Finance.Vote - GatedMerkleIdentity and Incinerator

Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **June 11th-June 18th, 2021**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	06/14/2021	Gabi Urrutia
0.2	Document Edits	06/18/2021	Gokberk Gulgun
1.0	Final Version	06/19/2021	Gabi Urrutia
1.1	Remediation Plan	06/25/2021	Gokberk Gulgun

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
Gokberk Gulgun	Halborn	Gokberk.Gulgun@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Finance.Vote engaged Halborn to conduct a security assessment on their Smart contracts beginning on June 11th, 2021 and ending June 18th, 2021. The security assessment was scoped to the smart contracts GatedMerkleIdentity.sol and Incinerator.sol. An audit of the security risk and implications regarding the changes introduced by the development team at Finance.Vote prior to its production release shortly following the assessments deadline.

1.2 AUDIT SUMMARY

The team at Halborn was provided a week timeframe for the engagement and assigned two full time security engineers to audit the security of the smart contract. The security engineers are blockchain and smart contract security experts, with experience in advanced penetration testing, smart contract hacking, and have a deep knowledge in multiple blockchain protocols.

The purpose of this audit to achieve the following:

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

In summary, Halborn identified few security risks, and 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.

1.3 TEST APPROACH & METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to 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.)
- Testnet deployment (Truffle, Ganache.)

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. It's quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that was 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.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL

9 - 8 - HIGH

7 - 6 - MEDIUM

5 - 4 - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

IN-SCOPE:

GatedMerkleIdentity.sol - Commit 7e1f247d7640edfe4bf68140328dd087c95c4700

Incinerator.sol - Commit 169e37393e4bb5eb81b4bd21cb9be61f932140af

GatedMerkleIdentity.sol - Fixed Commit ID 9433667973e86ebd76f3d3fe7d996086b73c2c0e

Incinerator.sol - Fixed Commit ID 9433667973e86ebd76f3d3fe7d996086b73c2c0e

OUT-OF-SCOPE:
Other smart contracts in the repository, external libraries and economics attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	3	4

LIKELIHOOD

IMPACT

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

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL01 - MISSING ADDRESS VALIDATION	Low	RISK ACCEPTED: 06/25/2021
HAL02 - MISSING EVENT HANDLER	Low	RISK ACCEPTED: 06/25/2021
HAL03 - IGNORED RETURN VALUES	Low	RISK ACCEPTED: 06/25/2021
HAL04 - MULTIPLE INCINERATE ON THE WITHDRAW PROGRESS	Informational	SOLVED: 06/25/2021
HAL05 - MISSING ARRAY ELEMENT CHECK	Informational	SOLVED: 06/25/2021
HAL06 - FOR LOOP OVER DYNAMIC ARRAY	Informational	RISK ACCEPTED: 06/25/2021
HAL07 - POSSIBLE MISUSE OF PUBLIC FUNCTIONS	Informational	SOLVED: 06/25/2021

FINDINGS & TECH DETAILS

3.1 (HAL-01) MISSING ADDRESS VALIDATION - LOW

Description:

The GatedMerkleIdentity.sol and Incinerator.sol contracts lack a safety check inside their constructors and functions. Setters of address type parameters should include a zero-address check. Otherwise, contract functionality may become inaccessible, or tokens could be burnt forever.

Code Location:

GatedMerkleIdentity.sol Line #~45

Listing 1: GatedMerkleIdentity.sol (Lines)

```
45     function setGateParameters(address _incinerator, address
           _burnToken, uint _ethCost) public managementOnly {
46         incinerator = IIncinerator(_incinerator);
47         burnToken = _burnToken;
48         ethCost = _ethCost;
49     }
```

GatedMerkleIdentity.sol Line #~52

Listing 2: GatedMerkleIdentity.sol (Lines)

```
52      // change the management key
53     function setManagement(address newMgmt) external
           managementOnly {
54         address oldMgmt =  management;
55         management = newMgmt;
56         emit ManagementUpdated(oldMgmt, newMgmt);
57     }
58
```

