BANCOR LIQUIDITYPOOL V2

Smart Contract Security Audit

Prepared by: **Halborn**
Date of Engagement: **08.16-17.2020**
Visit: [Halborn.com](https://halborn.com)

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	8/16/2020	Steven Walbroehl
0.2	Document Edits	8/17/2020	Steven Walbroehl
1.0	Document Final	8/17/2020	Steven Walbroehl

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
Nishit Majithia	Halborn	Nishit.Majithia@halborn.com

1.1 INTRODUCTION

Bancor engaged Halborn to conduct a security assessment on their oracle optimization smart contract beginning on August 14, 2020 and ending August 17th, 2020. The security assessment was scoped to the contract LiquidityPoolV2Converter.sol , and audit the security risk and implications regarding the changes introduced by the development team at Bancor prior to its production release shortly following the assessments deadline.

The contract scoped in this assessment introduces new functionality for the Bancor on-chain liquidity protocol, which is a DeFi platform that enables automated, decentralized exchange on Ethereum and across blockchains. The entire set of smart contracts create a protocol that is designed to pool liquidity and perform peer-to-contract trades in a single transaction with no counterparty. Users of the Bancor DeFi platform add liquidity to automated market makers in exchange for trading fees.

In particular, the LiquidityPoolV2Converter smart contract tested brings changes in the arithmetic and calculation of the weights/rates and averages in the assets within the liquidity pool. The formulas are dynamic, and are a composed of elements in the price oracle’s rate (provided by Chainlink TKN and BNT price), the arbitrage factor, the effective weight/rate of a token staked in the liquidity pool, spot price, and target weight/rate. Arbitrage incentives are intended to drive the price in the pool to nominal values to allow liquidity providers to get back tokens they stake at the same levels when originally staked.

Due to the importance of the dynamic arithmetic involved in the formulas controlling the balancing mechanics and calculations within the liquidity pool, the Halborn security team spent time manually reviewing and testing arithmetic properties within the source code, along with identifying any possible vulnerabilities in the new code due to common exploitation tactics.

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 two-day timeframe of testing allotted. The most significant observation made in the security assessment is in regards to the current solc version utilized in the Bancor smart contracts (0.4.26) The pragma version is used as a directive for solidity the contracts implemented version of solidity code. With the latest pragma being 0.7 at the time of this audit, the version in use on the scope of this audit has been superseded, and is several versions behind the recommended level. When discussing this with the development team at Bancor, it was stated that a project is soon underway to upgrade the smart contracts to a modern level.

Though the outcome of this security audit is satisfactory; due to time and resource constraints, only testing and verification of essential properties related to the LiquidityPoolV2Converter was performed to achieve objectives and deliverables set in the scope. Halborn recommends performing further testing to validate extended safety and correctness in context to the whole liquidity pool 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 in regards 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, purpose, and use of Bancor and its implementation of Liquidity Pools.
- Smart Contract manual code read and walkthrough.
- Graphing out functionality and contract calls/connectivity/functions within the oracle optimization Github repository. (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)
- Automated exploitation of deployed contract vulnerabilities (karl)
- **Symbolic Execution / EVM bytecode security assessment (limited-time)**

1.3 SCOPE

IN-SCOPE:
New code related to the LiquidityPoolV2Converter smart contract.

OUT-OF-SCOPE:
External contracts, External Oracles, other smart contracts in the oracle optimization repository or within the Bancor protocol liquidity pool, economic attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	1	5

SECURITY ANALYSIS	RISK LEVEL
DEPRECATED PRAGMA VERSION OF SOLC	Medium
BLOCK TIME STAMP ALIAS USAGE	Low
DIVIDE BEFORE MULTIPLY	Low
EXTERNAL FUNCTION CALLS WITHIN LOOP	Low
EXPLOITATION OF TESTNET DEPLOYED CONTRACT	Very Low
STRICT EQUALITIES	Very Low
STATIC ANALYSIS REPORT	Informational
AUTOMATED SECURITY SCAN REPORT	Informational
IN LINE ASSEMBLY USAGE	Informational

FINDINGS & TECH DETAILS

3.1 DEPRICATED PRAGMA VERSION OF SOLC - MEDIUM

Description

The current version in use for Bancor is pragma 0.4.26 While this version is still functional, and most security issues safely implemented by mitigating the Bancor contracts with other utility contracts such as SafeMath.sol and ReentrancyGuard.sol, the risk to the long term sustainability and integrity of the solidity code increases. At the time of this audit, the current version is already at 0.7 The newer versions provide features that provide checks and accounting, as well as prevent insecure use of code.

The follow list identifies areas of code improvements, areas in the contract where it may have been identified, and deprecated functionality that will need to be refactored into the existing version in order to come to the latest pragma level.

Deprecated or Upgraded Items

UPDATED OR DEPRECATED FEATURE DETAIL	VERSION RELEASED	IMPACTED CODE
Functions		
Function callcode is now disallowed (in favor of delegatecall). It is still possible to use it via inline assembly.	v0.5.0	None Located in Manual Audit
suicide is now disallowed (in favor of selfdestruct).	v0.5.0	None Located in Manual Audit
sha3 is now disallowed (in favor of keccak256).	v0.5.0	None Located in Manual Audit
throw is now disallowed (in favor of revert, require and assert).	v0.5.0	None Located in Manual Audit
The try/catch statement allows you to react on failed external calls.	v0.6.0	None Located in Manual Audit
Conversions from address to address payable are now possible via	v0.6.0	z

payable(x), where x must be of type address		
Yul and Inline Assembly have a new statement called leave that exits the current function.	v0.6.0	None Located in Manual Audit
The state mutability of functions can now be restricted during inheritance: Functions with default state mutability can be overridden by pure and view functions while view functions can be overridden by pure functions	v0.7.0	None Located in Manual Audit
Disallow virtual for library functions. This is a new type checker.	v0.7.0	None Located in Manual Audit
Multiple events with the same name and parameter types in the same inheritance hierarchy are disallowed.	v0.7.0	None Located in Manual Audit

Conversions

Explicit and implicit conversions from decimal literals to bytesXX types is now disallowed.	v0.5.0	None Located in Manual Audit
Explicit and implicit conversions from hex literals to bytesXX types of different size is now disallowed.	v0.5.0	None Located in Manual Audit
Exponentiation and shifts of literals by non-literals will always use either the type uint256 (for non-negative literals) or int256 (for negative literals) to perform the operation	v0.7.0	None Located in Manual Audit

Literals and Suffixes

The unit denomination years is now disallowed due to complications and confusions about leap years. (since v0.5.0)	v0.5.0	None Located in Manual Audit
Trailing dots that are not followed by	v0.5.0	None Located in Manual

a number are now disallowed. (since v0.5.0)		Audit
Combining hex numbers with unit denominations (e.g. 0x1e wei) is now disallowed. (since v0.5.0)	V0.5.0	None Located in Manual Audit
The prefix 0X for hex numbers is disallowed, only 0x is possible. (since v0.5.0)	v0.5.0	None Located in Manual Audit

Inline Assembly

Disallow . (a period) in user-defined function and variable names in inline assembly. It is still valid if you use Solidity in Yul-only mode.	v0.7.0	None Located in Manual Audit
Slot and offset of storage pointer variable x are accessed via x.slot and x.offset instead of x_slot and x_offset.	v0.7.0	None Located in Manual Audit

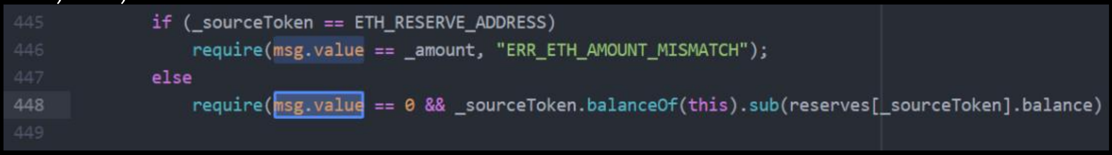
Explicitness Requirements

Using msg.value in non-payable functions (or introducing it via a modifier) is disallowed as a security feature. Turn the function into payable or create a new internal function for the program logic that uses msg.value.	v0.5.0	Several occurances of msg.value use. See Diagram A
Slot and offset of storage pointer variable x are accessed via x.slot and x.offset instead of x_slot and x_offset.	v0.5.0	None Located in Manual Audit
Except for constructors, which uses abstract explicit function visibility is now mandatory. Add public to every function and constructor, and external to every fallback or interface function that does not specify its visibility already.	v0.5.0	None Located in Manual Audit

