Cyberscope

Audit Report

Apeswap

January 2024

Network	ETH
Address	0x92df60c51c710a1b1c20e42d85e221f3a1bfc7f2
Audited by	© cyberscope

Table of Contents

Review

Contract Name	AnyswapV6ERC20
Compiler Version	v0.8.10+commit.fc410830
Optimization	200 runs
Explorer	https://etherscan.io/address/0x92df60c51c710a1b1c20e42d85e221f3a1bfc7f2
Address	0x92df60c51c710a1b1c20e42d85e221f3a1bfc7f2
Network	ETH
Symbol	BANANA
Decimals	18
Total Supply	80,025.924

Audit Updates

Initial Audit	13 Jan 2024

Source Files

Filename	SHA256
AnyswapV6ERC20.sol	01ea484ccbf21433fa82d4aba3d779761ddb497dd0f1eeeec798427c2df812b1

Overview

The AnyswapV6ERC20 contract is an advanced implementation of the ERC20 token standard, integrating features from `ERC2612` and `ERC677` standards. It is designed to facilitate cross-chain token swaps and interactions with smart contracts in a more efficient and flexible manner.

AnyswapV6ERC20 Contract Functionality

Token Operations and Management

At its core, the contract manages the AnyswapV6ERC20 tokens, including minting, burning, and transferring tokens. The minting function allows authorized addresses (minters) to create new tokens, while the burning function enables the destruction of tokens from a specified address. This mechanism is crucial for maintaining the token's supply in accordance with the underlying assets, especially in cross-chain scenarios.

Cross-Chain Swapping

The contract introduces unique functionalities for cross-chain token swaps, represented by the `Swapin` and `Swapout` functions. Swapin is used to mint AnyswapV6ERC20 tokens on one blockchain in response to a deposit on another chain, while Swapout allows users to burn AnyswapV6ERC20 tokens in exchange for the underlying asset on the original chain. This feature is essential for enabling seamless asset transfers across blockchain networks.

Enhanced Transfer Mechanisms

In addition to standard ERC20 transfer and transferFrom methods, the contract implements `transferAndCall` functionality, derived from the `ERC677` standard. This method allows tokens to be transferred to a contract address and simultaneously calls a function on the receiving contract. This feature facilitates more complex interactions with contracts in a single transaction, enhancing the token's utility in decentralized applications.

Transfers Permit

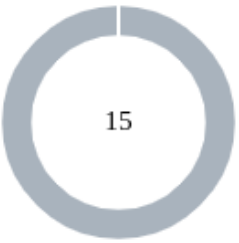
The AnyswapV6ERC20 contract incorporates the `ERC2612` standard's permit function, enabling off-chain signed approvals. This feature allows token holders to authorize transfers or other actions through a signature, rather than executing an on-chain transaction. It obviates the need for a blocking approved transaction before programmatic interactions with the token can occur. This is particularly useful in DeFi applications, where multiple approvals are often required. The permit function utilizes the ERC712 protocol, ensuring user safety and wallet compatibility by simplifying the process of signing data with private keys.

Administrative Controls

The contract includes administrative functions to manage its operation, such as setting minters, changing the vault address (the primary controller), and enabling or disabling certain features. These controls ensure the flexibility and adaptability of the token to different use cases and governance models.

Overall, the AnyswapV6ERC20 contract represents a sophisticated and versatile implementation of the ERC20 token standard, tailored for cross-chain interoperability, enhanced contract interactions, efficient transaction approval processes, and meta-transaction capabilities.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	15

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	15	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CR	Code Repetition	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	MVN	Misleading Variable Naming	Unresolved
	MEM	Missing Error Messages	Unresolved
	MEE	Missing Events Emission	Unresolved
	RSW	Redundant Storage Writes	Unresolved
	TUU	Time Units Usage	Unresolved
	VDB	Vault Delay Bypass	Unresolved
	L02	State Variables could be Declared Constant	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L09	Dead Code Elimination	Unresolved
	L16	Validate Variable Setters	Unresolved
	L17	Usage of Solidity Assembly	Unresolved

