// HALBORN

SIFCHAIN BALANCER

Smart Contract Security Audit

Prepared by: **Halborn**
Visit: Halborn.com

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	01/12/2021	Gabi Urrutia
0.2	Document Edits	01/14/2021	Gabi Urrutia
1.0	Final Version	01/19/2021	Gabi Urrutia

CONTACTS

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

1.1 INTRODUCTION

Sifchain engaged Halborn to conduct a security assessment on their Sifchain Balancer smart contracts. The security assessment was scoped to all the smart contracts and an audit of the security risk and implications regarding the changes introduced by the development team at Sifchain prior to its production release shortly following the assessments deadline.

No major vulnerabilities have been found in the audit. Overall, the smart contract code is extremely well documented, follows a high-quality software development standard, contains many utilities and automation scripts to support continuous deployment / testing / integration, and does NOT contain any obvious exploitation vectors that Halborn was able to leverage within the timeframe of testing allotted.

Though the outcome of this security audit is satisfactory; due to time and resource constraints, only testing and verification of essential properties were performed to achieve objectives and deliverables set in the scope. It is important to remark the use of the best practices for secure smart contract development. Halborn recommends performing further testing to validate extended safety and correctness in context to the whole set of contracts. External threats, such as economic attacks, oracle attacks, and inter-contract functions and calls should be validated for expected logic and state.

1.2 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of the smart contract audit. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts 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 read 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.
- Scanning of solidity files for vulnerabilities, security hotspots, or bugs. (MythX)
- Static Analysis of security for scoped contract, and imported functions. (Slither)
- Smart Contract analysis and automatic exploitation (teEther)
- Symbolic Execution / EVM bytecode security assessment (Manticore)

1.3 SCOPE

Code related to:

- BConst.sol
- BMath.sol
- BNum.sol
- BPool.sol
- Hash.sol
- OwnableWhitelist.sol
- MockRowanToken.sol

Specific commit of contract: Commit ID: a55ce057d0a2cc67859870059c7aaa13468cffbd

OUT-OF-SCOPE:

External contracts, External Oracles, other smart contracts in the repository or imported by Sifchain contracts, economic attacks

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

SECURITY ANALYSIS	RISK LEVEL
FOR LOOP OVER DYNAMIC ARRAY	Informational
STATIC ANALYSIS	Informational
AUTOMATED SECURITY SCAN	Informational
SYMBOLIC EXECUTION SECURITY ASSESSMENT	Informational
SECURITY TESTING EXPLOITATION	Informational

FINDINGS & TECH DETAILS

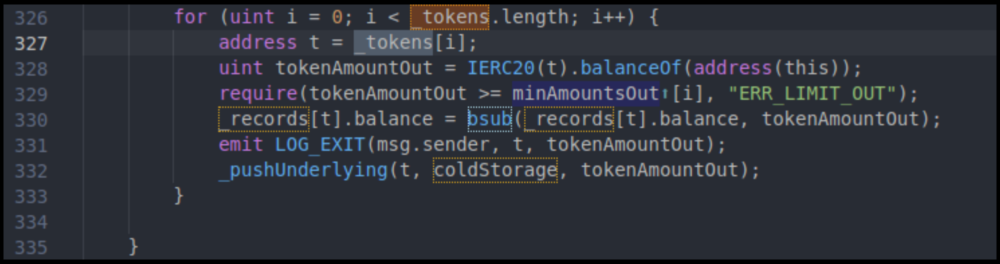
3.1 FOR LOOP OVER DYNAMIC ARRAY - INFORMATIONAL

Description

Calls inside a loop might lead to a denial-of-service attack. The function discovered is a for loop on variable `i` that iterates up to _tokens.length and _users.length variables. If these integers are evaluated at extremely large numbers, or `i` is reset by external calling functions, this can cause a DoS.

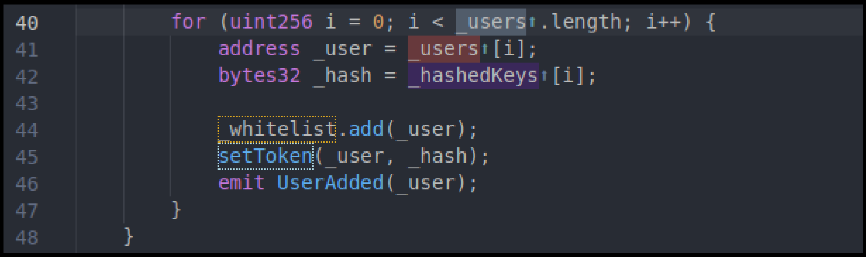
Code Location

BPool.sol Line #327-335



```
326         for (uint i = 0; i < _tokens.length; i++) {
327             address t = _tokens[i];
328             uint tokenAmountOut = IERC20(t).balanceOf(address(this));
329             require(tokenAmountOut >= minAmountsOut[i], "ERR_LIMIT_OUT");
330             _records[t].balance = bsub(_records[t].balance, tokenAmountOut);
331             emit LOG_EXIT(msg.sender, t, tokenAmountOut);
332             _pushUnderlying(t, coldStorage, tokenAmountOut);
333         }
334 
335     }
```

OwnableWhitelist.sol Line #40-48



```
40         for (uint256 i = 0; i < _users.length; i++) {
41             address _user = _users[i];
42             bytes32 _hash = _hashedKeys[i];
43 
44             _whitelist.add(_user);
45             setToken(_user, _hash);
46             emit UserAdded(_user);
47         }
48     }
```

Recommendation:

The possibility of a DoS occurring are reduced because of the applied security measures, such as whitelisting.