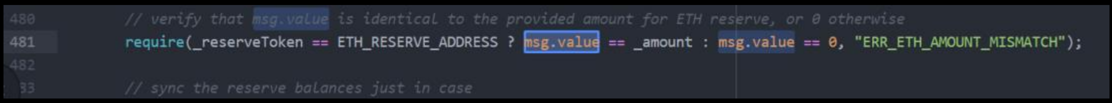
Explicit data location for all variables of struct, array or mapping types is now mandatory. This is also applied to function parameters and return variables.	v0.5.0	None Located in Manual Audit
Member-access to length of arrays is now always read-only, even for storage arrays. It is no longer possible to resize storage arrays by assigning a new value to their length. Use push(), push(value) or pop() instead, or assign a full array, which will of course overwrite the existing content. The reason behind this is to prevent storage collisions of gigantic storage arrays.	v0.6.0	None Located in Manual Audit
The new keyword abstract can be used to mark contracts as abstract. It has to be used if a contract does not implement all its functions. Abstract contracts cannot be created using the new operator, and it is not possible to generate bytecode for them during compilation.	v0.6.0	None Located in Manual Audit
The names of variables declared in inline assembly may no longer end in _slot or _offset.	v0.6.0	None Located in Manual Audit
Variable declarations in inline assembly may no longer shadow any declaration outside the inline assembly block. If the name contains a dot, its prefix up to the dot may not conflict with any declaration outside the inline assembly block.	v0.6.0	None Located in Manual Audit Some use outside audit scope.
State variable shadowing is now disallowed. A derived contract can only declare a state variable x, if there is no visible state variable with the same name in any of its bases.	v.0.6.0	None Located in Manual Audit

The size must now be adjusted within the type before the conversion. For example, you can convert a bytes4 (4 bytes) to a uint64 (8 bytes) by first converting the bytes4 variable to bytes8 and then to uint64. You get the opposite padding when converting through uint32. (since v0.6.0)	V0.6.0	None Located in Manual Audit

Diagram A: SEVERAL USES OF THIS IN THE CONTRACT LOCATED: Line 446,448, and 481



```
445         if (_sourceToken == ETH_RESERVE_ADDRESS)
446             require(msg.value == _amount, "ERR_ETH_AMOUNT_MISMATCH");
447         else
448             require(msg.value == 0 && _sourceToken.balanceOf(this).sub(reserves[_sourceToken].balance)
449
```



```
480         // verify that msg.value is identical to the provided amount for ETH reserve, or 0 otherwise
481         require(_reserveToken == ETH_RESERVE_ADDRESS ? msg.value == _amount : msg.value == 0, "ERR_ETH_AMOUNT_MISMATCH");
482
33          // sync the reserve balances just in case
```

Variables

Declaring empty structs is now disallowed for clarity.	v0.5.0	None Located in Manual Audit
Removal of unsafe features and methods. If a struct or array contains a mapping, it can only be used in storage. Previously, mapping members were silently skipped in memory, which is confusing and error-prone.	v0.7.0	None Located in Manual Audit
Assignments to structs or arrays in storage does not work if they contain mappings. Previously, mappings were silently skipped during the copy operation, which is misleading and error-prone.	v0.7.0	None Located in Manual Audit
The var keyword is now disallowed to favor explicitness. Assignments between tuples with different number of components is now disallowed.	v0.5.0	None Located in Manual Audit
Values for constants that are not	v0.5.0	None Located in Manual

compile-time constants are disallowed.		Audit
Assignments between tuples with different number of components is now disallowed.	V0.5.0	None Located in Manual Audit
Multi-variable declarations with mismatching number of values are now disallowed.	v0.5.0	None Located in Manual Audit
Uninitialized storage variables are now disallowed.	v0.5.0	None Located in Manual Audit
Empty tuple components are now disallowed.	v0.5.0	None Located in Manual Audit
Fixed-size arrays with a length of zero are now disallowed.	v.0.5.0	None Located in Manual Audit
struct and enum types can be declared at file level.	v.0.6.0	None Located in Manual Audit
The global variable now is deprecated, block.timestamp should be used instead. The single identifier now is too generic for a global variable and could give the impression that it changes during transaction processing, whereas block.timestamp correctly reflects the fact that it is just a property of the block.	v.0.6.0	Identified in code: See Finding 3.2

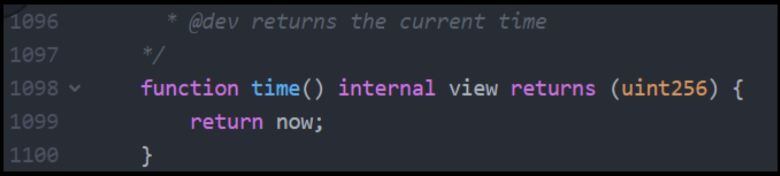
3.2 BLOCKTIME STAMP ALIAS USAGE - LOW

Description:

During a manual static review, the tester noticed the use of "now." The contract developers should be aware that his does not mean current time. "now" is an alias for "block.timestamp". "block.timestamp" can be influenced by miners to a certain degree, so the testers should be warned that this may have some risk if miners collude on time manipulation to influence the price oracles. Please note, while this contract is at version 0.4 in the supported version 0.6.0. the alias now for block.timestamp has been removed. (as indicated in the prior section)

Code Location:

\contracts\converter\types\liquidity-pool-v2\LiquidityPoolV2Converter.sol - Line 1099



```
1096      * @dev returns the current time
1097     */
1098     function time() internal view returns (uint256) {
1099         return now;
1100     }
```

Recommendation:

Refactor this from 0.4 in the latest version 0.7.0. to use the correct time variables that are applicable.

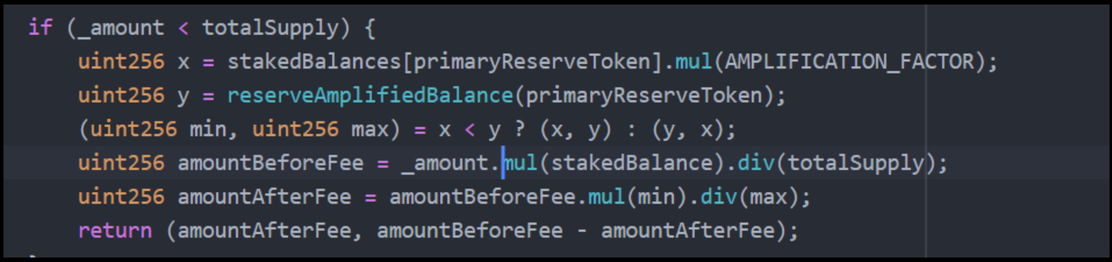
3.3 DIVIDE BEFORE MULTIPLY - LOW

Description:

Solidity integer division might truncate. As a result, performing multiplication before division might reduce precision. Due to the sensitivity of precision, and the amount of detail the development team is putting on the dynamic balancing mechanics involved in Bancor, this may be a factor in accuracy of weights/rates.

Code Location:

LiquidityPoolV2Converter.sol Line #622-633



```
if (_amount < totalSupply) {
    uint256 x = stakedBalances[primaryReserveToken].mul(AMPLIFICATION_FACTOR);
    uint256 y = reserveAmplifiedBalance(primaryReserveToken);
    (uint256 min, uint256 max) = x < y ? (x, y) : (y, x);
    uint256 amountBeforeFee = _amount.mul(stakedBalance).div(totalSupply);
    uint256 amountAfterFee = amountBeforeFee.mul(min).div(max);
    return (amountAfterFee, amountBeforeFee - amountAfterFee);
```

Recommendation:

Consider ordering multiplication before division.

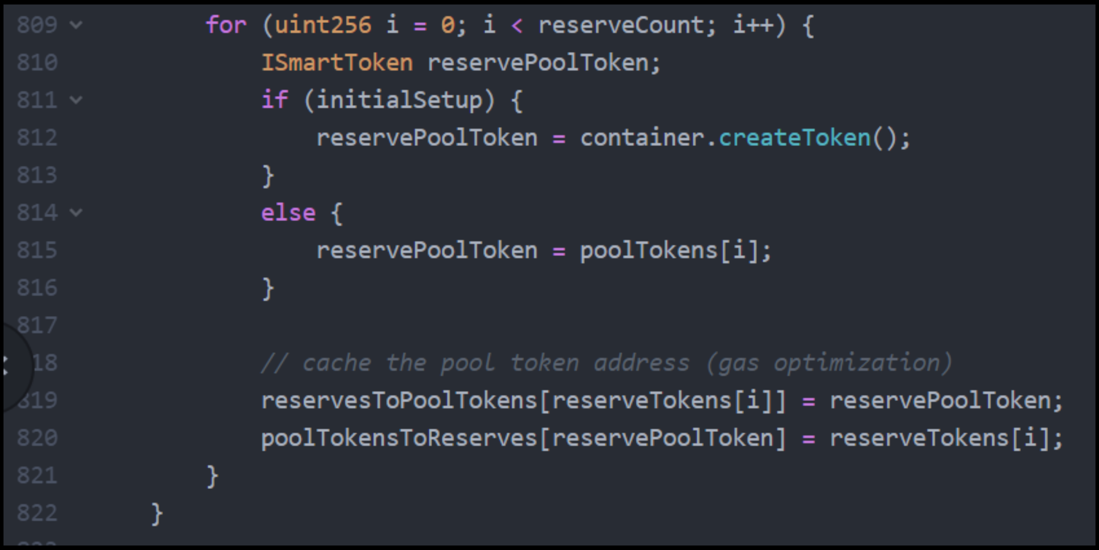
3.4 EXTERNAL CALLS WITHIN A LOOP - VERY LOW

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 the reserveCount variable. If this integer is evaluated at extremely large numbers, or `i` is reset by external calling functions, this can cause a DoS.

