Smart Contract Security Audit V1

Halborn

https://halborn.com

03/10/2021

https://saferico.com/

business@saferico.com
https://t.me/SFI_ANN

–

Table of Contents

Background

The purpose of the audit was to achieve the following:

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

The information in this report should be used to understand the risk exposure of the smart contract, and as a guide to improve the security posture of the smart contract by remediating the issues that were identified.

Project Information

- **Website**: https://halborn.com
- **Twitter**: https://twitter.com/halbornsecurity
- **Platform**: Binance Smart Chain or ETH network
- **Contract Address**: Not deployed to the main net yet
- **Halborn** is Elite Cybersecurity for Blockchain Organizations // Advanced Penetration Testing

Token Information

- Name: HAL
- Total Supply: 10,000,000,000
- Holders: Not yet
- Total transactions: Not yet

Contracts address deployed to test net (ETH,BSC)

Private Sale contract on Kovan (ETH Test Net)

https://kovan.etherscan.io/address/0xf6faf906f99ccefdac02fc97a15942cca84341df

Private Sale contract on testnet.bsc (BSC Test Net)

https://testnet.bscscan.com/address/0x91448b3ace9f9cedfda80ad42a5c0cf96f479aa7

Halborn (HAL) contract on Kovan (ETH Test Net)

https://kovan.etherscan.io/address/0xd70c8b3754c478337703d6d857851eecefaa201d

Halborn (HAL) contract on testnet.bsc (BSC Test Net)

https://testnet.bscscan.com/token/0xdd8708d16e7238817a0aacdf1dba0b02a69d554e

Executive Summary

According to our assessment, the customer`s solidity smart contract is **Secured**.

Well Secured	
Secured	✓
Poor Secured	
Insecure	

Automated checks are with remix IDE. All issues were performed by the team, which included the analysis of code functionality, manual audit found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the audit overview section. The general overview is presented in the Project Information section and all issues found are located in the audit overview section.

Team found 0 critical, 1 high, 1 medium, 3 low, 0 very low-level issues and 10 notes in all solidity files of the contract

The files:

HalbornInterface.sol

HalbornPool.sol

Migrations.sol

MyToken.sol

Private-Sale.sol

File and Function Level Report

File in Scope:

Contract Name	SHA 256 hash	Contract Address
Private-sale.sol	**270266612f5f0d1077dbbd8e939c6bedc860c8cc20d775a73a0f7f95dba6e58b**	0x91448B3aCE9F9CEdFda80Ad42a5C0cF96f479aA7

- Contract: PrivateSale
- Inherit: Context
- Observation: Passed only security check
- Test Report: Not passed
- Score: Not passed
- Conclusion: Not passed

Function	Test Result	Type/Return Type	Score
TICKET	✓	Read/public	**Passed**
mapping	✓	private	**Passed**
buyTickets	X	Write / public	**Not Passed**
getRefund	X	Write / public	**Not Passed**
Receive	✓	private	**Passed**

File in Scope:

Contracts Name	SHA 256 hash	Contract Address
HalbornInterface.sol HalbornPool.sol Migrations.sol MyToken.sol	**cb020e4c210a9bc234169cc1ebe6505bb5f89db727d543aa4ef5eaf5fe277b2d**	0xDd8708D16E7238817a0AACDF1Dba0B02A69D554e

- Contract: **Halborn**
- Inherit: ERC20, ERC20Burnable, Pausable, Ownable
- Observation: All passed including security check
- Test Report: passed
- Score: passed
- Conclusion: passed

Function	Test Result	Type / Return Type	Score
mapping	✓	private	**Passed**
pause	✓	Write / public	**Passed**
unpause	✓	Write / public	**Passed**
mint	✓	Write / public	**Passed**
transferbyOwner	✓	Write / public	**Passed**
EmergencyDestroy	✓	Write / public	**Passed**
name	✓	Read / public	**Passed**
symbol	✓	Read / public	**Passed**
decimals	✓	Read / public	**Passed**
totalSupply	✓	Read / public	**Passed**
balanceOf	✓	Read / public	**Passed**
owner	✓	Read / public	**Passed**
allowance	✓	Read/public	**Passed**
paused	✓	Read/public	**Passed**

approve	✓	Write / public	Passed
transfer	✓	Write / public	Passed
TransferFrom	✓	Write / public	Passed
burn	✓	Write / public	Passed
burnFrom	✓	Write / public	Passed
increaseAllowance	✓	Write / public	Passed
decreaseAllowance	✓	Write / public	Passed
transferOwnership	✓	Write / public	Passed
renounceOwnership	✓	Write / public	Passed

