Cyberscope

Audit Report

Surge Protocol

April 2023

Network	BSC, ETH, ARBITRUM
Address	0x9f19c8e321bD14345b797d43E01f0eED030F5Bff 0xcD682EF09d07668d49A8103ddD65Ff54AebFbfDe 0x31aD8255CB8E428E8B0f6Ab6a6f44804642720aF
Audited by	© cyberscope

Table of Contents

Review

Contract Name	SURGE
Compiler Version	v0.8.17+commit.8df45f5f
Optimization	200 runs
Explorer	https://bscscan.com/address/0x9f19c8e321bd14345b797d43e01f0eed030f5bff https://etherscan.io/address/0xcd682ef09d07668d49a8103ddd65ff54aebfbfde https://arbiscan.io/address/0x31ad8255cb8e428e8b0f6ab6a6f44804642720af
Address	0x9f19c8e321bd14345b797d43e01f0eed030f5bff 0xcd682ef09d07668d49a8103ddd65ff54aebfbfde 0x31ad8255cb8e428e8b0f6ab6a6f44804642720af
Network	BSC, ETH, ARBITRUM
Symbol	SRG
Decimals	9
Total Supply	100,000,000

Audit Updates

Initial Audit	18 Apr 2023

Source Files

Filename	SHA256
SURGE.sol	e51517635f0270b74701ed1470b25e6a95ae5eb63acceb049dd6ad1a1abb0b15

Overview

SURGE is a novel token that contains a built-in swap mechanism. The entire total supply is initially issued to the contract address. Thus, the team or contract owner does not have initial shares. The swapping mechanism started with a ratio of 20 ETH - total supply similar to a fair launch. The main difference with an ordinary fair launch is that the owner does not have to supply the initial liquidity. The price of the build-in swap is adjusted on every buy or sell operation. If someone wants to list in a Decentralized Exchange (DEX), it should first buy from the built-in swap and then provide liquidity to the DEX. The process to launch on the DEX is to deploy a contract that is compatible with the protocol (has a swapping feature compatible with SRG and stores the LP inside the smart contract). After deploying, the liquidity is pre-set like it was in SRG. No initial funding needs to be provided by the dev teams, they can instantly enable trading after deploying. Investors then can buy those tokens by using SRG

Findings Breakdown

Critical	0
Medium	0
Minor / Informative	8

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	8	0	0	0

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
	ST	Stops Transactions	Passed
	OCTD	Transfers Contract's Tokens	Passed
	OTUT	Transfers User's Tokens	Passed
	ELFM	Exceeds Fees Limit	Passed
	ULTW	Transfers Liquidity to Team Wallet	Passed
	MT	Mints Tokens	Passed
	BT	Burns Tokens	Passed
	BC	Blacklists Addresses	Passed

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
●	DDP	Decimal Division Precision	Unresolved
●	CR	Code Repetition	Unresolved
●	PRL	Potential Reentrance Leverage	Unresolved
●	MU	Modifiers Usage	Unresolved
●	AAO	Accumulated Amount Overflow	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	SURGE.sol#L654
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
(bool temp1, ) = payable(teamWallet).call{
    value: (taxBalance * teamShare) / SHAREDIVISOR
}("");
(bool temp2, ) = payable(treasuryWallet).call{
    value: (taxBalance * treasuryShare) / SHAREDIVISOR
}("");
```

Recommendation

The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

CR - Code Repetition

Criticality	Minor / Informative
Location	SURGE.sol#L436,543
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

The contract repeats the same code twice. Lines 436-472 and 543-573 are the same with the exception of the amount that is multiplied with the BNB price and one extra case that is handled when the operation `candleStickData[cTime].open == 0` is true. As a result, the contract uses a decent amount of code twice.

The code segment below demonstrates the parts that differ at the aforementioned lines.

```
uint256 cTime = block.timestamp;
uint256 dollarBuy = msg.value * getBNBPrice();
...
if (candleStickData[cTime].open == 0) {
    if (totalTx == 1) {
        candleStickData[cTime].open =
            ((liquidity - bnbAmount) / (_totalSupply)) *
            getBNBPrice();
    }
}
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

PRL - Potential Reentrance Leverage

Criticality	Minor / Informative
Location	SURGE.sol#L525
Status	Unresolved

Description

As part of the `_sell` method, the contract transfers ETH to the user. Since the user can be any address, the address could be a contract that implements the fallback methods. The contract will still have the entire balance during the fallback execution since the amount has not yet been subtracted.

In the first phase of the token launch this is not applicable since most of the tokens belong to the contract address and the internal methods are guarded by a re-entrance mutex. Thus, there are no remaining tokens to be listed in DApps, such as DEXs, etc. Later, if the users start using lending platforms like CEXs, DEXs, Fast Loans, etc then a user could leverage issues that will be produced by the market arbitrage by implementing the fallback method.

```
(bool successful, ) = isFeeExempt[msg.sender]
    ? payable(seller).call{value: amountBNB}("")
    : payable(seller).call{value: BNBToSend}("");
require(successful, "BNB/ETH transfer failed");

// subtract full amount from sender
_balances[seller] = _balances[seller] - tokenAmount;
```

Recommendation

The team is advised to prevent the re-entrance exploit as part of the solidity best practices. Some suggestions are:

- Not allow contract addresses to receive funds.
- Add a locker/mutex in the transfer method scope.