Code Location:

LiquidityPoolV2Converter.sol Line #809-820



```
809         for (uint256 i = 0; i < reserveCount; i++) {
810             ISmartToken reservePoolToken;
811             if (initialSetup) {
812                 reservePoolToken = container.createToken();
813             }
814             else {
815                 reservePoolToken = poolTokens[i];
816             }
817
818             // cache the pool token address (gas optimization)
819             reservesToPoolTokens[reserveTokens[i]] = reservePoolToken;
820             poolTokensToReserves[reservePoolToken] = reserveTokens[i];
821         }
822     }
```

Recommendation:

If possible, use pull over push strategy for external calls.

3.5 EXPLOITATION OF DEPLOYED CONTRACT - VERY LOW

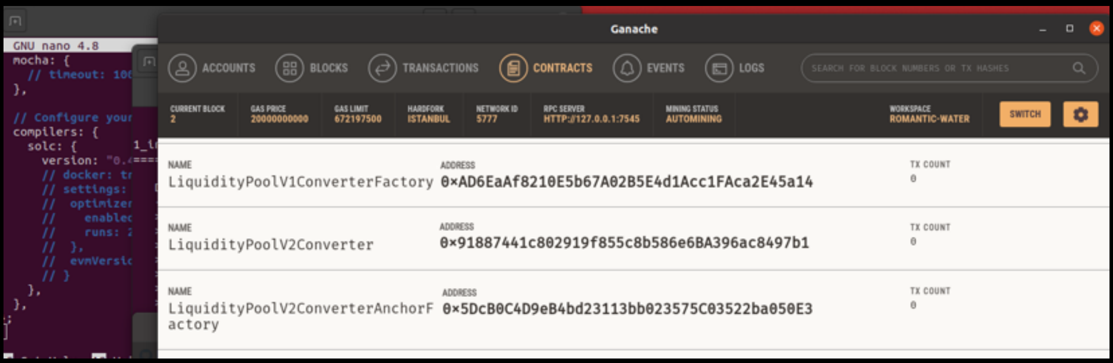
Description

Today, there exists many threat actors with advanced tools waiting to prey on vulnerable contracts deployed with weak programming structures, or flaws in the logic. One such tool used with great success is an automated solidity exploitation utility called karl (https://github.com/cleanunicorn/karl)

Karl monitors for new Smart Contracts deployed on the blockchain, checks for security vulnerabilities, and automatically lets users monitoring the blockchain if any exploitation vectors are identified through binary analysis.

Although karl is often used for evil, this tool is a great tool to use by defenders, auditors, and developers deploying their contract to trusted local testnets to verify the integrity of the smart contracts before deploying to production.

To test for automated exploitation in the Bancor Liquidity Pool contract, Halborn first compiled, and deployed the solidity project branch on a local Ganache Testnet with truffle. All 70 contracts were deployed and migrated, including the scoped contract “LiquidityPoolV2Converter.



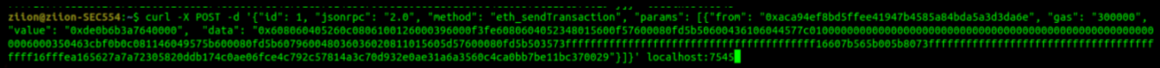
```
GNU nano 4.8
mocha: {
  // timeout: 10(
},

// Configure your
compilers: {
  solc: {
    version: "0.
    // docker: tr
    // settings:
    //  optimizer
    //    enablec
    //    runs: 2
    //  },
    //  evmVersic
    // }
  },
},
```

CURRENT BLOCK 2 | GAS PRICE 20000000000 | GAS LIMIT 672197500 | HARDFORK ISTANBUL | NETWORK ID 5777 | RPC SERVER HTTP://127.0.0.1:7545 | MINING STATUS AUTOMINING | WORKSPACE ROMANTIC-WATER

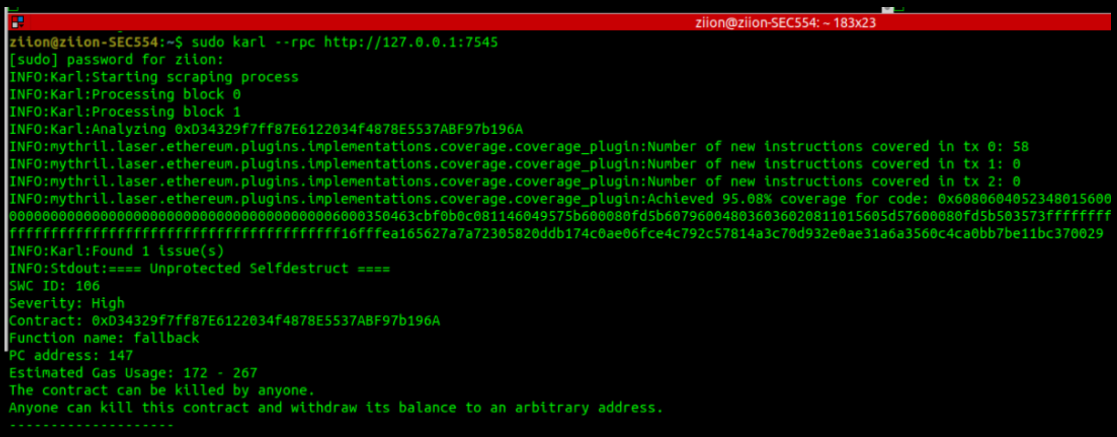
NAME	ADDRESS	TX COUNT
LiquidityPoolV1ConverterFactory	0×AD6EaAf8210E5b67A02B5E4d1Acc1FAca2E45a14	0
LiquidityPoolV2Converter	0×91887441c802919f855c8b586e6BA396ac8497b1	0
LiquidityPoolV2ConverterAnchorFactory	0×5DcB0C4D9eB4bd23113bb023575C03522ba050E3	0

To test the correct deployment of karl listening on the locally hosted blockchain at 127.0.0.1:7545, and intentionally vulnerable contract with a “suicide” function was uploaded to trigger a detection. We can see karl correctly discovered the exploit.



```
ziion@ziion-SEC554:~$ curl -X POST -d '{"id": 1, "jsonrpc": "2.0", "method": "eth_sendTransaction", "params": [{"from": "0xaca94ef8bd5ffee41947b4585a84bda5a3d3da6e", "gas": "300000",
"value": "0xde0b6b3a7640000",  "data": "0x608060405260c0806100126000396000f3fe6080604052348015600f57600080fd5b506004361060044577c0100000000000000000000000000000000000000000000000000000
0006000350463cbf0b0c081146049575b600080fd5b6079600480360360208110156 05d57600080fd5b503573ffffffffffffffffffffffffffffffffffffffff16607b565b005b8073ffffffffffffffffffffffffffffffffffff
ffff16fffea165627a7a72305820ddb174c0ae06fce4c792c57814a3c70d932e0ae31a6a3560c4ca0bb7be11bc370029"}]}' localhost:7545
```