Incinerator.sol Line #~36

Listing 3: Incinerator.sol (Lines)

```
    function setManagement(address newMgmt) external
        managementOnly {
        address oldMgmt =  management;
        management = newMgmt;
        emit ManagementUpdated(oldMgmt, newMgmt);
    }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Add proper address validation when assigning a value to a variable from user-supplied data. Better yet, address white-listing/black-listing should be implemented in relevant functions if possible.

For example:

Listing 4: Modifier.sol (Lines 2,3,4)

```
modifier validAddress(address addr) {
    require(addr != address(0), "Address cannot be 0x0");
    require(addr != address(this), "Address cannot be contract");
    _;
 }
```

Remediation Plan:

RISK ACCEPTED: Finance.Vote Team decided to continue without address validation.

3.2 (HAL-02) MISSING EVENT HANDLER - LOW

Description:

In the GatedMerkleIdentity.sol contract, some functions do not emit logging events. Events are a method of informing the transaction initiator about the actions taken by the called function. Logs are used for event subscriptions and are indexed. It is not possible to search for a specific event unless the contract logs it.

Code Location:

GatedMerkleIdentity.sol Line #~45

Listing 5: GatedMerkleIdentity.sol (Lines)

```
function setGateParameters(address _incinerator, address
   _burnToken, uint _ethCost) public managementOnly {
     burnToken = _burnToken;
     ethCost = _ethCost;
 }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

Where appropriate, declare events at the end of the function. Clients can use events to detect the end of the operation and aid in searching for the specific activity.

For example:

Listing 6: GatedMerkleIdentity.sol (Lines)

```
function setGateParameters(address _incinerator, address
    _burnToken, uint _ethCost) public managementOnly {
        burnToken = _burnToken;
        ethCost = _ethCost;
        emit SetGateEvent(burnToken,ethCost);
 }
```

Remediation Plan:

RISK ACCEPTED: Finance.Vote Team decided to continue without event emitting.

3.3 (HAL-03) IGNORED RETURN VALUES - LOW

Description:

The return value of an external call is not stored in a local or state variable. In the contract Incinerator.sol, there are a few instances where external methods are called and the return value (bool) is ignored.

Code Location:

GatedMerkleIdentity.sol Line #~63

Listing 7: GatedMerkleIdentity.sol (Lines 69)

```
63     function withdraw(uint merkleIndex, bytes32[] memory proof)
           external payable {
64         require(msg.value >= ethCost, 'Please send more ETH');
65         require(verifyEntitled(merkleRoots[merkleIndex], msg.
               sender, proof), "The proof could not be verified.");
66
67         // burn token cost
68         if (msg.value > 0) {
69             incinerator.incinerate{value: msg.value}(burnToken);
70         }
71
72         // note that this effectively prevents inclusion of the
               same address in multiple merkle roots
73         require(! withdrawn[msg.sender], "You have already
               withdrawn your nft.");
74
75         withdrawn[msg.sender] = true;
76
77         token.createIdentityFor(msg.sender);
78
79     }
80
```

Incinerator.sol Line #~43

Listing 8: Incinerator.sol (Lines)

```
    // buy tokens at market rate and burn them
    function incinerate(address tokenAddr) external payable {
        uint amountOutMin = 0;
        address[] memory path = new address[](2);
        path[0] = WETH;
        path[1] = tokenAddr;

        address burnAddress = address(0);
        uint deadline = block.timestamp + 1;
        uint[] memory amounts = router.swapExactETHForTokens{value
            : msg.value}(amountOutMin, path, burnAddress, deadline)
            ;
        emit TokensIncinerated(tokenAddr, amounts[1]);
    }

```

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

Add a return value check to avoid an unexpected crash of the contract. Return value checks provide better exception handling.

Remediation Plan:

RISK ACCEPTED: Finance.Vote Team decided to continue without checking return values.

3.4 (HAL-04) MULTIPLE INCINERATE ON THE WITHDRAW PROGRESS - INFORMATIONAL

Description:

In the GatedMerkleIdentity.sol contract, a user can only withdraw when providing the correct merkleIndex and proof. However, repeated incinerate calls can occur if the user attempts multiple withdrawals because incinerate is called before checking for a previous withdraw.