- Transfer the funds as the last statement of the transfer method, so that the balance will have been subtracted during the re-entrance phase.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	SURGE.sol#L390,499
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(deadline >= block.timestamp, "Deadline EXPIRED");
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

AAO - Accumulated Amount Overflow

Criticality	Minor / Informative
Location	SURGE.sol#L438,545
Status	Unresolved

Description

The contract is using variables to accumulate values. The contract could lead to an overflow when the total value of a variable exceeds the maximum value that can be stored in that variable's data type. This can happen when an accumulated value is updated repeatedly over time, and the value grows beyond the maximum value that can be represented by the data type.

```
totalVolume += dollarBuy;
totalVolume += dollarSell;
```

Recommendation

The team is advised to carefully investigate the usage of the variables that accumulate value. A suggestion is to add checks to the code to ensure that the value of a variable does not exceed the maximum value that can be stored in its data type.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	SURGE.sol#L144,145,146,147,150,153,154,195,383,493
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
string private constant _name = "SURGE"
string private constant _symbol = "SRG"
uint8 private constant _decimals = 9
uint256 private constant _decMultiplier = 10**_decimals
uint256 public constant _totalSupply = 10**8 * _decMultiplier
mapping(address => uint256) public _balances
mapping(address => mapping(address => uint256)) internal _allowances

struct candleStick {
        uint256 time;
        uint256 open;
        uint256 close;
        uint256 high;
        uint256 low;
    }

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	SURGE.sol#L406,449,476,604
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause a loss of prediction.

```
return
            ((_balances[holder] * liquidity) / _balances[address(this)]) *
            getBNBPrice()
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

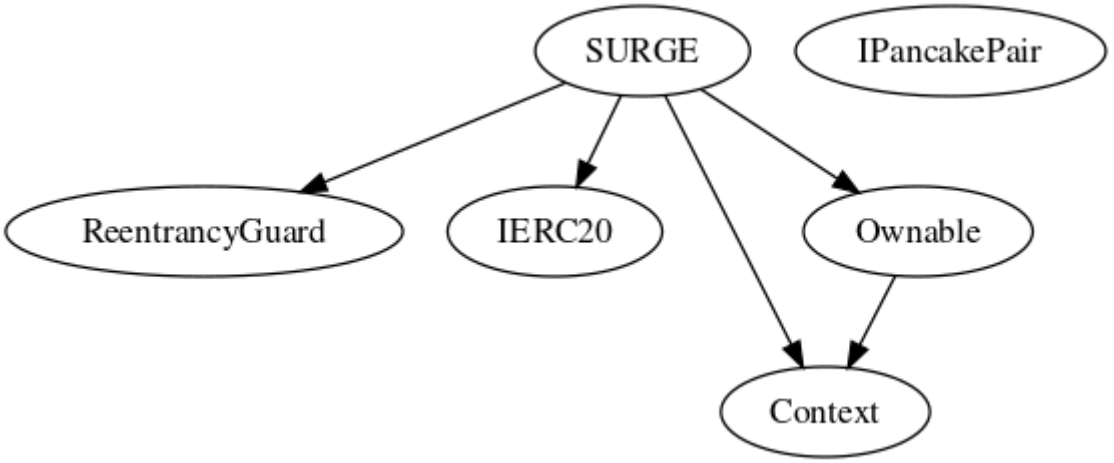
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
ReentrancyGuard	Implementation			
		Public	✓	-
IPancakePair	Interface			
	token0	External		-
	token1	External		-
	getReserves	External		-
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
	decimals	External		-
Context	Implementation			

	_msgSender	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
SURGE	Implementation	IERC20, Context, Ownable, ReentrancyGuard		
		Public	✓	-
	totalSupply	External		-
	balanceOf	Public		-
	allowance	External		-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	approve	Public	✓	-
	approveMax	External	✓	-
	getCirculatingSupply	Public		-
	changeWalletLimit	External	✓	onlyOwner
	changeIsFeeExempt	External	✓	onlyOwner
	changeIsTxLimitExempt	External	✓	onlyOwner

	transfer	External	✓	-
	transferFrom	External	✓	-
	_transferFrom	Internal	✓	
	_spendAllowance	Internal	✓	
	_approve	Internal	✓	
	_buy	Public	Payable	nonReentrant
	buy	Internal	✓	
	_sell	Public	✓	nonReentrant
	getLiquidity	Public		-
	getValueOfHoldings	Public		-
	changeFees	External	✓	onlyOwner
	changeTaxDistribution	External	✓	onlyOwner
	changeFeeReceivers	External	✓	onlyOwner
	withdrawTaxBalance	External	✓	nonReentrant onlyOwner
	getTokenAmountOut	External		-
	getBNBAmountOut	Public		-
	addLiquidity	External	Payable	onlyOwner
	getMarketCap	External		-
	changeStablePair	External	✓	onlyOwner
	getBNBPrice	Public		-
	calculatePrice	Public		-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
SURGE
<Constructor>
totalSupply
balanceOf
allowance
name
symbol
decimals
changeWalletLimit
changeIsFeeExempt
changeIsTxLimitExempt
approveMax
approve
transfer
_transferFrom
type
transferFrom
_spendAllowance
_approve
getValueOfHoldings
_buy
buy
getLiquidity
getMarketCap
changeFees
changeTaxDistribution
changeFeeReceivers
getBNBPrice
_sell
getCirculatingSupply
getTokenAmountOut
calculatePrice
getBNBAmountOut
addLiquidity
withdrawTaxBalance
changeStablePair
Ownable
<Constructor>
owner
onlyOwner
renounceOwnership
transferOwnership
Context
_msgSender
payable
IPancakePair (iface)
token0
token1
getReserves
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
decimals
ReentrancyGuard
<Constructor>
nonReentrant

Summary

Surge Protocol contract implements a token mechanism. This audit investigates security issues, business logic concerns, and potential improvements. Surge Protocol is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler errors or critical issues. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions. There is also a limit of max 10% buy and sell fees for the built-in buy/sell functionality of the contract.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. 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 Cyberscope's prior written consent. 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 regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document 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 report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment 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.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

https://www.cyberscope.io