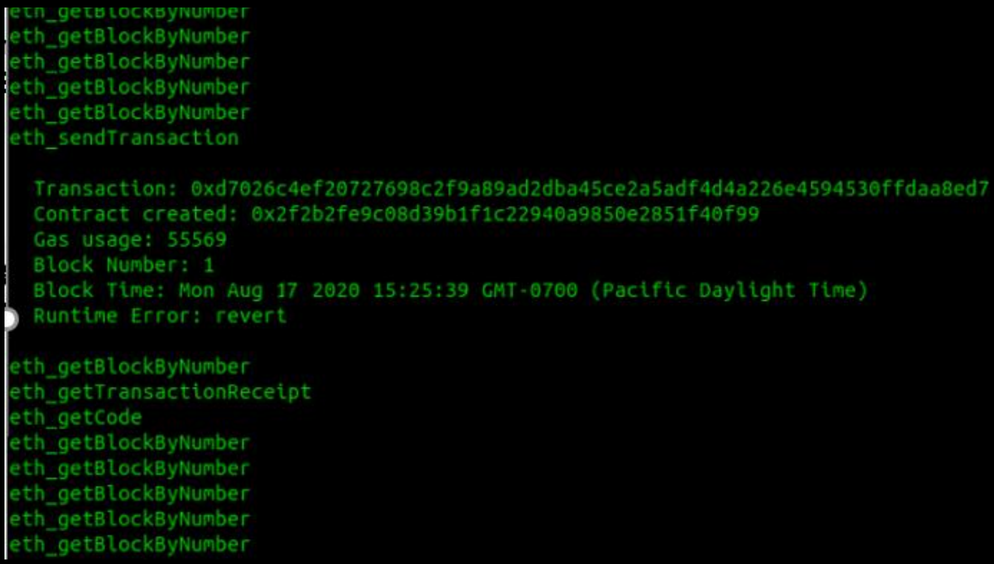
Intentionally vulnerable contract posted to local testnet.:



```
ziion@ziion-SEC554:~$ sudo karl --rpc http://127.0.0.1:7545
[sudo] password for ziion:
INFO:Karl:Starting scraping process
INFO:Karl:Processing block 0
INFO:Karl:Processing block 1
INFO:Karl:Analyzing 0xD34329f7ff87E6122034f4878E5537ABF97b196A
INFO:mythril.laser.ethereum.plugins.implementations.coverage.coverage_plugin:Number of new instructions covered in tx 0: 58
INFO:mythril.laser.ethereum.plugins.implementations.coverage.coverage_plugin:Number of new instructions covered in tx 1: 0
INFO:mythril.laser.ethereum.plugins.implementations.coverage.coverage_plugin:Number of new instructions covered in tx 2: 0
INFO:mythril.laser.ethereum.plugins.implementations.coverage.coverage_plugin:Achieved 95.08% coverage for code: 0x6080604052348015600
00000000000000000000000000000000006000350463cbf0b0c081146049575b600080fd5b6079600480360360208110156 05d57600080fd5b503573fffffff
ffffffffffffffffffffffffffffffffffffffff16fffea165627a7a72305820ddb174c0ae06fce4c792c57814a3c70d932e0ae31a6a3560c4ca0bb7be11bc370029
INFO:Karl:Found 1 issue(s)
INFO:Stdout:==== Unprotected Selfdestruct ====
SWC ID: 106
Severity: High
Contract: 0xD34329f7ff87E6122034f4878E5537ABF97b196A
Function name: fallback
PC address: 147
Estimated Gas Usage: 172 - 267
The contract can be killed by anyone.
Anyone can kill this contract and withdraw its balance to an arbitrary address.
--------------------
```

Karl detection trigger on exploitation of vulnerable contract.

The Bancor Contracts were then redeployed to the testnet. We can see that no Vulnerabilities were triggered, which include several detections on reentrancy, overflow/underflow, and other security vectors found in the binaries send to the blockchain.OnlyMinter conditions correctly work.



```
eth_getBlockByNumber
eth_getBlockByNumber
eth_getBlockByNumber
eth_getBlockByNumber
eth_getBlockByNumber
eth_sendTransaction

  Transaction: 0xd7026c4ef20727698c2f9a89ad2dba45ce2a5adf4d4a226e4594530ffdaa8ed7
  Contract created: 0x2f2b2fe9c08d39b1f1c22940a9850e2851f40f99
  Gas usage: 55569
  Block Number: 1
  Block Time: Mon Aug 17 2020 15:25:39 GMT-0700 (Pacific Daylight Time)
  Runtime Error: revert

eth_getBlockByNumber
eth_getTransactionReceipt
eth_getCode
eth_getBlockByNumber
eth_getBlockByNumber
eth_getBlockByNumber
eth_getBlockByNumber
eth_getBlockByNumber
```

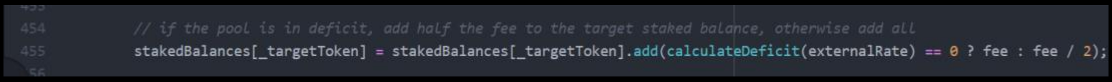
3.6 STRICT EQUALITIES - VERY LOW

Description:

Use of strict equalities that can be easily manipulated by an attacker.

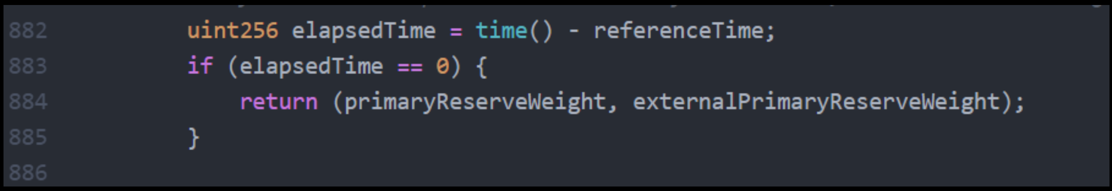
Code Location:

LiquidityPoolV2Converter.sol Line #455



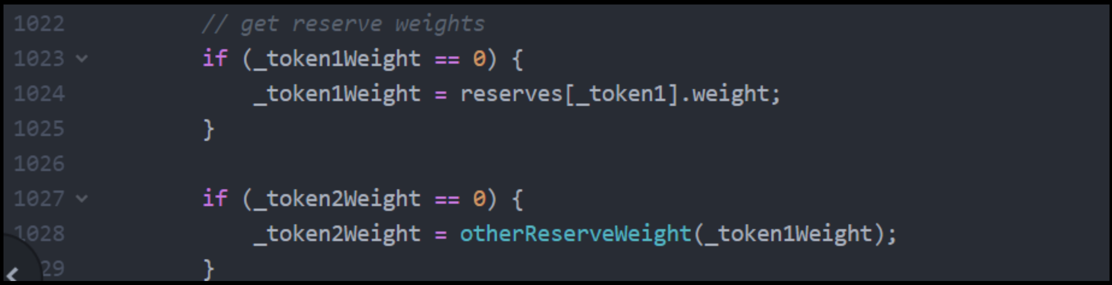
```
454         // if the pool is in deficit, add half the fee to the target staked balance, otherwise add all
455         stakedBalances[_targetToken] = stakedBalances[_targetToken].add(calculateDeficit(externalRate) == 0 ? fee : fee / 2);
```

LiquidityPoolV2Converter.sol Line #883



```
882         uint256 elapsedTime = time() - referenceTime;
883         if (elapsedTime == 0) {
884             return (primaryReserveWeight, externalPrimaryReserveWeight);
885         }
886
```

LiquidityPoolV2Converter.sol Line #1023-1028



```
1022         // get reserve weights
1023         if (_token1Weight == 0) {
1024             _token1Weight = reserves[_token1].weight;
1025         }
1026
1027         if (_token2Weight == 0) {
1028             _token2Weight = otherReserveWeight(_token1Weight);
  29         }
```

Recommendation:

While these sections of code use it for time, and weight adjustments, Don't use strict equality to determine if an account has enough Ether or tokens.

3.7 STATIC ANALYSIS REPORT - INFORMATIONAL

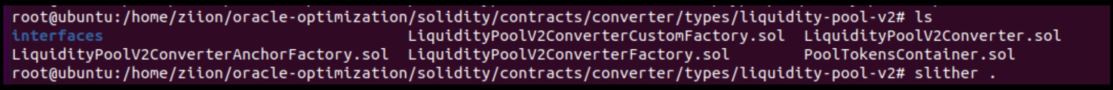
Description:

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 Bancor Smart Contract set including the new version of the LiquidityPoolV2Converter. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.

