Cyberscope
A TAC Security Company

Audit Report
BSD

September 2025

Network	ARBITRUM
Address	0x9addec9052d757fa5c138e257417edbc1b03f580
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	AME	Address Manipulation Exploit	Unresolved
●	IVA	Inconsistent Vesting Amount	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSRS	Redundant SafeMath Require Statement	Unresolved
●	UV	Unused Variables	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L15	Local Scope Variable Shadowing	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Contract Name	Token
Compiler Version	v0.8.0+commit.c7dfd78e
Optimization	200 runs
Explorer	https://arbiscan.io/address/0x9addec9052d757fa5c138e257417edbc1b03f580
Address	0x9addec9052d757fa5c138e257417edbc1b03f580
Network	ARBITRUM
Symbol	BSD
Decimals	18
Total Supply	1.000.000.000

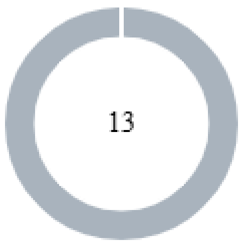
Audit Updates

Initial Audit	10 Sep 2025

Source Files

Filename	SHA256
contracts/Token.sol	faf642c54ad3c8d67df966147673d1d0a71e0356bcd655c54700a9bce53413c1

Findings Breakdown



●	Critical	0
●	Medium	0
●	Minor / Informative	13

	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	13	0	0	0

AME - Address Manipulation Exploit

Criticality	Minor / Informative
Location	contracts/Token.sol#L2506
Status	Unresolved

Description

The contract's design includes functions that accept external contract addresses as parameters without performing adequate validation or authenticity checks. This lack of verification introduces a significant security risk, as input addresses could be controlled by attackers and point to malicious contracts. Such vulnerabilities could enable attackers to exploit these functions, potentially leading to unauthorized actions or the execution of malicious code under the guise of legitimate operations.

For example, since the `release` function of the `TokenVesting` contract accepts any token as input, external parties can send different tokens than the one intended, and if such a token has different functionality (e.g., rebasing) it can break the consistency of the vesting logic or execute unintended code.

Shell
```
function release(address token) public {
     ...
     IERC20(token).safeTransfer(beneficiary,
unreleased);
     ...
}
```

Recommendation

To mitigate this risk and enhance the contract's security posture, it is imperative to incorporate comprehensive validation mechanisms for any external contract addresses passed as parameters to functions. This could include checks against a whitelist of approved addresses, verification that the address implements a specific contract interface or other methods that confirm the legitimacy and integrity of the external contract. Implementing such validations helps prevent malicious exploits and ensures that only trusted contracts can interact with sensitive functions.

IVA - Inconsistent Vesting Amount

Criticality	Minor / Informative
Location	contracts/Token.sol#L2531
Status	Unresolved

Description

The TokenVesting contract calculates vested tokens using the balanceOf( ) method. This allows anyone to transfer tokens directly to the vesting contract, artificially increasing its balance. As a result more tokens may become releasable than intended.

Shell

```
uint256 currentBalance =
IERC20(token).balanceOf(address(this));
```

Recommendation

To ensure correctness, the team should compute vesting from a fixed allocation recorded at funding time rather than the live balance; treat any subsequent inbound tokens as non-vested surplus that is excluded from vest calculations and can be swept by an authorized party.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	contracts/Token.sol#L2493,2494,2509
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

Shell
```
require(_beneficiary != address(0))
require(_cliff <= _duration)

require(unreleased > 0)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	contracts/Token.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

Shell
```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RSRS - Redundant SafeMath Require Statement

Criticality	Minor / Informative
Location	contracts/Token.sol#L2259
Status	Unresolved

Description

The contract utilizes a `require` statement within the `add` function aiming to prevent overflow errors. This function is designed based on the SafeMath library's principles. In Solidity version 0.8.0 and later, arithmetic operations revert on overflow and underflow, making the overflow check within the function redundant. This redundancy could lead to extra gas costs and increased complexity without providing additional security.

