SURGE

Smart Contract Audit Report

SYSFIXED

13 Jan 2023

TABLE OF CONTENTS

AUDITED DETAILS

Audited Project

Project name	Token ticker	Blockchain
SURGE	SRG	Binance Smart Chain

Addresses

Contract address	0x9f19c8e321bd14345b797d43e01f0eed030f5bff
Contract deployer address	0xc207cd3f61Da958AA6f4209C5f0a145C056B576f

Project Website

https://surgeprotocol.io/

Codebase

https://bscscan.com/address/0x9f19c8e321bd14345b797d43e01f0eed030f5bff#code

SUMMARY

Launching a token on SURGE comes with a first of its kind advantage: project owners can set the starting price of their tokens manually and can therefore enable trading without the need of a starting liquidity. This eliminates the need and the risk of any form of presale. On top, the token can benefit from the already existing swap and charting system for free. To be compatible, the token has to meet a set of predetermined rules, which include the buy and sell functions of the $SRG contract, as well as security standards like the everlasting liquidity pool. After meeting the criteria, any project owner / developer can launch with SURGE without the need to ask for permission.

Contract Summary

Documentation Quality

SURGE provides a very poor documentation with standard of solidity base code.

- The technical description is provided unclear and disorganized.

Code Quality

The Overall quality of the basecode is poor.

- Solidity basecode and rules are unclear and disorganized by SURGE.

Test Coverage

Test coverage of the project is 100% (Through Codebase)

Audit Findings Summary

- SWC-107 | It is recommended to use a reentrancy lock, reentrancy weaknesses detected on lines 535, 538, 539, 541, 544, 722, 723, 540, 661, 662, 662, 661, 665 and 33.
- SWC-113 SWC-128 | It is recommended to implement the contract logic to handle failed calls and block gas limit on lines 723 and 661.
- SWC-116 | It is recommended to use oracles for block values as a proxy for time on lines 394, 404, 403 and 503.
- SWC-120 | It is recommended to use external sources of randomness via oracles on lines 397, 507 and 506.

CONCLUSION

We have audited the SURGE project released on January 2023 to find issues and identify potential security vulnerabilities in the SURGE project. This process is used to find technical issues and security loopholes that may be found in smart contracts.

The security audit report yielded unsatisfactory results, discovering medium-risk and low-risk issues.

Writing a contract that does not follow the Solidity style guide can pose a significant risk. The medium and low problems we found in the smart contract are Multiple calls are executed in the same transaction, This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase). Low-risk issue read or write of persistent state following the external call, control flow decision is made based on The block.timestamp environment variable, and potential use of "block.number" as a source of randomness. The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

We were recommended to keep being aware of investing in this risky smart contract.

AUDIT RESULT

Article	Category	Description	Result
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	PASS
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	PASS
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	PASS
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	PASS
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	PASS
Unprotected Ether Withdrawal	SWC-105	Due to missing or insufficient access controls, malicious parties can withdraw from the contract.	PASS
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	PASS
Reentrancy	SWC-107	Check effect interaction pattern should be followed if the code performs recursive call.	ISSUE FOUND
Uninitialized Storage Pointer	SWC-109	Uninitialized local storage variables can point to unexpected storage locations in the contract.	PASS
Assert Violation	SWC-110 SWC-123	Properly functioning code should never reach a failing assert statement.	PASS
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	PASS
Delegate call to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	PASS

DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless required.	ISSUE FOUND
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	PASS
Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	PASS
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	ISSUE FOUND
Signature Unique ID	SWC-117 SWC-121 SWC-122	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.	PASS
Incorrect Constructor Name	SWC-118	Constructors are special functions that are called only once during the contract creation.	PASS
Shadowing State Variable	SWC-119	State variables should not be shadowed.	PASS
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	ISSUE FOUND
Write to Arbitrary Storage Location	SWC-124	The contract is responsible for ensuring that only authorized user or contract accounts may write to sensitive storage locations.	PASS
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order. The rule of thumb is to inherit contracts from more /general/ to more /specific/.	PASS
Insufficient Gas Griefing	SWC-126	Insufficient gas griefing attacks can be performed on contracts which accept data and use it in a sub-call on another contract.	PASS
Arbitrary Jump Function	SWC-127	As Solidity doesnt support pointer arithmetics, it is impossible to change such variable to an arbitrary value.	PASS