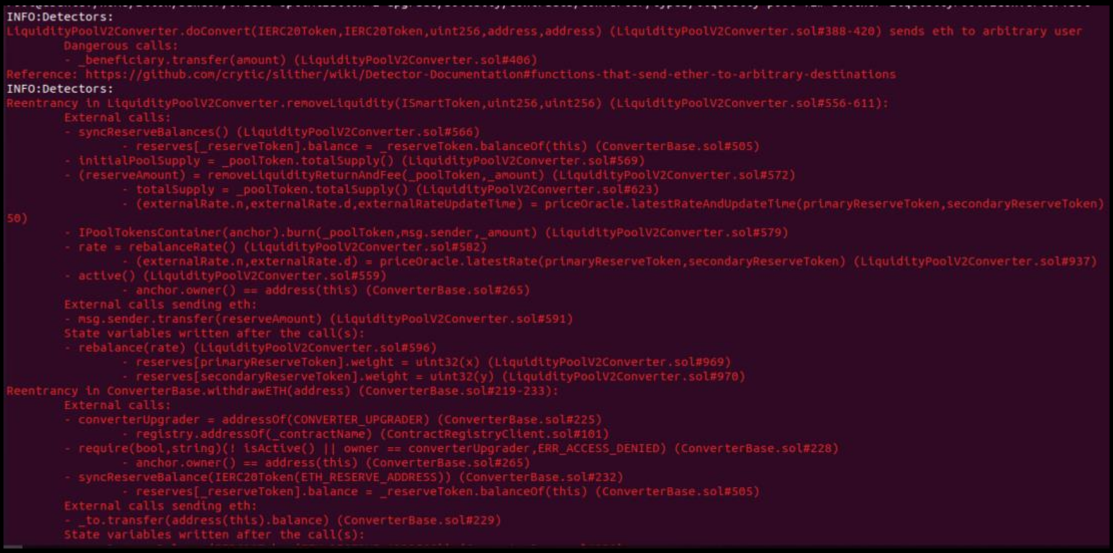
Results:

Slither responded with the majority of detections being Reentrancy Attacks due to External Calls and the state variables associated being written after the call is finished. While these are valid conditions of a reentrancy attack, most of the External calls are within the same contract, and from a function elsewhere in the codebase.

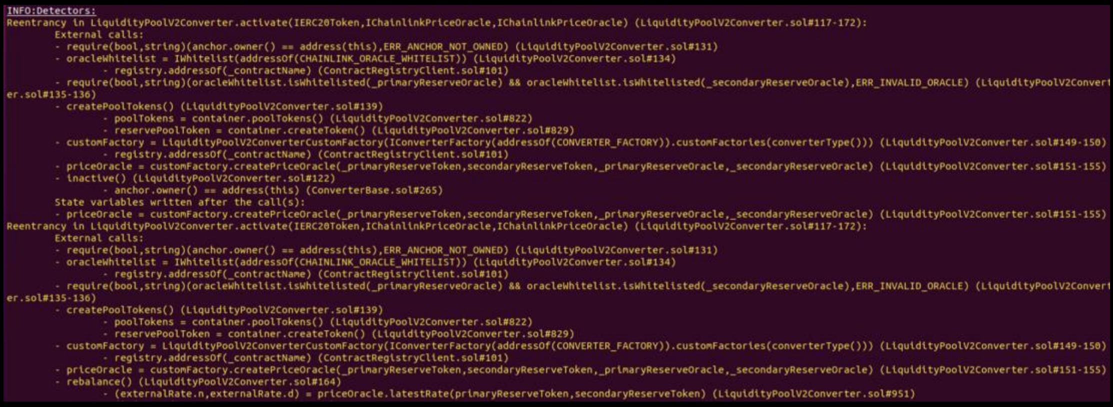
There are also several variables in regards to what can make a reentrancy bug benign or exploitable. The worst condition is in the transfer or change in ether or token balance. These are often bugs in which a contract can call withdrawBalance two times, and withdraw more than its initial deposit to the contract. The tester did not see any direct balance transfers, however, the developers are encouraged to check the list detected by Slither for any issues.



```
root@ubuntu:/home/ziion/oracle-optimization/solidity/contracts/converter/types/liquidity-pool-v2# ls
interfaces                                   LiquidityPoolV2ConverterCustomFactory.sol  LiquidityPoolV2Converter.sol
LiquidityPoolV2ConverterAnchorFactory.sol  LiquidityPoolV2ConverterFactory.sol        PoolTokensContainer.sol
root@ubuntu:/home/ziion/oracle-optimization/solidity/contracts/converter/types/liquidity-pool-v2# slither .
```



```
INFO:Detectors:
LiquidityPoolV2Converter.doConvert(IERC20Token,IERC20Token,uint256,address,address) (LiquidityPoolV2Converter.sol#388-420) sends eth to arbitrary user
        Dangerous calls:
        - _beneficiary.transfer(amount) (LiquidityPoolV2Converter.sol#406)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#functions-that-send-ether-to-arbitrary-destinations
INFO:Detectors:
Reentrancy in LiquidityPoolV2Converter.removeLiquidity(ISmartToken,uint256,uint256) (LiquidityPoolV2Converter.sol#556-611):
        External calls:
        - syncReserveBalances() (LiquidityPoolV2Converter.sol#566)
                - reserves[_reserveToken].balance = _reserveToken.balanceOf(this) (ConverterBase.sol#505)
        - initialPoolSupply = _poolToken.totalSupply() (LiquidityPoolV2Converter.sol#569)
        - (reserveAmount) = removeLiquidityReturnAndFee(_poolToken,_amount) (LiquidityPoolV2Converter.sol#572)
                - totalSupply = _poolToken.totalSupply() (LiquidityPoolV2Converter.sol#623)
                - (externalRate.n,externalRate.d,externalRateUpdateTime) = priceOracle.latestRateAndUpdateTime(primaryReserveToken,secondaryReserveToken)
50)
        - IPoolTokensContainer(anchor).burn(_poolToken,msg.sender,_amount) (LiquidityPoolV2Converter.sol#579)
        - rate = rebalanceRate() (LiquidityPoolV2Converter.sol#582)
                - (externalRate.n,externalRate.d) = priceOracle.latestRate(primaryReserveToken,secondaryReserveToken) (LiquidityPoolV2Converter.sol#937)
        - active() (LiquidityPoolV2Converter.sol#559)
                - anchor.owner() == address(this) (ConverterBase.sol#265)
        External calls sending eth:
        - msg.sender.transfer(reserveAmount) (LiquidityPoolV2Converter.sol#591)
        State variables written after the call(s):
        - rebalance(rate) (LiquidityPoolV2Converter.sol#596)
                - reserves[primaryReserveToken].weight = uint32(x) (LiquidityPoolV2Converter.sol#969)
                - reserves[secondaryReserveToken].weight = uint32(y) (LiquidityPoolV2Converter.sol#970)
Reentrancy in ConverterBase.withdrawETH(address) (ConverterBase.sol#219-233):
        External calls:
        - converterUpgrader = addressOf(CONVERTER_UPGRADER) (ConverterBase.sol#225)
                - registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
        - require(bool,string)(! isActive() || owner == converterUpgrader,ERR_ACCESS_DENIED) (ConverterBase.sol#228)
                - anchor.owner() == address(this) (ConverterBase.sol#265)
        - syncReserveBalance(IERC20Token(ETH_RESERVE_ADDRESS)) (ConverterBase.sol#232)
                - reserves[_reserveToken].balance = _reserveToken.balanceOf(this) (ConverterBase.sol#505)
        External calls sending eth:
        - _to.transfer(address(this).balance) (ConverterBase.sol#229)
        State variables written after the call(s):
```



```
INFO:Detectors:
Reentrancy in LiquidityPoolV2Converter.activate(IERC20Token,IChainlinkPriceOracle,IChainlinkPriceOracle) (LiquidityPoolV2Converter.sol#117-172):
        External calls:
        - require(bool,string)(anchor.owner() == address(this),ERR_ANCHOR_NOT_OWNED) (LiquidityPoolV2Converter.sol#131)
        - oracleWhitelist = IWhitelist(addressOf(CHAINLINK_ORACLE_WHITELIST)) (LiquidityPoolV2Converter.sol#134)
                - registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
        - require(bool,string)(oracleWhitelist.isWhitelisted(_primaryReserveOracle) && oracleWhitelist.isWhitelisted(_secondaryReserveOracle),ERR_INVALID_ORACLE) (LiquidityPoolV2Convert
er.sol#135-136)
        - createPoolTokens() (LiquidityPoolV2Converter.sol#139)
                - poolTokens = container.poolTokens() (LiquidityPoolV2Converter.sol#822)
                - reservePoolToken = container.createToken() (LiquidityPoolV2Converter.sol#829)
        - customFactory = LiquidityPoolV2ConverterCustomFactory(IConverterFactory(addressOf(CONVERTER_FACTORY)).customFactories(converterType())) (LiquidityPoolV2Converter.sol#149-150)
                - registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
        - priceOracle = customFactory.createPriceOracle(_primaryReserveToken,secondaryReserveToken,_primaryReserveOracle,_secondaryReserveOracle) (LiquidityPoolV2Converter.sol#151-155)
        - inactive() (LiquidityPoolV2Converter.sol#122)
                - anchor.owner() == address(this) (ConverterBase.sol#265)
        State variables written after the call(s):
        - priceOracle = customFactory.createPriceOracle(_primaryReserveToken,secondaryReserveToken,_primaryReserveOracle,_secondaryReserveOracle) (LiquidityPoolV2Converter.sol#151-155)
Reentrancy in LiquidityPoolV2Converter.activate(IERC20Token,IChainlinkPriceOracle,IChainlinkPriceOracle) (LiquidityPoolV2Converter.sol#117-172):
        External calls:
        - require(bool,string)(anchor.owner() == address(this),ERR_ANCHOR_NOT_OWNED) (LiquidityPoolV2Converter.sol#131)
        - oracleWhitelist = IWhitelist(addressOf(CHAINLINK_ORACLE_WHITELIST)) (LiquidityPoolV2Converter.sol#134)
                - registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
        - require(bool,string)(oracleWhitelist.isWhitelisted(_primaryReserveOracle) && oracleWhitelist.isWhitelisted(_secondaryReserveOracle),ERR_INVALID_ORACLE) (LiquidityPoolV2Convert
er.sol#135-136)
        - createPoolTokens() (LiquidityPoolV2Converter.sol#139)
                - poolTokens = container.poolTokens() (LiquidityPoolV2Converter.sol#822)
                - reservePoolToken = container.createToken() (LiquidityPoolV2Converter.sol#829)
        - customFactory = LiquidityPoolV2ConverterCustomFactory(IConverterFactory(addressOf(CONVERTER_FACTORY)).customFactories(converterType())) (LiquidityPoolV2Converter.sol#149-150)
                - registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
        - priceOracle = customFactory.createPriceOracle(_primaryReserveToken,secondaryReserveToken,_primaryReserveOracle,_secondaryReserveOracle) (LiquidityPoolV2Converter.sol#151-155)
        - rebalance() (LiquidityPoolV2Converter.sol#164)
                - (externalRate.n,externalRate.d) = priceOracle.latestRate(primaryReserveToken,secondaryReserveToken) (LiquidityPoolV2Converter.sol#951)
```