- Contract: **Migrations**
- Inherit: Context
- Observation: All passed including security check
- Test Report: passed
- Score: passed
- Conclusion: passed

Function	Test Result	Return Type	Score
setCompleted	✓	Public	Passed

- Interface: **HalbornInterface**
- Inherit: Context
- Observation: All passed including security check
- Test Report: passed
- Score: passed
- Conclusion: passed

Function	Test Result	Return Type	Score
deposit	✓	Public	Passed
withdraw	✓	Public	Passed
getContractBalance	✓	Public	Passed

- Contract: **DefiPool**
- Inherit: Context
- Observation: All passed including security check
- Test Report: passed
- Score: passed
- Conclusion: passed

Function	Test Result	Return Type	Score
transfer	✓	Public	**Passed**
mapping	✓	Public	**Passed**
deposit	✓	Public	**Passed**
withdraw	✓	Public	**Passed**
getContractBalance	✓	Public	**Passed**

Issues Checking Status

No.	Issue Description	Checking Status
1	Compiler warnings.	**Passed**
2	Race conditions and Reentrancy. Cross-function race conditions.	**Passed**
3	Possible delays in data delivery.	**Passed**
4	Oracle calls.	**Passed**
5	Front running.	**Passed**
6	Timestamp dependence.	**Passed**
7	Integer Overflow and Underflow.	**Passed**
8	DoS with Revert.	**Passed**
9	DoS with block gas limit.	**Passed**
10	Methods execution permissions.	**Passed**

11	Economy model. If application logic is based on an incorrect economic model, the application would not function correctly and participants would incur financial losses. This type of issue is most often found in bonus rewards systems, Staking and Farming contracts, Vault and Vesting contracts, etc.	**Passed**
12	The impact of the exchange rate on the logic.	**Passed**
13	Private user data leaks.	**Passed**
14	Malicious Event log.	**Passed**
15	Scoping and Declarations.	**Passed**
16	Uninitialized storage pointers.	**Passed**
17	Arithmetic accuracy.	**Passed**
18	Design Logic.	**Passed**

Severity Definitions

Risk Level	**Description**
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to tokens loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have significant impact on smart contract execution, e.g. public access to crucial functions
Medium	Medium-level vulnerabilities are important to fix; however, they can’t lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can’t have significant impact on execution
Note	Lowest-level vulnerabilities, code style violations and info statements can’t affect smart contract execution and can be ignored.

Audit Findings

Critical:

No critical severity vulnerabilities were found.

High:

Issue #1

In detail in MyToken.sol file

Due to missing or insufficient access controls, malicious parties can self-destruct the contract.

```
selfdestruct(_to);
```

Medium:

Issue #1.

In detail

Using an outdated compiler version can be problematic especially if there are publicly disclosed bugs and issues that affect the current compiler version.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.2;
```

Low:

Issue #1.

In detail in HalbornPool.sol

Functions that do not have a function visibility type specified are public by default. This can lead to a vulnerability if a developer forgot to set the visibility and a malicious user is able to make unauthorized or unintended state changes.

```
*/
constructor(address _tokenAddress) {
HAL = IERC20(_tokenAddress);
}
```

Issue #2.

In detail in Migrations.sol

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

```
pragma solidity >=0.4.22 <0.9.0;
```

Issue #3.

In detail in MyToken.sol

Functions that do not have a function visibility type specified are public by default. This can lead to a vulnerability if a developer forgot to set the visibility and a malicious user is able to make unauthorized or unintended state changes.