Typographical Error	SWC-129	A typographical error can occur for example when the intent of a defined operation is to sum a number to a variable.	PASS
Override control character	SWC-130	Malicious actors can use the Right-To-Left-Override unicode character to force RTL text rendering and confuse users as to the real intent of a contract.	PASS
Unused variables	SWC-131 SWC-135	Unused variables are allowed in Solidity and they do not pose a direct security issue.	PASS
Unexpected Ether balance	SWC-132	Contracts can behave erroneously when they strictly assume a specific Ether balance.	PASS
Hash Collisions Variable	SWC-133	Using abi.encodePacked() with multiple variable length arguments can, in certain situations, lead to a hash collision.	PASS
Hardcoded gas amount	SWC-134	The transfer() and send() functions forward a fixed amount of 2300 gas.	PASS
Unencrypted Private Data	SWC-136	It is a common misconception that private type variables cannot be read.	PASS

SMART CONTRACT ANALYSIS

Started	Thursday Jan 12 2023 05:41:37 GMT+0000 (Coordinated Universal Time)
Finished	Friday Jan 13 2023 05:42:33 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Main Source File	SURGE.sol

Detected Issues

ID	Title	Severity	Status
SWC-113	MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.	medium	acknowledged
SWC-113	MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.	medium	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL	low	acknowledged

SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-107	WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.	low	acknowledged
SWC-116	A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.	low	acknowledged
SWC-116	A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.	low	acknowledged
SWC-116	A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.	low	acknowledged
SWC-116	A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.	low	acknowledged
SWC-120	POTENTIAL USE OF "BLOCK.NUMBER" AS SOURCE OF RANDOMNESS.	low	acknowledged
SWC-120	POTENTIAL USE OF "BLOCK.NUMBER" AS SOURCE OF RANDOMNESS.	low	acknowledged
SWC-120	A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.NUMBER ENVIRONMENT VARIABLE.	low	acknowledged

SWC-113 | MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.

LINE 723

medium SEVERITY

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source File

- SURGE.sol

Locations

```
722   IPancakePair pair = IPancakePair(stablePairAddress);
723   IERC20 token1 = pair.token0() == stableAddress
724   ? IERC20(pair.token1())
725   : IERC20(pair.token0());
726
727
```

SWC-113 | MULTIPLE CALLS ARE EXECUTED IN THE SAME TRANSACTION.

LINE 661

medium SEVERITY

This call is executed following another call within the same transaction. It is possible that the call never gets executed if a prior call fails permanently. This might be caused intentionally by a malicious callee. If possible, refactor the code such that each transaction only executes one external call or make sure that all callees can be trusted (i.e. they're part of your own codebase).

Source File

- SURGE.sol

Locations

```
660   }("");
661   (bool temp2, ) = payable(treasuryWallet).call{
662   value: (taxBalance * treasuryShare) / SHAREDIVISOR
663   }("");
664   assert(temp1 && temp2);
665
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 535

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
534   // subtract full amount from sender
535   _balances[seller] = _balances[seller] - tokenAmount;
536
537   //add tax allowance to be withdrawn and remove from liq the amount of beans taken
by the seller
538   taxBalance = isFeeExempt[msg.sender]
539
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 538

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
537   //add tax allowance to be withdrawn and remove from liq the amount of beans taken
by the seller
538   taxBalance = isFeeExempt[msg.sender]
539   ? taxBalance
540   : taxBalance + amountTax;
541   liquidity = liquidity - amountBNB;
542
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 539

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
538   taxBalance = isFeeExempt[msg.sender]
539   ? taxBalance
540   : taxBalance + amountTax;
541   liquidity = liquidity - amountBNB;
542
543
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 541

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
540   : taxBalance + amountTax;
541   liquidity = liquidity - amountBNB;
542
543   // add tokens back into the contract
544   _balances[address(this)] = _balances[address(this)] + tokenAmount;
545
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 544

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
543   // add tokens back into the contract
544   _balances[address(this)] = _balances[address(this)] + tokenAmount;
545
546   //update volume
547   uint256 cTime = block.timestamp;
548
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 722

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
721   function getBNBPrice() public view returns (uint256) {
722   IPancakePair pair = IPancakePair(stablePairAddress);
723   IERC20 token1 = pair.token0() == stableAddress
724   ? IERC20(pair.token1())
725   : IERC20(pair.token0());
726
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 723

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
722   IPancakePair pair = IPancakePair(stablePairAddress);
723   IERC20 token1 = pair.token0() == stableAddress
724   ? IERC20(pair.token1())
725   : IERC20(pair.token0());
726
727
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 540

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
539   ? taxBalance
540   : taxBalance + amountTax;
541   liquidity = liquidity - amountBNB;
542
543   // add tokens back into the contract
544
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 661

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
660   }("");
661   (bool temp2, ) = payable(treasuryWallet).call{
662   value: (taxBalance * treasuryShare) / SHAREDIVISOR
663   }("");
664   assert(temp1 && temp2);
665
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 662

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
661   (bool temp2, ) = payable(treasuryWallet).call{
662   value: (taxBalance * treasuryShare) / SHAREDIVISOR
663   }("");
664   assert(temp1 && temp2);
665   taxBalance = 0;
666
```