Recommendation:

While reentrancy attacks are among the worst classifications of vulnerabilities, Bancor has implemented mitigating contracts to help protect the platform from this threat. Among the contracts compiled in the oracle-optimization repository is ReentrancyGuard.sol
(https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/ReentrancyGuard.sol)

This is a utility contract created by Open-Zeppelin and is a Contract module that helps prevent reentrant calls to a function. Inheriting from `ReentrancyGuard.sol` will make the {nonReentrant} modifier available, which can be applied to functions to make sure there are no nested (reentrant) calls to them.

Since testing the valid use and implementation of ReentrancyGuard is not in scope, the tester is marking this as an Informational Level, and encourages the developers to validate correct use of the utility contract, as well as checking the list of Reentrancy Detection output from Slither for true positives. They are listed below:

Reference Data:

1: Reentrancy in LiquidityPoolV2Converter.removeLiquidity(ISmartToken,uint256,uint256) (LiquidityPoolV2Converter.sol#556-611):

```
External calls:
- syncReserveBalances() (LiquidityPoolV2Converter.sol#566)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this) (ConverterBase.sol#505)
- initialPoolSupply = _poolToken.totalSupply() (LiquidityPoolV2Converter.sol#569)
- (reserveAmount) = removeLiquidityReturnAndFee(_poolToken,_amount)
(LiquidityPoolV2Converter.sol#572)
```

```
- totalSupply = _poolToken.totalSupply() (LiquidityPoolV2Converter.sol#623)
- (externalRate.n,externalRate.d,externalRateUpdateTime) =
priceOracle.latestRateAndUpdateTime(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#850)
- IPoolTokensContainer(anchor).burn(_poolToken,msg.sender,_amount)
(LiquidityPoolV2Converter.sol#579)
- rate = rebalanceRate() (LiquidityPoolV2Converter.sol#582)
- (externalRate.n,externalRate.d) =
priceOracle.latestRate(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#937)
- active() (LiquidityPoolV2Converter.sol#559)
- anchor.owner() == address(this) (ConverterBase.sol#265)
External calls sending eth:
- msg.sender.transfer(reserveAmount) (LiquidityPoolV2Converter.sol#591)
State variables written after the call(s):
- rebalance(rate) (LiquidityPoolV2Converter.sol#596)
- reserves[primaryReserveToken].weight = uint32(x) (LiquidityPoolV2Converter.sol#969)
- reserves[secondaryReserveToken].weight = uint32(y) (LiquidityPoolV2Converter.sol#970)
```

2: Reentrancy in ConverterBase.withdrawETH(address) (ConverterBase.sol#219-233):

```
External calls:
- converterUpgrader = addressOf(CONVERTER_UPGRADER) (ConverterBase.sol#225)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- require(bool,string)(! isActive() || owner == converterUpgrader,ERR_ACCESS_DENIED)
(ConverterBase.sol#228)
- anchor.owner() == address(this) (ConverterBase.sol#265)
- syncReserveBalance(IERC20Token(ETH_RESERVE_ADDRESS)) (ConverterBase.sol#232)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this) (ConverterBase.sol#505)
External calls sending eth:
- _to.transfer(address(this).balance) (ConverterBase.sol#229)
State variables written after the call(s):
- syncReserveBalance(IERC20Token(ETH_RESERVE_ADDRESS)) (ConverterBase.sol#232)
- reserves[_reserveToken].balance = address(this).balance (ConverterBase.sol#503)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this) (ConverterBase.sol#505)
```

3. Reentrancy in LiquidityPoolV2Converter.activate(IERC20Token,IChainlinkPriceOracle,IChainlinkPriceOracle) (LiquidityPoolV2Converter.sol#117-172):

```
External calls:
- require(bool,string)(anchor.owner() == address(this),ERR_ANCHOR_NOT_OWNED)
(LiquidityPoolV2Converter.sol#131)
- oracleWhitelist = IWhitelist(addressOf(CHAINLINK_ORACLE_WHITELIST))
(LiquidityPoolV2Converter.sol#134)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- require(bool,string)(oracleWhitelist.isWhitelisted(_primaryReserveOracle) &&
oracleWhitelist.isWhitelisted(_secondaryReserveOracle),ERR_INVALID_ORACLE)
(LiquidityPoolV2Converter.sol#135-136)
- createPoolTokens() (LiquidityPoolV2Converter.sol#139)
- poolTokens = container.poolTokens() (LiquidityPoolV2Converter.sol#822)
- reservePoolToken = container.createToken() (LiquidityPoolV2Converter.sol#829)
- customFactory =
LiquidityPoolV2ConverterCustomFactory(IConverterFactory(addressOf(CONVERTER_FACTORY)).cu
stomFactories(converterType())) (LiquidityPoolV2Converter.sol#149-150)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- priceOracle =
customFactory.createPriceOracle(_primaryReserveToken,secondaryReserveToken,_primaryReser
veOracle,_secondaryReserveOracle) (LiquidityPoolV2Converter.sol#151-155)
- inactive() (LiquidityPoolV2Converter.sol#122)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- priceOracle =
customFactory.createPriceOracle(_primaryReserveToken,secondaryReserveToken,_primaryReser
veOracle,_secondaryReserveOracle) (LiquidityPoolV2Converter.sol#151-155)
```

4. Reentrancy in LiquidityPoolV2Converter.activate(IERC20Token,IChainlinkPriceOracle,IChainlin kPriceOracle) (LiquidityPoolV2Converter.sol#117-172):

```
External calls:
- require(bool,string)(anchor.owner() == address(this),ERR_ANCHOR_NOT_OWNED)
(LiquidityPoolV2Converter.sol#131)
- oracleWhitelist = IWhitelist(addressOf(CHAINLINK_ORACLE_WHITELIST))
(LiquidityPoolV2Converter.sol#134)
```

```
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- require(bool,string)(oracleWhitelist.isWhitelisted(_primaryReserveOracle) &&
oracleWhitelist.isWhitelisted(_secondaryReserveOracle),ERR_INVALID_ORACLE)
(LiquidityPoolV2Converter.sol#135-136)
- createPoolTokens() (LiquidityPoolV2Converter.sol#139)
- poolTokens = container.poolTokens() (LiquidityPoolV2Converter.sol#822)
- reservePoolToken = container.createToken() (LiquidityPoolV2Converter.sol#829)
- customFactory =
LiquidityPoolV2ConverterCustomFactory(IConverterFactory(addressOf(CONVERTER_FACTORY)).cu
stomFactories(converterType())) (LiquidityPoolV2Converter.sol#149-150)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- priceOracle =
customFactory.createPriceOracle(_primaryReserveToken,secondaryReserveToken,_primaryReser
veOracle,_secondaryReserveOracle) (LiquidityPoolV2Converter.sol#151-155)
- rebalance() (LiquidityPoolV2Converter.sol#164)
- (externalRate.n,externalRate.d) =
priceOracle.latestRate(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#951)
- inactive() (LiquidityPoolV2Converter.sol#122)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- rebalance() (LiquidityPoolV2Converter.sol#164)
- reserves[primaryReserveToken].weight = uint32(x) (LiquidityPoolV2Converter.sol#969)
- reserves[secondaryReserveToken].weight = uint32(y) (LiquidityPoolV2Converter.sol#970)
```