Code Location:

GatedMerkleIdentity.sol Line #~63

Listing 9: GatedMerkleIdentity.sol (Lines 69)

```
63    function withdraw(uint merkleIndex, bytes32[] memory proof)
          external payable {
64        require(msg.value >= ethCost, 'Please send more ETH');
65        require(verifyEntitled(merkleRoots[merkleIndex], msg.
              sender, proof), "The proof could not be verified.");
66
67        // burn token cost
68        if (msg.value > 0) {
69            incinerator.incinerate{value: msg.value}(burnToken);
70        }
71
72        // note that this effectively prevents inclusion of the
              same address in multiple merkle roots
73        require(!withdrawn[msg.sender], "You have already
              withdrawn your nft.");
74
75        withdrawn[msg.sender] = true;
76
77        token.createIdentityFor(msg.sender);
78
79    }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

The workflow should be checked according to the incinerator progress. As an solution, require(! withdrawn[msg.sender] line should move to the top of the function.

Listing 10: GatedMerkleIdentity.sol (Lines 69)

```
63     function withdraw(uint merkleIndex, bytes32[] memory proof)
           external payable {
64         require(msg.value >= ethCost, 'Please send more ETH');
65         require(verifyEntitled(merkleRoots[merkleIndex], msg.
               sender, proof), "The proof could not be verified.");
66         require(! withdrawn[msg.sender], "You have already
               withdrawn your nft.");
67
68         // burn token cost
69         if (msg.value > 0) {
70             incinerator.incinerate{value: msg.value}(burnToken);
71         }
72
73         // note that this effectively prevents inclusion of the
               same address in multiple merkle roots
74
75         withdrawn[msg.sender] = true;
76
77         token.createIdentityFor(msg.sender);
78
79     }
```

Remediation Plan:

SOLVED: Finance.Vote Team changed the location of modifier.

Listing 11: GatedMerkleIdentity.sol (Lines 67)

```
63    function withdraw(uint merkleIndex, bytes32[] memory proof) external payable {
64        require(msg.value >= ethCost, 'Please send more ETH');
65        require(verifyEntitled(merkleRoots[merkleIndex], msg.sender, proof), "The proof could not be verified.");
66        // note that this effectively prevents inclusion of the same address in multiple merkle roots
67        require(!withdrawn[msg.sender], "You have already withdrawn your nft.");
68
69        withdrawn[msg.sender] = true;
70
71        // burn token cost
72        if (msg.value > 0) {
73            incinerator.incinerate{value: msg.value}(burnToken);
74        }
75
76
77        token.createIdentityFor(msg.sender);
78
79    }
```

3.5 (HAL-05) MISSING ARRAY ELEMENT CHECK - INFORMATIONAL

Description:

The verifyProof function in the GatedMerkleIdentity.sol contract discards the first bytes32 element in the user-provided proof array.

Code Location:

GatedMerkleIdentity.sol Line #~45

Listing 12: GatedMerkleIdentity.sol (Lines 99)

```
96      function verifyProof(bytes32 root, bytes32 leaf, bytes32[]
            memory proof) public pure returns (bool) {
97          bytes32 currentHash = leaf;
98
99          for (uint i = 1; i < proof.length; i += 1) {
100             currentHash = parentHash(currentHash, proof[i]);
101         }
102
103         return currentHash == root;
104     }
```

Example Inputs

Listing 13

```
function withdraw(uint merkleIndex, bytes32[] memory proof)
merkleIndex: 3 - 0xd778161eb220a7790fc703d428fb65c50de8c9fa37ea6a5cd8ae6d1513ee7a3f
proof : ["0x0000000000000000000000000000000000000000000000000000000000000000","0x0d015084d5b21f5040c727e73691438a3d401c651c8c4f5d0f6dc480f57ece5c"]
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to check the proof array's first element. However, if this is the intended behavior of the function, the first element should not be considered in the proof calculation.

Remediation Plan:

SOLVED: Finance.Vote Team checked proof array's first element.