SWC-107 | READ OF PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 662

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
661   (bool temp2, ) = payable(treasuryWallet).call{
662   value: (taxBalance * treasuryShare) / SHAREDIVISOR
663   }("");
664   assert(temp1 && temp2);
665   taxBalance = 0;
666
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL

LINE 661

low SEVERITY

The contract account state is accessed after an external call to a fixed address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
660   }("");
661   (bool temp2, ) = payable(treasuryWallet).call{
662   value: (taxBalance * treasuryShare) / SHAREDIVISOR
663   }("");
664   assert(temp1 && temp2);
665
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 665

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
664   assert(temp1 && temp2);
665   taxBalance = 0;
666   }
667
668   function getTokenAmountOut(uint256 amountBNBIn)
669
```

SWC-107 | WRITE TO PERSISTENT STATE FOLLOWING EXTERNAL CALL.

LINE 33

low SEVERITY

The contract account state is accessed after an external call. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.

Source File

- SURGE.sol

Locations

```
32  _;
33  _status = _NOT_ENTERED;
34  }
35  }
36
37
```

SWC-116 | A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.

LINE 394

low SEVERITY

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source File

- SURGE.sol

Locations

```
393   // deadline requirement
394   require(deadline >= block.timestamp, "Deadline EXPIRED");
395
396   // Frontrun Guard
397   _lastBuyBlock[msg.sender] = block.number;
398
```

SWC-116 | A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.

LINE 404

low SEVERITY

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source File

- SURGE.sol

Locations

```
403   require(
404   block.timestamp >= TRADE_OPEN_TIME ||
405   msg.sender == MIGRATION_WALLET,
406   "Trading is not Open"
407   );
408
```

SWC-116 | A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.

LINE 403

low SEVERITY

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source File

- SURGE.sol

Locations

```
402   // check if trading is open or whether the buying wallet is the migration one
403   require(
404   block.timestamp >= TRADE_OPEN_TIME ||
405   msg.sender == MIGRATION_WALLET,
406   "Trading is not Open"
407
```

SWC-116 | A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.TIMESTAMP ENVIRONMENT VARIABLE.

LINE 503

low SEVERITY

The block.timestamp environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source File

- SURGE.sol

Locations

```
502   // deadline requirement
503   require(deadline >= block.timestamp, "Deadline EXPIRED");
504
505   //Frontrun Guard
506   require(
507
```

SWC-120 | POTENTIAL USE OF "BLOCK.NUMBER" AS SOURCE OF RANDOMNESS.

LINE 397

low SEVERITY

The environment variable "block.number" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source File

- SURGE.sol

Locations

```
396   // Frontrun Guard
397   _lastBuyBlock[msg.sender] = block.number;
398
399   // liquidity is set
400   require(liquidity > 0, "The token has no liquidity");
401
```

SWC-120 | POTENTIAL USE OF "BLOCK.NUMBER" AS SOURCE OF RANDOMNESS.

LINE 507

low SEVERITY

The environment variable "block.number" looks like it might be used as a source of randomness. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source File

- SURGE.sol

Locations

```
506   require(
507   _lastBuyBlock[msg.sender] != block.number,
508   "Buying and selling in the same block is not allowed!"
509   );
510
511
```

SWC-120 | A CONTROL FLOW DECISION IS MADE BASED ON THE BLOCK.NUMBER ENVIRONMENT VARIABLE.

LINE 506

low SEVERITY

The block.number environment variable is used to determine a control flow decision. Note that the values of variables like coinbase, gaslimit, block number and timestamp are predictable and can be manipulated by a malicious miner. Also keep in mind that attackers know hashes of earlier blocks. Don't use any of those environment variables as sources of randomness and be aware that use of these variables introduces a certain level of trust into miners.

Source File

- SURGE.sol

Locations

```
505   //Frontrun Guard
506   require(
507   _lastBuyBlock[msg.sender] != block.number,
508   "Buying and selling in the same block is not allowed!"
509   );
510
```

DISCLAIMER

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to, or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without Sysfixed's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Sysfixed to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model, or legal compliance.

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

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and Sysfixed and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (Sysfixed) owe no duty of care.

ABOUT US

Sysfixed is a blockchain security certification organization established in 2021 with the objective to provide smart contract security services and verify their correctness in blockchain-based protocols. Sysfixed automatically scans for security vulnerabilities in Ethereum and other EVM-based blockchain smart contracts. Sysfixed a comprehensive range of analysis techniques—including static analysis, dynamic analysis, and symbolic execution—can accurately detect security vulnerabilities to provide an in-depth analysis report. With a vibrant ecosystem of world-class integration partners that amplify developer productivity, Sysfixed can be utilized in all phases of your project's lifecycle. Our team of security experts is dedicated to the research and improvement of our tools and techniques used to fortify your code.