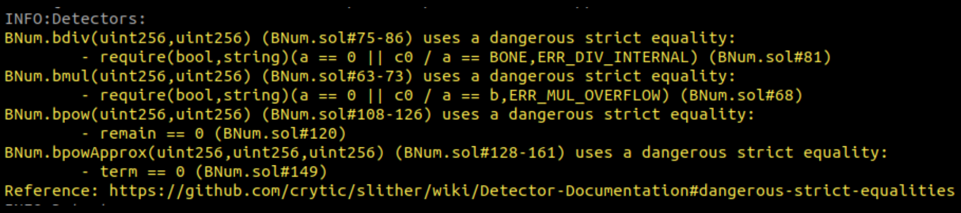
3.2 STATIC ANALYSIS - INFORMATIONAL

Description

certain areas of the scoped contract. Among the tools used was Slither, a Solidity static analysis framework. After Halborn verified all the contracts in the repository and was able to compile them correctly into their abi and binary formats, Slither was run on all contracts.

Results

BPool.sol



```
INFO:Detectors:
BNum.bdiv(uint256,uint256) (BNum.sol#75-86) uses a dangerous strict equality:
        - require(bool,string)(a == 0 || c0 / a == BONE,ERR_DIV_INTERNAL) (BNum.sol#81)
BNum.bmul(uint256,uint256) (BNum.sol#63-73) uses a dangerous strict equality:
        - require(bool,string)(a == 0 || c0 / a == b,ERR_MUL_OVERFLOW) (BNum.sol#68)
BNum.bpow(uint256,uint256) (BNum.sol#108-126) uses a dangerous strict equality:
        - remain == 0 (BNum.sol#120)
BNum.bpowApprox(uint256,uint256,uint256) (BNum.sol#128-161) uses a dangerous strict equality:
        - term == 0 (BNum.sol#149)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
```

Analysis

No major vulnerabilities have been found in the contracts. The possible dangerous strict equality in BPool.sol can be considered as a false positive, because it is not used to check the ERC20 or ETH balance.

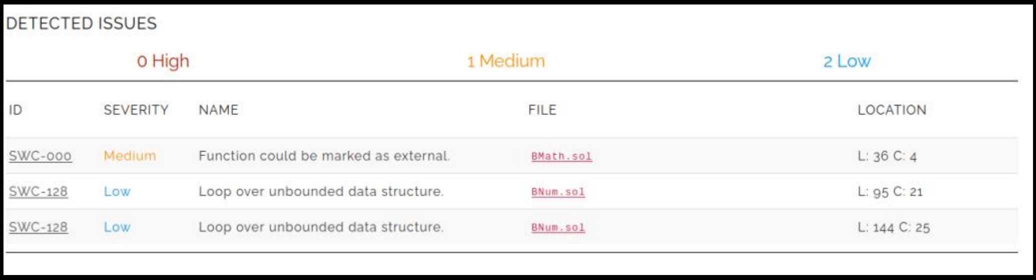
3.3 AUTOMATED SECURITY SCAN - INFORMATIONAL

Description

Halborn used automated security scanners to assist with detection of well-known security issues, and to identify low-hanging fruit on the targets for this engagement. Among the tools used was MythX, a security analysis service for Ethereum smart contracts. MythX performed a scan on the testers machine and sent the compiled results to the analyzers to locate any vulnerabilities.

Results

Mainly, all issues founded by MythX are related to mark functions as external instead of public and possible loops.



DETECTED ISSUES

0 High | 1 Medium | 2 Low

ID	SEVERITY	NAME	FILE	LOCATION
SWC-000	Medium	Function could be marked as external.	BMath.sol	L: 36 C: 4
SWC-128	Low	Loop over unbounded data structure.	BNum.sol	L: 95 C: 21
SWC-128	Low	Loop over unbounded data structure.	BNum.sol	L: 144 C: 25

Recommendation

Consider as much as possible declaring external variables instead of public variables. As for best practices, you should use external if you expect that the function will only ever be called externally and use public if you need to call the function internally. Mainly, marking the function as external can save gas.