Listing 14: GatedMerkleIdentity.sol (Lines 99)

```
96     function verifyProof(bytes32 root, bytes32 leaf, bytes32[]
           memory proof) public pure returns (bool) {
97         bytes32 currentHash = leaf;
98
99         for (uint i = 0; i < proof.length; i += 1) {
100            currentHash = parentHash(currentHash, proof[i]);
101        }
102
103        return currentHash == root;
104    }
105
```

3.6 (HAL-06) FOR LOOP OVER DYNAMIC ARRAY - INFORMATIONAL

Description:

When smart contracts are deployed or functions inside them are called, the execution of these actions always requires a certain amount of gas, based on how much computation is needed to complete them. The Ethereum network specifies a block gas limit and the sum of all transactions included in a block cannot exceed the threshold.

Programming patterns that are harmless in centralized applications can lead to Denial of Service conditions in smart contracts when the cost of executing a function exceeds the block gas limit. Modifying an array of unknown size, that increases in size over time, can lead to such a Denial of Service condition.

A situation in which the block gas limit can be an issue is in sending funds to an array of addresses. Even without any malicious intent, this can easily go wrong. Just by having too large an array of users to pay can max out the gas limit and prevent the transaction from ever succeeding.

Example Location:

Listing 15: GatedMerkleIdentity.sol (Lines 99)

```
96    function verifyProof(bytes32 root, bytes32 leaf, bytes32[]
          memory proof) public pure returns (bool) {
97        bytes32 currentHash = leaf;
98
99        for (uint i = 1; i < proof.length; i += 1) {
100           currentHash = parentHash(currentHash, proof[i]);
101       }
102
103       return currentHash == root;
104   }
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Actions that require looping across the entire data structure should be avoided. If you absolutely must loop over an array of unknown size, then you should plan for it to potentially take multiple blocks, and therefore require multiple transactions. As an other solution, the function should be marked as an internal.

Remediation Plan:

RISK ACCEPTED: Finance.Vote Team decided to continue without checking proof array size.

3.7 (HAL-07) POSSIBLE MISUSE OF PUBLIC FUNCTIONS - INFORMATIONAL

Description:

In public functions, array arguments are immediately copied to memory, while external functions can read directly from calldata. Reading calldata is cheaper than memory allocation. Public functions need to write the arguments to memory because public functions may be called internally. Internal calls are passed internally by pointers to memory. Thus, the function expects its arguments being located in memory when the compiler generates the code for an internal function.

Code Location:

Listing 16: GatedMerkleIdentity.sol (Lines)

```
function verifyProof(bytes32 root, bytes32 leaf, bytes32[]
    memory proof) public pure returns (bool) {
    bytes32 currentHash = leaf;

    for (uint i = 1; i < proof.length; i += 1) {
        currentHash = parentHash(currentHash, proof[i]);
    }

    return currentHash == root;
}
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

Consider declaring external variables instead of public variables. A best practice is to use external if expecting a function to only be called externally and public if called internally. Public functions are always accessible, but external functions are only available to external callers.

Remediation Plan:

SOLVED: Finance.Vote Team provided the function is called internally and externally.

MANUAL TESTING

During the manual testing multiple questions where considered while evaluation each of the defined functions:

- Can it be re-called changing admin/roles and permissions?
- Can an externally controlled contract recursively call functions during execution? (Re-entrancy)
- Do we control sensitive or vulnerable parameters?
- Does the function check for boundaries on the parameters and internal values? Bigger than zero or equal? Argument count, array sizes, integer truncation.
- Are there any hash collisions in the merkle proof calculation?
- Can an attacker withdraw multiple times?

4.1 Access Control Test

First, all contract's access-control policies were evaluated. During the tests, the following functions were reachable only by the management address.

Listing 17

```
function setGateParameters(address _incinerator, address
    _burnToken, uint _ethCost)
function setManagement(address newMgmt)
function addMerkleRoot(bytes32 newRoot)
```

According to policies, No issues have been found on the dynamic analysis.