5. Reentrancy in LiquidityPoolV2Converter.activate(IERC20Token,IChainlinkPriceOracle,IChainlinkPriceOracle) (LiquidityPoolV2Converter.sol#117-172):

```
External calls:
- require(bool,string)(anchor.owner() == address(this),ERR_ANCHOR_NOT_OWNED)
(LiquidityPoolV2Converter.sol#131)
- oracleWhitelist = IWhitelist(addressOf(CHAINLINK_ORACLE_WHITELIST))
(LiquidityPoolV2Converter.sol#134)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- require(bool,string)(oracleWhitelist.isWhitelisted(_primaryReserveOracle) &&
oracleWhitelist.isWhitelisted(_secondaryReserveOracle),ERR_INVALID_ORACLE)
(LiquidityPoolV2Converter.sol#135-136)
- createPoolTokens() (LiquidityPoolV2Converter.sol#139)
- poolTokens = container.poolTokens() (LiquidityPoolV2Converter.sol#822)
- reservePoolToken = container.createToken() (LiquidityPoolV2Converter.sol#829)
- customFactory =
LiquidityPoolV2ConverterCustomFactory(IConverterFactory(addressOf(CONVERTER_FACTORY)).cu
stomFactories(converterType())) (LiquidityPoolV2Converter.sol#149-150)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- priceOracle =
customFactory.createPriceOracle(_primaryReserveToken,secondaryReserveToken,_primaryReser
veOracle,_secondaryReserveOracle) (LiquidityPoolV2Converter.sol#151-155)
- rebalance() (LiquidityPoolV2Converter.sol#168)
- (externalRate.n,externalRate.d) =
priceOracle.latestRate(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#951)
- inactive() (LiquidityPoolV2Converter.sol#122)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- rebalance() (LiquidityPoolV2Converter.sol#168)
- reserves[primaryReserveToken].weight = uint32(x) (LiquidityPoolV2Converter.sol#969)
- reserves[secondaryReserveToken].weight = uint32(y) (LiquidityPoolV2Converter.sol#970)
```

6. Reentrancy in LiquidityPoolV2Converter.addLiquidity(IERC20Token,uint256,uint256) (LiquidityPoolV2Converter.sol#475-545):

```
External calls:
- syncReserveBalances() (LiquidityPoolV2Converter.sol#489)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
- active() (LiquidityPoolV2Converter.sol#479)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- reserves[ETH_RESERVE_ADDRESS].balance =
reserves[ETH_RESERVE_ADDRESS].balance.sub(msg.value) (LiquidityPoolV2Converter.sol#493)
Reentrancy in LiquidityPoolV2Converter.addLiquidity(IERC20Token,uint256,uint256)
(LiquidityPoolV2Converter.sol#475-545):
External calls:
- syncReserveBalances() (LiquidityPoolV2Converter.sol#489)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
- poolTokenSupply = reservePoolToken.totalSupply() (LiquidityPoolV2Converter.sol#505)
- rate = rebalanceRate() (LiquidityPoolV2Converter.sol#512)
- (externalRate.n,externalRate.d) =
priceOracle.latestRate(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#937)
- active() (LiquidityPoolV2Converter.sol#479)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- reserves[_reserveToken].balance = reserves[_reserveToken].balance.add(_amount)
(LiquidityPoolV2Converter.sol#515)
- stakedBalances[_reserveToken] = initialStakedBalance.add(_amount)
(LiquidityPoolV2Converter.sol#516)
```

7. Reentrancy in LiquidityPoolV2Converter.addLiquidity(IERC20Token,uint256,uint256) (LiquidityPoolV2Converter.sol#475-545):

```
External calls:
- syncReserveBalances() (LiquidityPoolV2Converter.sol#489)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
- poolTokenSupply = reservePoolToken.totalSupply() (LiquidityPoolV2Converter.sol#505)
- rate = rebalanceRate() (LiquidityPoolV2Converter.sol#512)
- (externalRate.n,externalRate.d) =
priceOracle.latestRate(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#937)
- IPoolTokensContainer(anchor).mint(reservePoolToken,msg.sender,poolTokenAmount)
(LiquidityPoolV2Converter.sol#529)
- active() (LiquidityPoolV2Converter.sol#479)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- rebalance(rate) (LiquidityPoolV2Converter.sol#532)
- reserves[primaryReserveToken].weight = uint32(x) (LiquidityPoolV2Converter.sol#969)
- reserves[secondaryReserveToken].weight = uint32(y) (LiquidityPoolV2Converter.sol#970)
```

8. Reentrancy in LiquidityPoolV2Converter.doConvert(IERC20Token,IERC20Token,uint256,address,address) (LiquidityPoolV2Converter.sol#388-420):

```
External calls:
- (amount,fee) = doConvert(_sourceToken,_targetToken,_amount)
(LiquidityPoolV2Converter.sol#399)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- (externalRate.n,externalRate.d,externalRateUpdateTime) =
priceOracle.latestRateAndUpdateTime(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#437)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
- targetAmount =
IBancorFormula(addressOf(BANCOR_FORMULA)).crossReserveTargetAmount(sourceBalance,_source
Weight,targetBalance,_targetWeight,_amount) (LiquidityPoolV2Converter.sol#681-687)
- require(bool,string)(msg.value == 0 &&
_sourceToken.balanceOf(this).sub(reserves[_sourceToken].balance) >=
_amount,ERR_INVALID_AMOUNT) (LiquidityPoolV2Converter.sol#453)
- active() (LiquidityPoolV2Converter.sol#390)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- prevConversionTime = time() (LiquidityPoolV2Converter.sol#402)
```

9. Reentrancy in LiquidityPoolV2Converter.doConvert(IERC20Token,IERC20Token,uint256) (LiquidityPoolV2Converter.sol#433-464):

```
External calls:
- (externalRate.n,externalRate.d,externalRateUpdateTime) =
priceOracle.latestRateAndUpdateTime(primaryReserveToken,secondaryReserveToken)
```

```
(LiquidityPoolV2Converter.sol#437)
- (targetAmount,fee) =
prepareConversion(_sourceToken,_targetToken,_amount,externalRate,externalRateUpdateTime)
(LiquidityPoolV2Converter.sol#440)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- targetAmount =
IBancorFormula(addressOf(BANCOR_FORMULA)).crossReserveTargetAmount(sourceBalance,_source
Weight,targetBalance,_targetWeight,_amount) (LiquidityPoolV2Converter.sol#681-687)
- require(bool,string)(msg.value == 0 &&
_sourceToken.balanceOf(this).sub(reserves[_sourceToken].balance) >=
_amount,ERR_INVALID_AMOUNT) (LiquidityPoolV2Converter.sol#453)
- syncReserveBalance(_sourceToken) (LiquidityPoolV2Converter.sol#456)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
State variables written after the call(s):
- syncReserveBalance(_sourceToken) (LiquidityPoolV2Converter.sol#456)
- reserves[_reserveToken].balance = address(this).balance (ConverterBase.sol#503)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
- reserves[_targetToken].balance = targetReserveBalance.sub(targetAmount)
(LiquidityPoolV2Converter.sol#457)
- stakedBalances[_targetToken] = stakedBalances[_targetToken].add(fee)
(LiquidityPoolV2Converter.sol#460)
- stakedBalances[_targetToken] = stakedBalances[_targetToken].add(fee / 2)
(LiquidityPoolV2Converter.sol#460)
```

10. Reentrancy in LiquidityPoolV2Converter.removeLiquidity(ISmartToken,uint256,uint256) (LiquidityPoolV2Converter.sol#556-611):

```
External calls:
- syncReserveBalances() (LiquidityPoolV2Converter.sol#566)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
- initialPoolSupply = _poolToken.totalSupply() (LiquidityPoolV2Converter.sol#569)
- (reserveAmount) = removeLiquidityReturnAndFee(_poolToken,_amount)
(LiquidityPoolV2Converter.sol#572)
- totalSupply = _poolToken.totalSupply() (LiquidityPoolV2Converter.sol#623)
- (externalRate.n,externalRate.d,externalRateUpdateTime) =
priceOracle.latestRateAndUpdateTime(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#850)
- IPoolTokensContainer(anchor).burn(_poolToken,msg.sender,_amount)
(LiquidityPoolV2Converter.sol#579)
- rate = rebalanceRate() (LiquidityPoolV2Converter.sol#582)
- (externalRate.n,externalRate.d) =
priceOracle.latestRate(primaryReserveToken,secondaryReserveToken)
(LiquidityPoolV2Converter.sol#937)
- active() (LiquidityPoolV2Converter.sol#559)
- anchor.owner() == address(this) (ConverterBase.sol#265)
State variables written after the call(s):
- reserves[reserveToken].balance = reserves[reserveToken].balance.sub(reserveAmount)
(LiquidityPoolV2Converter.sol#585)
- stakedBalances[reserveToken] = newStakedBalance (LiquidityPoolV2Converter.sol#587)
```