```
constructor() ERC20("Halborn", "HAL") {

_mint(msg.sender, 10000000000 * 10 ** decimals());
}
```

Very Low:

No very Low severity vulnerabilities were found.

Notes:

Note #1:

In detail in HalbornPool.sol

Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

```
mapping(address => mapping (address => uint256)) allowed;
```

Note #2:

In detail in HalbornPool.sol

Several functions and operators in Solidity are deprecated. Using them leads to reduced code quality. With new major versions of the Solidity compiler, deprecated functions and operators may result in side effects and compile errors.

```
emit Deposit(msg.sender, msg.value, block.timestamp);
```

Note #3:

In detail in HalbornPool.sol

Contracts often need access to the current timestamp to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

```
depositStart[msg.sender] = depositStart[msg.sender] + block.timestamp;
```

Note #4:

In detail in HalbornPool.sol

Contracts often need access to the current timestamp to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

```
emit Deposit(msg.sender, msg.value, block.timestamp);
```

Note #5:

In detail in HalbornPool.sol

Contracts often need access to the current timestamp to trigger time-dependent events. As Ethereum is decentralized, nodes can synchronize time only to some degree. Moreover, malicious miners can alter the timestamp of their blocks, especially if they can gain advantages by doing so. However, miners can't set timestamp smaller than the previous one (otherwise the block will be rejected), nor can they set the timestamp too far ahead in the future. Taking all of the above into consideration, developers can't rely on the preciseness of the provided timestamp.

```
uint time;
depositTime[_user] = block.timestamp - depositStart[_user];
time = depositTime[_user];
```

Note #6:

In detail in HalbornPool.sol

Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can:
- cause an increase in computations (and unnecessary gas consumption)
- indicate bugs or malformed data structures and they are generally a sign of poor code quality
- cause code noise and decrease readability of the code

```
mapping(address => mapping (address => uint256)) allowed;
```

Note #7:

In detail in HalbornPool.sol

Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

```
mapping(address => uint256) balances;
```

Note #8:

In detail in MyToken.sol

Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can:
- cause an increase in computations (and unnecessary gas consumption)
- indicate bugs or malformed data structures and they are generally a sign of poor code quality
- cause code noise and decrease readability of the code

```
mapping(address => uint256) private _balances;
```

Note #9:

In detail in Migrations.sol

Constructors are special functions that are called only once during the contract creation. They often perform critical, privileged actions such as setting the owner of the contract. Before Solidity version 0.4.22, the only way of defining a constructor was to create a function with the same name as the contract class containing it. A function meant to become a constructor becomes a normal, callable function if its name doesn't exactly match the contract name. This behavior sometimes leads to security issues, in particular when smart contract code is re-used with a different name but the name of the constructor function is not changed accordingly.

```
contract Migrations {

address public owner = msg.sender;
uint public last_completed_migration;
modifier restricted() {
require(
msg.sender == owner,

"This function is restricted to the contract's owner"
);

_;
}
function setCompleted(uint completed) public restricted {
last_completed_migration = completed;
}
}
```

Note #10:

In detail in Migrations.sol

Unused variables are allowed in Solidity and they do not pose a direct security issue. It is best practice though to avoid them as they can:
- cause an increase in computations (and unnecessary gas consumption)
- indicate bugs or malformed data structures and they are generally a sign of poor code quality
- cause code noise and decrease readability of the code

```
address public owner = msg.sender;
```

Conclusion

The contracts are written systematically. Team found no critical issues. So, it is good to go for production.

Since possible test cases can be unlimited and developer level documentation (code flow diagram with function level description) not provided, for such an extensive smart contract protocol, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan
Everything.

Security state of the reviewed contract is “secured”.

✓ No mint function.
✓ No volatile code.
✓ Not many high severity issues were found.
✓ Contract Ownership Renounced.

Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice as of the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against the team on the basis of what it says or doesn’t say, or how team produced it, and it is important for you to conduct your own independent investigations before making any decisions. team go into more detail on this in the below disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis, and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Saferico and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers and other representatives) (Saferico s) owe no duty of care towards you or any other person, nor does Saferico make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties or other terms of any kind except as set out in this disclaimer, and Saferico hereby excludes all representations, warranties, conditions and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, Saferico hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against Saferico, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed.