Elk

Smart Contract Audit Report

SYSFIXED

08 Apr 2022

TABLE OF CONTENTS

Audited Details

- Audited Project
- Blockchain
- Addresses
- Project Website
- Codebase

Summary

- Contract Summary
- Audit Findings Summary
- Vulnerabilities Summary

Conclusion

Audit Results

Smart Contract Analysis

- Detected Vulnerabilities

Disclaimer

About Us

AUDITED DETAILS

Audited Project

Project name	Token ticker	Blockchain
Elk	Elk	Binance Smart Chain

Addresses

Contract address	0xeeeeeb57642040be42185f49c52f7e9b38f8eeee
Contract deployer address	0x6bc5Fc9d0D908eF8444A7d8f6A7E1A7050A82084

Project Website

https://elk.finance/

Codebase

https://bscscan.com/address/0xeeeeeb57642040be42185f49c52f7e9b38f8eeee#code

SUMMARY

Cross-chain value exchange is the next battleground for cryptocurrencies and the next major hurdle for adoption. Moving or exchanging tokens across chains is an excruciating and expensive process. Elk.Finance aims to make this process easy and intuitive. We aim to be the Forex market for the decentralized economy, providing sub-second value transfers across chains. "Any chain, anytime, anywhere" is our motto. Join us as we embark on this exciting adventure! The ELK Token The ELK token is an ERC20-compatible utility that underpins the Elk.Finance ecosystem. Central to the Elk network's design is that all liquidity pools pair exchange tokens with ELK. This design decision allows for a sub-second transfer of value across chains and provides deeper liquidity for pools, reducing slippage and fees. ELK also doubles as the governance token for the Elk network.

Contract Summary

Documentation Quality

Elk provides a very good documentation with standard of solidity base code.

- The technical description is provided clearly and structured and also dont have any high risk issue.

Code Quality

The Overall quality of the basecode is standard.

- Standard solidity basecode and rules are already followed by Elk with the discovery of several low issues.

Test Coverage

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

Audit Findings Summary

- SWC-103 | Pragma statements can be allowed to float when a contract is intended on lines 11, 254, 319, 389, 621, 727, 790, 836, 882, 932, 959, 1037, 1122, 1152, 1537, 1626, 1877 and 2072.
- SWC-120 | It is recommended to use external sources of randomness via oracles on lines 1698, 1711, 1856 and 1859.

CONCLUSION

We have audited the Elk Project released on April 2022 to discover issues and identify potential security vulnerabilities in Elk Project. This process is used to find technical issues and security loopholes which might be found in the smart contract.

The security audit report provides a satisfactory result with some low-risk issues.

The issues found in the Elk smart contract code do not pose a considerable risk. The writing of the contract is close to the standard of writing contracts in general. The low-risk issues found are some a floating pragma is set, and potential use of "block.number" as a source of randomness. 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 the use of these variables introduces a certain level of trust in miners. The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

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.	ISSUE FOUND
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.	PASS
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.	PASS
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.	PASS
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 Apr 07 2022 11:50:37 GMT+0000 (Coordinated Universal Time)
Finished	Friday Apr 08 2022 18:56:32 GMT+0000 (Coordinated Universal Time)
Mode	Standard
Main Source File	Elk.sol

Detected Issues

ID	Title	Severity	Status
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged

SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	low	acknowledged
SWC-103	A FLOATING PRAGMA IS SET.	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	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-103 | A FLOATING PRAGMA IS SET.

LINE 11

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
10
11   pragma solidity ^0.8.0;
12
13   /**
14   * @dev Wrappers over Solidity's uintXX/intXX casting operators with added overflow
15
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 254

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
253   // OpenZeppelin Contracts (last updated v4.5.0) (governance/utils/IVotes.sol)
254   pragma solidity ^0.8.0;
255
256   /**
257    * @dev Common interface for {ERC20Votes}, {ERC721Votes}, and other {Votes}-enabled
contracts.
258
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 319

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
318
319   pragma solidity ^0.8.0;
320
321   /**
322   * @dev String operations.
323
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 389

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
388
389   pragma solidity ^0.8.0;
390
391
392   /**
393
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 621

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
620
621   pragma solidity ^0.8.0;
622
623
624   /**
625
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 727

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
726
727   pragma solidity ^0.8.0;
728
729   /**
730   * @dev Interface of the ERC20 Permit extension allowing approvals to be made via
signatures, as defined in
731
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 790

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
789
790   pragma solidity ^0.8.0;
791
792   /**
793   * @title Counters
794
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 836

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
835
836   pragma solidity ^0.8.0;
837
838   /**
839   * @dev Standard math utilities missing in the Solidity language.
840
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 882

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
881
882    pragma solidity ^0.8.0;
883
884
885    /**
886
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 932

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
931
932   pragma solidity ^0.8.0;
933
934   /**
935   * @dev Provides information about the current execution context, including the
936
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 959

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
958
959   pragma solidity ^0.8.0;
960
961
962   /**
963
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 1037

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
1036
1037   pragma solidity ^0.8.0;
1038
1039   /**
1040   * @dev Interface of the ERC20 standard as defined in the EIP.
1041
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 1122

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
1121
1122   pragma solidity ^0.8.0;
1123
1124
1125   /**
1126
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 1152

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
1151
1152    pragma solidity ^0.8.0;
1153
1154
1155
1156
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 1537

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
1536
1537    pragma solidity ^0.8.0;
1538
1539
1540
1541
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 1626

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
1625
1626   pragma solidity ^0.8.0;
1627
1628
1629
1630
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 1877

low SEVERITY

The current pragma Solidity directive is ""^0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
1876
1877    pragma solidity ^0.8.0;
1878
1879
1880
1881
```

SWC-103 | A FLOATING PRAGMA IS SET.

LINE 2072

low SEVERITY

The current pragma Solidity directive is "">=0.8.0"". It is recommended to specify a fixed compiler version to ensure that the bytecode produced does not vary between builds. This is especially important if you rely on bytecode-level verification of the code.

Source File

- Elk.sol

Locations

```
2071
2072   pragma solidity >=0.8.0;
2073
2074
2075
2076
```

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

LINE 1698

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

- Elk.sol

Locations

```
1697   function getPastVotes(address account, uint256 blockNumber) public view virtual
override returns (uint256) {
1698   require(blockNumber < block.number, "ERC20Votes: block not yet mined");
1699   return _checkpointsLookup(_checkpoints[account], blockNumber);
1700   }
1701
1702
```

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

LINE 1711

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

- Elk.sol

Locations

```
1710   function getPastTotalSupply(uint256 blockNumber) public view virtual override
returns (uint256) {
1711   require(blockNumber < block.number, "ERC20Votes: block not yet mined");
1712   return _checkpointsLookup(_totalSupplyCheckpoints, blockNumber);
1713   }
1714
1715
```

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

LINE 1856

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

- Elk.sol

Locations

```
1855
1856   if (pos > 0 && ckpts[pos - 1].fromBlock == block.number) {
1857   ckpts[pos - 1].votes = SafeCast.toUint224(newWeight);
1858   } else {
1859   ckpts.push(Checkpoint({fromBlock: SafeCast.toUint32(block.number), votes:
SafeCast.toUint224(newWeight)}));
1860
```

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

LINE 1859

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

- Elk.sol

Locations

```
1858   } else {
1859   ckpts.push(Checkpoint({fromBlock: SafeCast.toUint32(block.number), votes:
SafeCast.toUint224(newWeight)}));
1860   }
1861   }
1862
1863
```

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.