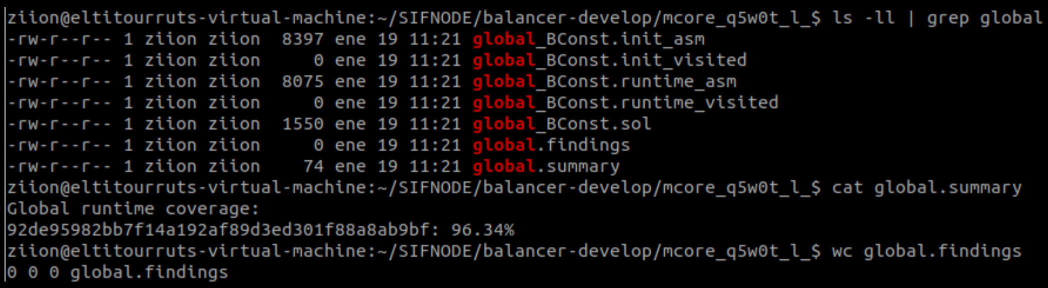
3.4 SYMBOLIC EXECUTION SECURITY ASSESSMENT - INFORMATIONAL

Description

The tool used to perform the symbolic execution for analyzing Smart Contract is Manticore. In general, obtaining all possible states of a Smart Contract by symbolic execution is the main goal for using this tool. Briefly, concrete values are replaced by symbolic values and variables which are used to generate path conditions which are logic formulas that represent the state of the program and the transformations between program states. Some detectors were used in the audit to find some vulnerabilities such as: Integer Overflow, Simple and Advance Reentrancy and Delegate calls.

Results

After performing the symbolic execution, vulnerabilities are placed in global.findings file. No vulnerabilities were founded in any contract.



```
ziion@eltitourruts-virtual-machine:~/SIFNODE/balancer-develop/mcore_q5w0t_l_$ ls -ll | grep global
-rw-r--r-- 1 ziion ziion  8397 ene 19 11:21 global_BConst.init_asm
-rw-r--r-- 1 ziion ziion     0 ene 19 11:21 global_BConst.init_visited
-rw-r--r-- 1 ziion ziion  8075 ene 19 11:21 global_BConst.runtime_asm
-rw-r--r-- 1 ziion ziion     0 ene 19 11:21 global_BConst.runtime_visited
-rw-r--r-- 1 ziion ziion  1550 ene 19 11:21 global_BConst.sol
-rw-r--r-- 1 ziion ziion     0 ene 19 11:21 global.findings
-rw-r--r-- 1 ziion ziion    74 ene 19 11:21 global.summary
ziion@eltitourruts-virtual-machine:~/SIFNODE/balancer-develop/mcore_q5w0t_l_$ cat global.summary
Global runtime coverage:
92de95982bb7f14a192af89d3ed301f88a8ab9bf: 96.34%
ziion@eltitourruts-virtual-machine:~/SIFNODE/balancer-develop/mcore_q5w0t_l_$ wc global.findings
0 0 0 global.findings
```

3.5 SECURITY TESTING EXPLOITATION - INFORMATIONAL

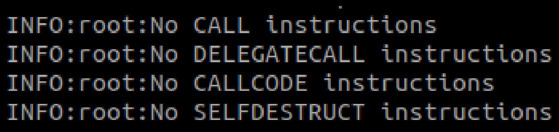
Description:

teEther is a tool to perform analysis and automatic exploitation over smart contracts. teEther try to exploit the most common vulnerabilities such as DELEGATE CALL and SELFDESTRUCT on Smart Contracts in the Bytecode level.

Results:

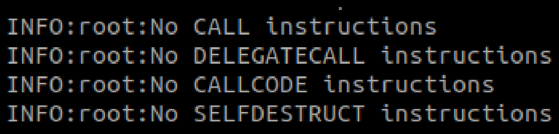
The same result was obtained in all smart contracts. No vulnerabilities were founded in smart contracts.

- BConst.sol



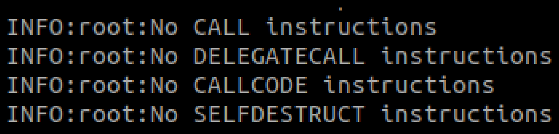
```
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
```

- BMath.sol



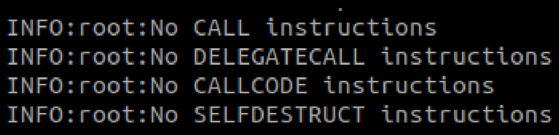
```
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
```

- BNum.sol



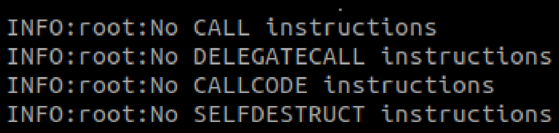
```
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
```

- BPool.sol



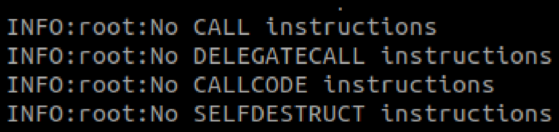
```
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
```

- Hash.sol



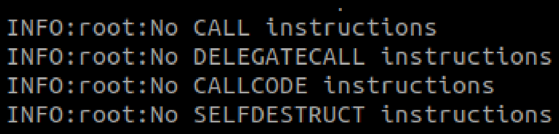
```
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
```

- OwnableWhitelist.sol



```
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
```

- MockRowanToken.sol



```
INFO:root:No CALL instructions
INFO:root:No DELEGATECALL instructions
INFO:root:No CALLCODE instructions
INFO:root:No SELFDESTRUCT instructions
```

THANK YOU FOR CHOOSING

// HALBORN