Figure 1

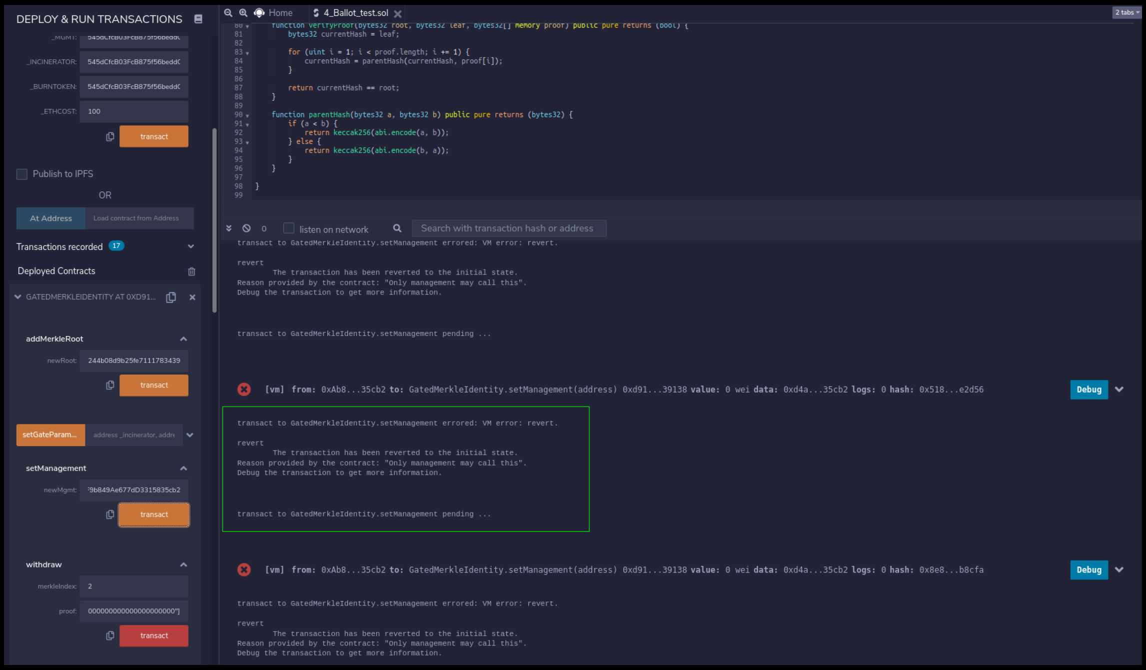

Figure 1: Testing Access Control Policy

4.2 Merkle Tree Test

Next, the Merkle Proof functionality was examined. In the merkle proof verification, Msg.sender was used for a leaf on the tree. Figure 2 It has been observed that, the proof array's first element was not included in the hash calculation. Therefore, It is marked as an informational issue in the report. (HAL04 - MISSING ARRAY ELEMENT CHECK)

Listing 18: GatedMerkleIdentity.sol (Lines 99)

```
96      function verifyProof(bytes32 root, bytes32 leaf, bytes32[]
            memory proof) public pure returns (bool) {
97          bytes32 currentHash = leaf;
98
99          for (uint i = 1; i < proof.length; i += 1) {
100             currentHash = parentHash(currentHash, proof[i]);
101         }
102
103         return currentHash == root;
104     }
```

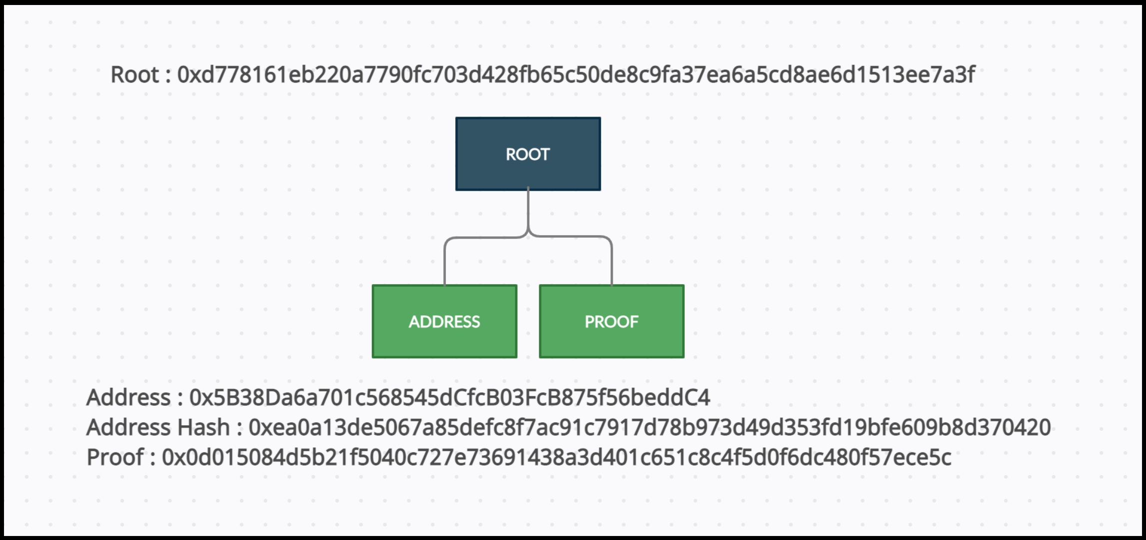

Figure 2: Testing Merkle Proof

4.3 Multiple Withdraw Test

Then the withdraw progress was tested. The Halborn Team tried to manipulate withdraw progress. Figure 3 From the test results, It was observed that the user could not create an identity multiple times. On the other hand, the user could incinerate multiple times due to check statements being completed after an incinerator call.

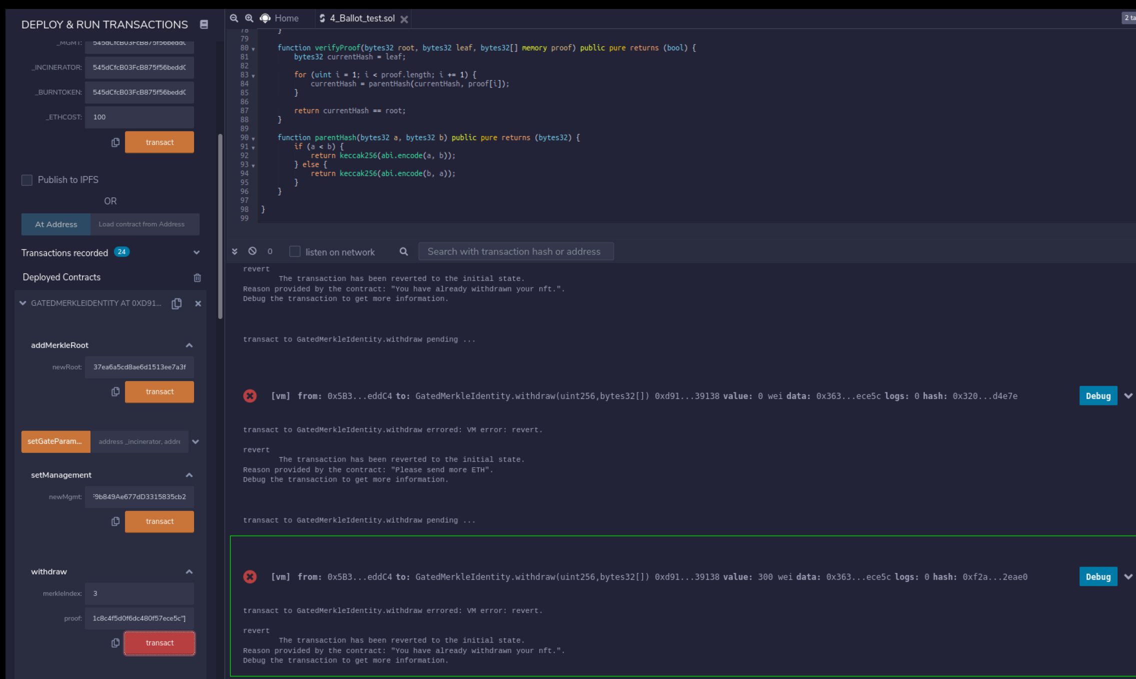

Figure 3: Testing Multiple Withdraws

AUTOMATED TESTING

5.1 STATIC ANALYSIS REPORT

Description:

Halborn used automated testing techniques to enhance coverage of 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 the all-scoped contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.

Results:

GatedMerkleIdentity.sol

```
INFO:Detectors:
GatedMerkleIdentity.constructor(IVotingIdentity,bytes32,address,address,address,uint256)._mgmt (identity/GatedMerkleIdentity.sol#34) lacks a zero-check on :
                - management = _mgmt (identity/GatedMerkleIdentity.sol#40)
GatedMerkleIdentity.setGateParameters(address,address,uint256)._burnToken (identity/GatedMerkleIdentity.sol#45) lacks a zero-check on :
                - burnToken = _burnToken (identity/GatedMerkleIdentity.sol#47)
GatedMerkleIdentity.setManagement(address).newMgmt (identity/GatedMerkleIdentity.sol#52) lacks a zero-check on :
                - management = newMgmt (identity/GatedMerkleIdentity.sol#54)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Reentrancy in GatedMerkleIdentity.withdraw(uint256,bytes32[]) (identity/GatedMerkleIdentity.sol#63-79):
        External calls:
        - incinerator.incinerate{value: msg.value}(burnToken) (identity/GatedMerkleIdentity.sol#69)
        State variables written after the call(s):
        - withdrawn[msg.sender] = true (identity/GatedMerkleIdentity.sol#75)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Pragma version0.7.4 (identity/GatedMerkleIdentity.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version0.7.4 (interfaces/IIncinerator.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version0.7.4 (interfaces/IVotingIdentity.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
solc-0.7.4 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Parameter GatedMerkleIdentity.setGateParameters(address,address,uint256)._incinerator (identity/GatedMerkleIdentity.sol#45) is not in mixedCase
Parameter GatedMerkleIdentity.setGateParameters(address,address,uint256)._burnToken (identity/GatedMerkleIdentity.sol#45) is not in mixedCase
Parameter GatedMerkleIdentity.setGateParameters(address,address,uint256)._ethCost (identity/GatedMerkleIdentity.sol#45) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Slither:identity/GatedMerkleIdentity.sol analyzed (3 contracts with 72 detectors), 11 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Incinerator.sol

AUTOMATED TESTING

```
INFO:Detectors:
Incinerator.constructor(address,address).mgmt (bank/Incinerator.sol#23) lacks a zero-check on :
                - management = mgmt (bank/Incinerator.sol#25)
Incinerator.setManagement(address).newMgmt (bank/Incinerator.sol#36) lacks a zero-check on :
                - management = newMgmt (bank/Incinerator.sol#38)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-address-validation
INFO:Detectors:
Reentrancy in Incinerator.incinerate(address) (bank/Incinerator.sol#43-53):
        External calls:
        - amounts = router.swapExactETHForTokens{value: msg.value}(amountOutMin,path,burnAddress,deadline) (bank/Incinerator.sol#51)
        Event emitted after the call(s):
        - TokensIncinerated(tokenAddr,amounts[1]) (bank/Incinerator.sol#52)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Pragma version0.7.4 (SafeMathLib.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version0.7.4 (bank/Incinerator.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
Pragma version0.7.4 (interfaces/IUniswapRouter.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.11
solc-0.7.4 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
times(uint256,uint256) should be declared external:
        - SafeMathLib.times(uint256,uint256) (SafeMathLib.sol#6-10)
minus(uint256,uint256) should be declared external:
        - SafeMathLib.minus(uint256,uint256) (SafeMathLib.sol#12-15)
plus(uint256,uint256) should be declared external:
        - SafeMathLib.plus(uint256,uint256) (SafeMathLib.sol#17-21)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:bank/Incinerator.sol analyzed (3 contracts with 72 detectors), 10 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

5.2 AUTOMATED SECURITY SCAN RESULTS

Description:

Halborn used automated security scanners to assist with detection of well known security issues, and identify low-hanging fruit on the scoped contract targeted 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 MythX to locate any vulnerabilities. Security Detections are only in scope, and the analysis was pointed towards issues with the GatedMerkleIdentity.sol and Incinerator.sol.

Results:

GatedMerkleIdentity.sol

No issues were found by MythX.

Incinerator.sol

No issues were found by MythX.

THANK YOU FOR CHOOSING

HALBORN