTECTION - INFORMATIONAL

Description:

It was identified that one function within the Miner contract that made external calls to other contracts, which would then make use of a low-level call function to transfer funds, did not implement a reentrancy guard. While this function does not look exploitable due to additional validation, it is still recommended to implement the reentrancy guard to protect against extreme cases.

Code Location:

Miner.sol - Function withdraw Line #260

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Halborn recommends that the aforementioned function implement the nonReentrant modifier from OpenZeppelin's contracts.

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by adding the nonReentrant modifier to the withdraw function.

3.7 (HAL-07) OUTDATED SOLIDITY VERSION - INFORMATIONAL

Description:

While reviewing the contracts, it was identified that the pragma version in use was outdated. While this might not pose any direct security risks, the versions in use lack certain security features and optimizations which could greatly benefit the protocol.

Some benefits are described below:

- Safemath by default since 0.8.0 (can be more gas efficient than the SafeMath library)
- Low level inline: as of 0.8.2, leads to cheaper gas runtime. This is especially relevant when the contract has small functions. For example, OpenZeppelin libraries typically have a lot of small helper functions, and if they are not built in, they cost an additional 20 to 40 gas due to the 2 extra jump instructions and additional stack operations needed for function calls.
- Optimizer improvements in packed structs: Before 0.8.3, storing packed structs, in some cases, used an additional storage read operation. After EIP-2929, if the slot was already cold, this means an unnecessary stack operations and extra deploy time costs. However, if the slot was already warm, this means additional cost of 100 gas alongside the same unnecessary stack operations and extra deploy time costs.
- Custom errors from 0.8.4, leads to cheaper deploy time cost and run-time cost. Note: the run-time cost is only relevant when the revert condition is met. In short, replace revert strings with custom errors.

Furthermore, having built-in overflow/underflow checks could benefit the contract code to assure that calculations that use unsafe calculations (using `+`, `-`, `/`, `*`) will not be vulnerable under extreme test cases.

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Halborn recommends that the contracts are upgraded to a newer version of Solidity. The recommended version is >= 0.8.10 as these versions include several improvements and enhanced security features such as built-in overflow/underflow protection. It should be noted that this does not cover the casting of variable types.

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by using the 0.8.9 pragma version.

3.8 (HAL-08) REDUNDANT CODE - INFORMATIONAL

Description:

While reviewing the code, it was identified that while a function to calculate the Aave reward amount was included in the Miner contract code, this was not used within the reward function, and instead, the same calculations were replicated. While not posing a security risk, this increases the complexity of the contracts, as well as the gas cost for deployment.

Code Location:

Miner.sol Lines #211-214

Listing 4

```
uint256 interestAmount;
if(totalShareSnapshots[_args.rewardNullifier][0] > 0) {
  interestAmount = totalShareSnapshots[_args.rewardNullifier][1].
↳ mul(_args.apAmount).div(totalShareSnapshots[_args.rewardNullifier
↳ ][0]);
}
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Halborn recommends that the `getAaveInterestAmount` function is used instead of re-writing the code into the `reward` function. This could be rewritten as:

Listing 5

```
uint256 interestAmount = getAaveInterestAmount(args.
↳ rewardNullifier, args.apAmount);
```

Remediation Plan:

SOLVED: The Sacred Finance team solved the issue by using the getAaveInterestAmount function.

3.9 Circuit Review

Halborn reviewed the Circom circuits that Sacred Finance implemented in their code. The Circuits were reviewed for common ZK Snark circuits issues such as the following:

- Integer overflow/underflow
- Non-binding constrains
- Double spending
- Optimizations
- Logic errors

While most of the code was forked from Tornado Cash, there were specific differences which are described below.

Sacred-Anonymity-Mining

Within the **Reward.circom** circuits, an additional field was added, which represents the noteNullifierHash hash. This is later used to invalidate spent notes, and only the hashed value is used. This can be observed within the circuit code when computing the reward nullifier, which is the Poseidon hash of the noteNullifierHash instead of the noteNullifier as in Tornado Cash.

Furthermore, to incorporate Aave lending interests, the following two signals were added:

Listing 6

```
1 signal private input inputAaveInterestAmount
2 signal private input outputAaveInterestAmount
```

This resulted in extra parameters used in the Poseidon hash function.

Furthermore, the following was added:

Listing 7

```
apAmount === rate * (withdrawalBlock - depositBlock)
aaveInterestAmount === (outputAaveInterestAmount -
↳ inputAaveInterestAmount);
```

This ensures that the ap amount is constrained to the rate multiplied by the difference between the withdrawal and deposit block (thus the length of a deposit).

Finally, the second line verifies the aaveInterestAmount input, asserting that the inputs to the circuit respect this equality.

Within **Withdraw.circom**, Sacred Finance includes the AaveInterestAmount as shown below:

Listing 8

```
inputAaveInterestAmount === outputAaveInterestAmount +
↳ aaveInterestAmount;
```

This allows to ensure that the correct reward amount is validated via the ZK proof, upon withdrawal; In more detail, the input to the circuit needs to be equal to the Aave interest left in the contract after withdrawal plus the actual Aave interest amount rewarded to users.

Sacred-Contracts-Eth

The circuits within this repository include a template for MerkleTree computation, as well as a utility to compute the Pedersen hash of the commitment and the nullified for each proof. These circuits are the same as Tornado Cash. These utilities are used within the withdraw circuit.

No security risks were identified within these circuits.

SACRED TREES

Within the BatchTreeUpdate.circom circuits, the only difference with Tornado cash is the CHUNK_TREE_HEIGHT being set to 1. This is correctly

replicated in the sacredTrees.sol smart contract.

THANK YOU FOR CHOOSING

// HALBORN