Shell
```
function add(uint256 a, uint256 b) internal pure
returns (uint256) {
    uint256 c = a + b;
    require(c >= a, "SafeMath: addition
overflow");

    return c;

}
```

Recommendation

It is recommended to remove the `require` statement from the add function since the contract is using a Solidity pragma version equal to or greater than 0.8.0. By doing so, the contract will leverage the built-in overflow and underflow checks provided by the Solidity language itself, simplifying the code and reducing gas consumption. This change will uphold the contract's integrity in handling arithmetic operations while optimizing for efficiency and cost-effectiveness.

UV - Unused Variables

Criticality	Minor / Informative
Location	contracts/Token.sol#L2490,2553
Status	Unresolved

Description

The constructor of `TokenVesting` has a `_cliff` parameter. This parameter is used to set the `cliff` state variable that restricts users from releasing tokens before a certain time passes. However in the `Token` contract all the `_cliff` parameters are added as 0 so the cliff will not be utilized therefore it is redundant.

```Shell
constructor(**args**, uint256 _cliff)
```

Similarly the `total` state variable in the `Token` contract is defined but is never used.

```Shell
uint256 total = 1e9;
```

Recommendation

The team should ensure that contracts are optimized by removing unused variables such `_cliff`.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/Token.sol#L2551,2552,2553
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

Shell
```shell
name_ = "BSD";

symbol_ = "BSD";
_total = 1e9;
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/Token.sol#L1688,1984,1985,1986,1988,1989,1990,2158,2213
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.

```
Shell
function DOMAIN_SEPARATOR() external view returns
(bytes32);

 private immutable _CACHED_DOMAIN_SEPARATOR;
 private immutable _CACHED_CHAIN_ID;
...
 private immutable _HASHED_NAME;
 private immutable _HASHED_VERSION;

...
```

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/stable/style-guide.html#naming-conventions.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	contracts/Token.sol#L1091,1100
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

Shell
```
function _setupRole(bytes32 role, address account)
internal virtual {
        _grantRole(role, account);
    }

function _setRoleAdmin(bytes32 role, bytes32
adminRole) internal virtual {
        bytes32 previousAdminRole =
getRoleAdmin(role);
        _roles[role].adminRole = adminRole;
        emit RoleAdminChanged(role,
previousAdminRole, adminRole);

    }
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	contracts/Token.sol#L2165
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

Shell
```
constructor(string memory name) EIP712(name, "1")
{}
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	contracts/Token.sol#L1456,1541,1760
Status	Unresolved

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
Shell
assembly {
            result := store
        }

assembly {
                r := mload(add(signature, 0x20))
                s := mload(add(signature, 0x40))
                v := byte(0, mload(add(signature,
0x60)))
            }
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	contracts/Token.sol#L7,98,126,152,559,597,702,736,816,843,874,1139,1553,1633,1695,1959,2085,2130,2231,2401,2455,2544
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

Shell
```
pragma solidity ^0.8.0;
pragma solidity ^0.8.0;
pragma solidity >=0.6.0 <0.9.0;
...
pragma solidity >=0.6.0;
pragma solidity 0.8.0;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	contracts/Token.sol#L98
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

Shell
```
pragma solidity ^0.8.0;
```

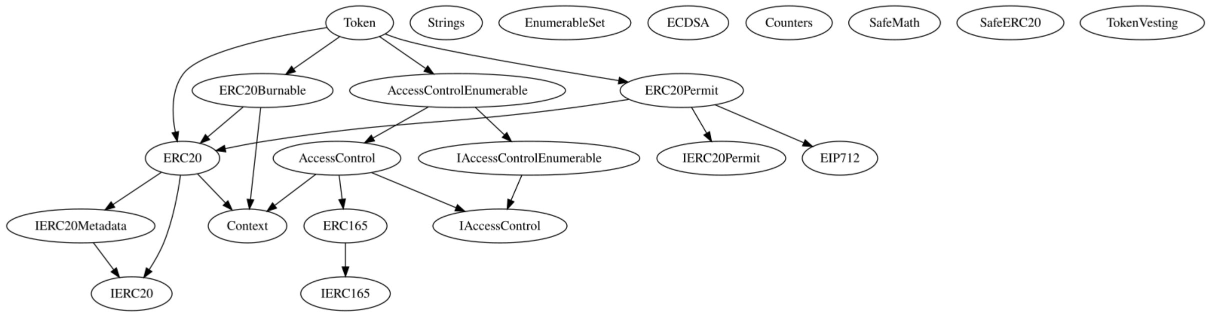
Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