11. Reentrancy in ContractRegistryClient.updateRegistry() (ContractRegistryClient.sol#55-73):

```
External calls:
- newRegistry = IContractRegistry(addressOf(CONTRACT_REGISTRY))
(ContractRegistryClient.sol#60)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- require(bool,string)(newRegistry.addressOf(CONTRACT_REGISTRY) !=
address(0),ERR_INVALID_REGISTRY) (ContractRegistryClient.sol#66)
State variables written after the call(s):
- registry = newRegistry (ContractRegistryClient.sol#72)
Reentrancy in ConverterBase.upgrade() (ConverterBase.sol#349-358):
External calls:
- converterUpgrader = IConverterUpgrader(addressOf(CONVERTER_UPGRADER))
(ConverterBase.sol#350)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- converterUpgrader.upgrade(version) (ConverterBase.sol#356)
```

```
State variables written after the call(s):
- acceptOwnership() (ConverterBase.sol#357)
- owner = newOwner (Owned.sol#55)
```

12. Reentrancy in ConverterBase.withdrawTokens(IERC20Token,address,uint256) (ConverterBase.sol#331-342):

```
External calls:
- converterUpgrader = addressOf(CONVERTER_UPGRADER) (ConverterBase.sol#332)
- registry.addressOf(_contractName) (ContractRegistryClient.sol#101)
- require(bool,string)(! reserves[_token].isSet || ! isActive() || owner ==
converterUpgrader,ERR_ACCESS_DENIED) (ConverterBase.sol#336)
- anchor.owner() == address(this) (ConverterBase.sol#265)
- syncReserveBalance(_token) (ConverterBase.sol#341)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
State variables written after the call(s):
- syncReserveBalance(_token) (ConverterBase.sol#341)
- reserves[_reserveToken].balance = address(this).balance (ConverterBase.sol#503)
- reserves[_reserveToken].balance = _reserveToken.balanceOf(this)
(ConverterBase.sol#505)
```

3.8 AUTOMATED SECURITY SCAN - INFORMATIONAL

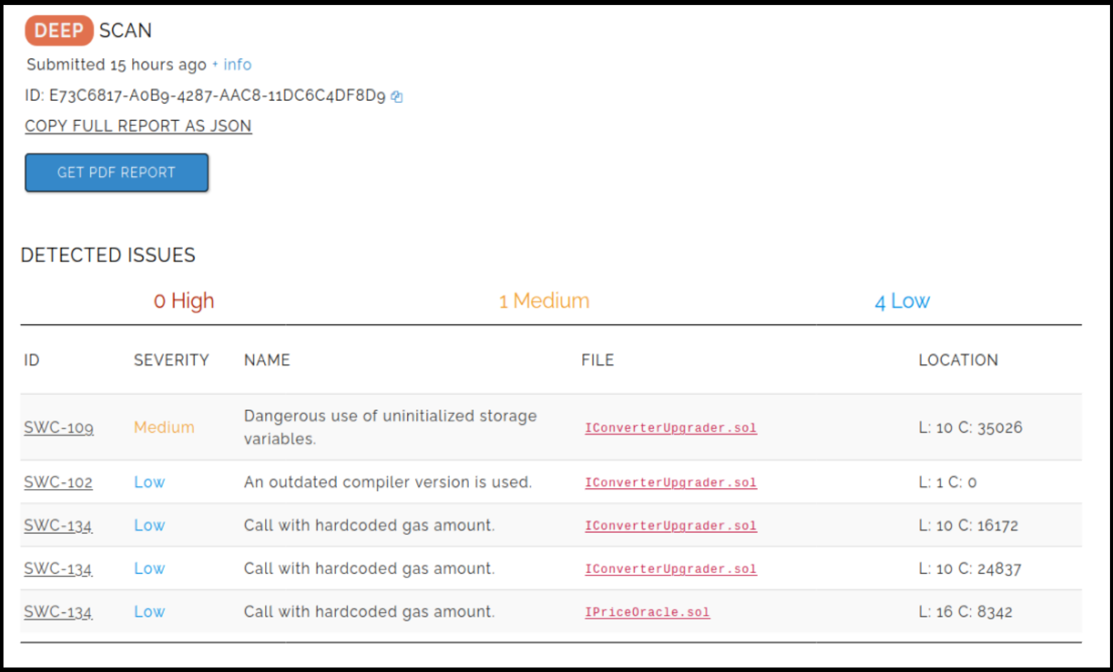
Description:

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 LiquidityPoolV2Converter.sol

Results:

MythX detected 0 High findings, 1 Medium, and 4 Low.
The Medium Finding is detected on a Contract outside the scope of the assessment. (IConverterUpgrader.sol)

One of the Low findings is also identified by Halborn in the Manual review process with detailed recommendations specific to Bancor, and the upgraded smart contract in scope. The others are detailing the use of hard-coded gas amounts.



DEEP SCAN

Submitted 15 hours ago • info

ID: E73C6817-A0B9-4287-AAC8-11DC6C4DF8D9

COPY FULL REPORT AS JSON

DETECTED ISSUES

0 High 1 Medium 4 Low

ID	SEVERITY	NAME	FILE	LOCATION
SWC-109	Medium	Dangerous use of uninitialized storage variables.	IConverterUpgrader.sol	L: 10 C: 35026
SWC-102	Low	An outdated compiler version is used.	IConverterUpgrader.sol	L: 1 C: 0
SWC-134	Low	Call with hardcoded gas amount.	IConverterUpgrader.sol	L: 10 C: 16172
SWC-134	Low	Call with hardcoded gas amount.	IConverterUpgrader.sol	L: 10 C: 24837
SWC-134	Low	Call with hardcoded gas amount.	IPriceOracle.sol	L: 16 C: 8342

3.9 INLINE ASSEMBLY USAGE - INFORMATIONAL

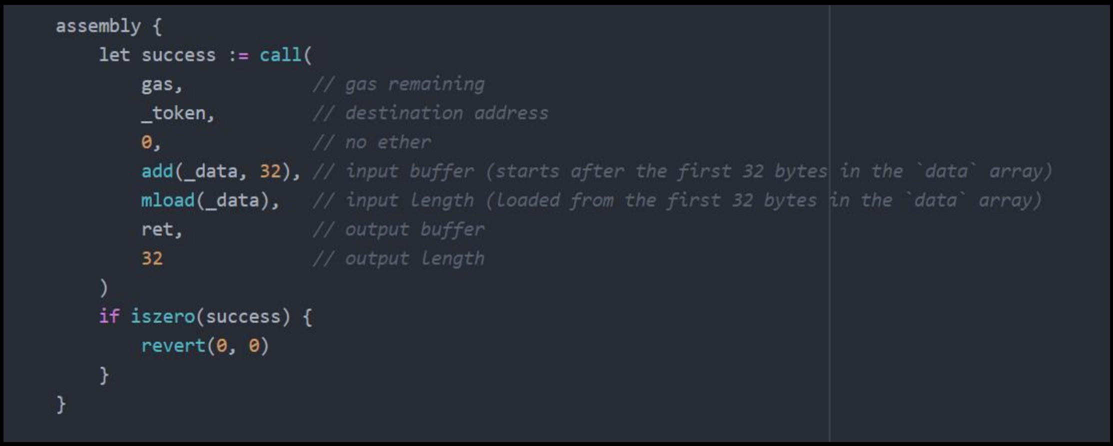
Description:

Inline assembly is a way to access the Ethereum Virtual Machine at a low level. This discards several important safety features of Solidity, and the static compiler. Due to the fact that the EVM is a stack machine, it is often hard to address the correct stack slot and provide arguments to opcodes at the correct point on the stack. Solidity’s inline assembly tries to facilitate that and other issues arising when writing manual assembly. Assembly is much more difficult to write because the compiler does not perform checks, so the developer of the contract should be aware of this warning.

This wasn’t detected on the scoped contract, but the tester wanted to make it aware to the development team.

Code Location:

\contracts\utility\TokenHandler.sol - Line 60



```
assembly {
    let success := call(
        gas,           // gas remaining
        _token,        // destination address
        0,             // no ether
        add(_data, 32), // input buffer (starts after the first 32 bytes in the `data` array)
        mload(_data),  // input length (loaded from the first 32 bytes in the `data` array)
        ret,           // output buffer
        32             // output length
    )
    if iszero(success) {
        revert(0, 0)
    }
}
```

THANK YOU FOR CHOOSING

HALBORN