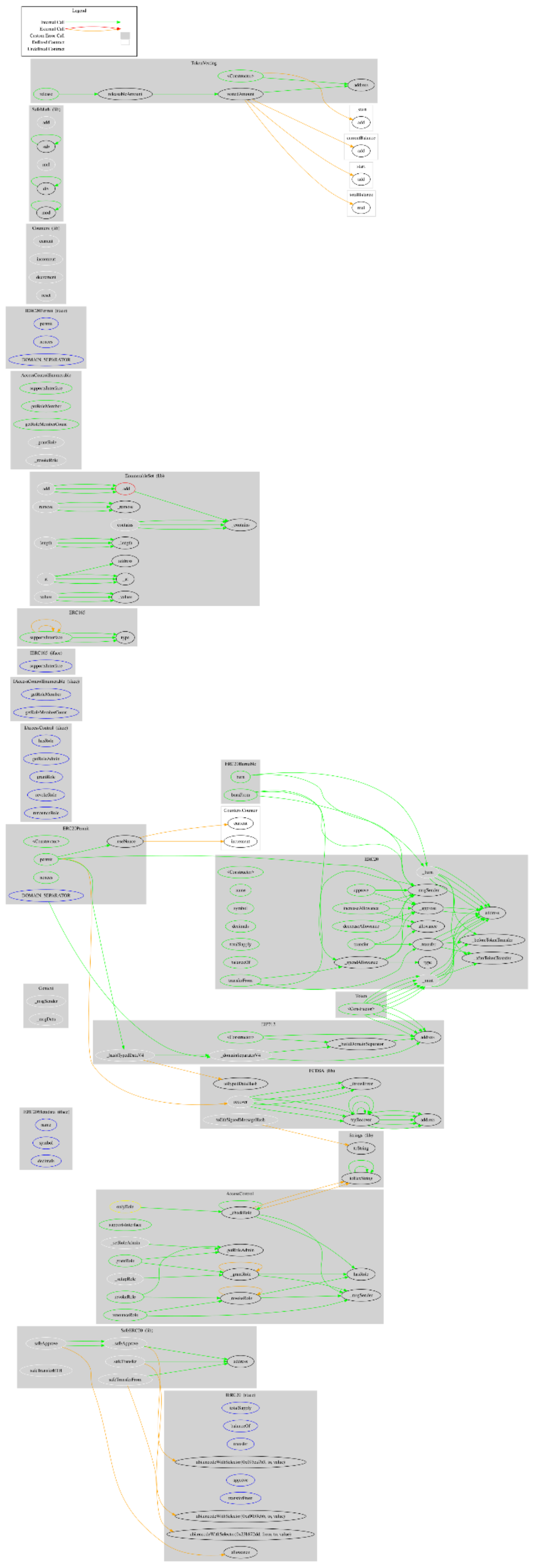
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
TokenVesting	Implementation			
		Public	✓	-
	release	Public	✓	-
	releasableAmount	Public		-
	vestedAmount	Public		-
Token	Implementation	ERC20, ERC20Burnable, AccessControlEnumerable, ERC20Permit		
		Public	✓	ERC20 ERC20Permit

Inheritance Graph



Flow Graph



Summary

BSD contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. BSD is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error 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.

Disclaimer

About Cyberscope

Cyberscope is a TAC 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.

A *TAC Security* Company

The Cyberscope team

cyberscope.io