●	L19	Stable Compiler Version	Unresolved

CR - Code Repetition

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L481,533,555,586
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. Specifically the `ransferWithPermit` , `transfer` , `transferFrom` , and `transferAndCall` functions share similar code segments.

```
   function transferWithPermit(address target, address to, uint256
value, uint256 deadline, uint8 v, bytes32 r, bytes32 s) external
override returns (bool) {
        require(block.timestamp <= deadline, "AnyswapV3ERC20: Expired
permit");

        bytes32 hashStruct = keccak256(
            abi.encode(
                TRANSFER_TYPEHASH,
                target,
                to,
                value,
                nonces[target]++,
                deadline));

        require(verifyEIP712(target, hashStruct, v, r, s) ||
verifyPersonalSign(target, hashStruct, v, r, s));

        require(to != address(0) || to != address(this));

        uint256 balance = balanceOf[target];
        require(balance >= value, "AnyswapV3ERC20: transfer amount
exceeds balance");

        balanceOf[target] = balance - value;
        balanceOf[to] += value;
        emit Transfer(target, to, value);

        return true;
    }

 function transfer(address to, uint256 value) external override returns
(bool) {
        ...
    }

    function transferFrom(address from, address to, uint256 value)
external override returns (bool) {
        ...
    }


    function transferAndCall(address to, uint value, bytes calldata
data) external override returns (bool) {
        ...
    }
```

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.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L274,279
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Specifically, addresses designated as minters, as indicated by the `onlyAuth` modifier, possess the authority to `mint` and `burn` tokens. This authority extends to any address passed as the `from` parameter in these functions, or as the `account` parameter in the `Swapin` function. Such a level of control concentrated in the hands of a few addresses raises concerns about the centralization, as these minters can unilaterally influence the token supply by minting new tokens or burning existing ones to or from any account.

```
    function mint(address to, uint256 amount) external onlyAuth returns
(bool) {
        _mint(to, amount);
        return true;
    }

    function burn(address from, uint256 amount) external onlyAuth
returns (bool) {
        require(from != address(0), "AnyswapV3ERC20: address(0x0)");
        _burn(from, amount);
        return true;
    }

    function Swapin(bytes32 txhash, address account, uint256 amount)
public onlyAuth returns (bool) {
        _mint(account, amount);
        emit LogSwapin(txhash, account, amount);
        return true;
    }
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L320,323,325,326,327
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
_init
_vaultOnly
vault
pendingVault
delayVault
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MVN - Misleading Variable Naming

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L220,292
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

The contract is currently utilizing the `_vaultOnly` variable in a manner that might lead to confusion regarding its intended purpose. Specifically, in the `Swapout` function, there is a requirement that the `_vaultOnly` variable must be set to `false` to proceed with the function execution. However, the naming of `_vaultOnly` suggests that it is intended to restrict the execution of this function to only the vault address. This discrepancy between the variable's name and its actual use in the logic could lead to misunderstandings about the function's behavior and its security implications.

```
    function setVaultOnly(bool enabled) external onlyVault {
        _vaultOnly = enabled;
    }

    function Swapout(uint256 amount, address bindaddr) public returns
(bool) {
        require(!_vaultOnly, "AnyswapV4ERC20: onlyAuth");
        ...
    }
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code. It is recommended to reassess the intended functionality of the `_vaultOnly` variable. If the purpose is solely to toggle the `Swapout` function's ability to be executed, then renaming

`_vaultOnly` to more accurately reflect this toggle functionality would be beneficial. A name that clearly indicates its role in enabling or disabling the `Swapout` function would prevent confusion and enhance code readability. Alternatively, if the original intent was to ensure that the `Swapout` function can only be invoked by the vault address, then the logic should be revised to align with this intention. This might involve implementing additional checks to verify that the caller is the vault address when `_vaultOnly` is true. Clarifying and aligning the variable name and functionality will improve the contract's transparency and security.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L225,241,252,316,366,474,493,495,534,556,587
Status	Unresolved

Description

The contract is missing error messages. The 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.

```
require(_init)
require(block.timestamp >= delayVault)
require(block.timestamp >= delayMinter)
require(_decimals == IERC20(_underlying).decimals())
require(underlying != address(0x0) && underlying != address(this))
require(verifyEIP712(target, hashStruct, v, r, s) ||
verifyPersonalSign(target, hashStruct, v, r, s))
require(to != address(0) || to != address(this))
```

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.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L221
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
_vaultOnly = enabled;
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L219,236,247,258
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
_vaultOnly = enabled;
pendingVault = _vault;
pendingMinter = _auth;
isMinter[_auth] = false;
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L183
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint public delay = 2*24*3600;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

VDB - Vault Delay Bypass

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L234,240,266
Status	Unresolved

Description

The contract contains the `setVault` and `applyVault` functions to manage the `vault` address, incorporating a delay mechanism for added security. The `setVault` function allows setting a new vault address, stored in `pendingVault`, and establishes a delay period before this change can be applied. The `applyVault` function, which can only be executed after the delay period, finalizes the change by updating the vault address. However, the contract also includes the `changeVault` function that immediately changes the `vault` address without adhering to the delay mechanism. This function directly updates both `vault` and `pendingVault` variables, bypassing the intended security measure of the delay. As a result, the `changeVault` function undermines the delay logic implemented in the other functions.

```
    function setVault(address _vault) external onlyVault {
        require(_vault != address(0), "AnyswapV3ERC20: address(0x0)");
        pendingVault = _vault;
        delayVault = block.timestamp + delay;
    }

    function applyVault() external onlyVault {
        require(block.timestamp >= delayVault);
        vault = pendingVault;
    }

    function changeVault(address newVault) external onlyVault returns
(bool) {
        require(newVault != address(0), "AnyswapV3ERC20: address(0x0)");
        vault = newVault;
        pendingVault = newVault;
        emit LogChangeVault(vault, pendingVault, block.timestamp);
        return true;
    }
```

Recommendation

It is recommended to reconsider the approach to changing the vault address. If the intended functionality is to apply the change only after a specific delay, then the `changeVault` function should be modified to align with this logic. This could involve integrating the delay mechanism into `changeVault` or removing the function altogether if it contradicts the intended security protocol. Otherwise, maintaining a function that can instantly override the delay renders the delay logic redundant and potentially exposes the contract to security risks. Ensuring consistency in the implementation of the delay mechanism across all relevant functions is crucial for maintaining the integrity and security of the contract.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L183
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.

```
uint public delay = 2*24*3600
```

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	AnyswapV6ERC20.sol#L170,224,234,245,258,285,291
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.

```
bytes32 public immutable DOMAIN_SEPARATOR
address _vault
address _auth

function Swapin(bytes32 txhash, address account, uint256 amount) public
onlyAuth returns (bool) {
        _mint(account, amount);
        emit LogSwapin(txhash, account, amount);
        return true;
    }

...
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L140
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.

```
function safeApprove(IERC20 token, address spender, uint value) internal
{
        require((value == 0) || (token.allowance(address(this), spender)
== 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        callOptionalReturn(token,
abi.encodeWithSelector(token.approve.selector, spender, value));
    }
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L226,314,325
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
vault = _vault
underlying = _underlying
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L124,330
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.

```
assembly { codehash := extcodehash(account) }
assembly {chainId := chainid()}
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	AnyswapV6ERC20.sol#L47
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.

```
pragma solidity ^0.8.2;
```

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

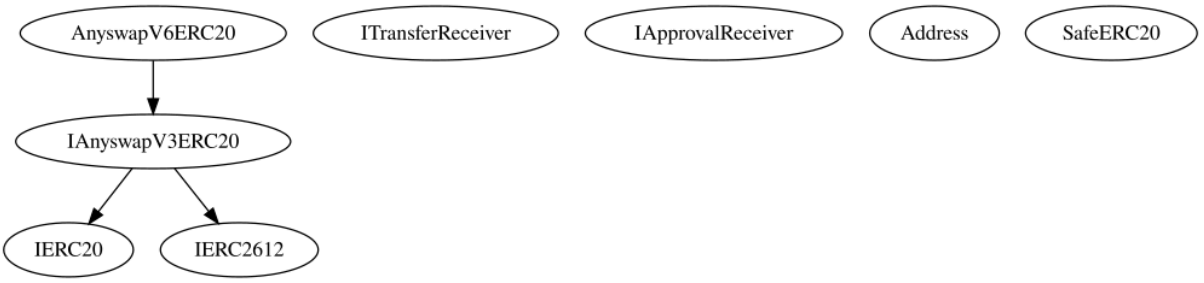
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	totalSupply	External		-
	decimals	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IERC2612	Interface			
	nonces	External		-
	permit	External	✓	-
	transferWithPermit	External	✓	-
IAnyswapV3ERC20	Interface	IERC20, IERC2612		
	approveAndCall	External	✓	-
	transferAndCall	External	✓	-

ITransferReceiver	Interface			
	onTokenTransfer	External	✓	-
IApprovalReceiver	Interface			
	onTokenApproval	External	✓	-
Address	Library			
	isContract	Internal		
SafeERC20	Library			
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
	callOptionalReturn	Private	✓	
AnyswapV6ERC20	Implementation	IAnyswapV3ERC20		
	owner	Public		-
	mpc	Public		-
	setVaultOnly	External	✓	onlyVault
	initVault	External	✓	onlyVault
	setVault	External	✓	onlyVault
	applyVault	External	✓	onlyVault
	setMinter	External	✓	onlyVault

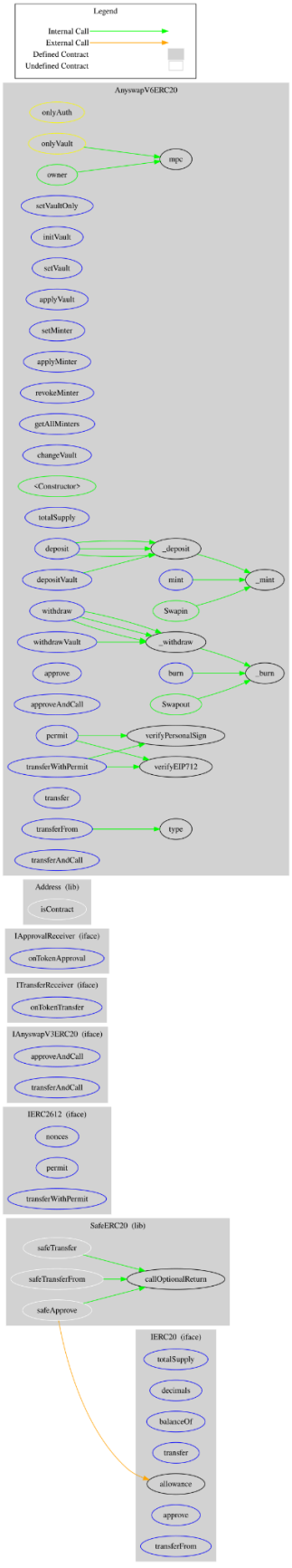
	applyMinter	External	✓	onlyVault
	revokeMinter	External	✓	onlyVault
	getAllMinters	External		-
	changeVault	External	✓	onlyVault
	mint	External	✓	onlyAuth
	burn	External	✓	onlyAuth
	Swapin	Public	✓	onlyAuth
	Swapout	Public	✓	-
		Public	✓	-
	totalSupply	External		-
	deposit	External	✓	-
	deposit	External	✓	-
	deposit	External	✓	-
	depositVault	External	✓	onlyVault
	_deposit	Internal	✓	
	withdraw	External	✓	-
	withdraw	External	✓	-
	withdraw	External	✓	-
	withdrawVault	External	✓	onlyVault
	_withdraw	Internal	✓	
	_mint	Internal	✓	
	_burn	Internal	✓	
	approve	External	✓	-

	approveAndCall	External	✓	-
	permit	External	✓	-
	transferWithPermit	External	✓	-
	verifyEIP712	Internal		
	verifyPersonalSign	Internal		
	transfer	External	✓	-
	transferFrom	External	✓	-
	transferAndCall	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
AnyswapV6ERC20
onlyAuth
onlyVault
owner
mpc
setVaultOnly
initVault
setVault
applyVault
setMinter
applyMinter
revokeMinter
getAllMinters
changeVault
<Constructor>
totalSupply
deposit
_deposit
depositVault
mint
_mint
withdraw
Swapin
withdrawVault
_withdraw
approve
burn
_burn
approveAndCall
Swapout
permit
verifyPersonalSign
transferWithPermit
verifyEIP712
transfer
transferFrom
type
transferAndCall
Address (lib)
isContract
IApprovalReceiver (iface)
onTokenApproval
ITransferReceiver (iface)
onTokenTransfer
IAnyswapV3ERC20 (iface)
approveAndCall
transferAndCall
IERC2612 (iface)
nonces
permit
transferWithPermit
SafeERC20 (lib)
safeTransfer
safeTransferFrom
callOptionalReturn
safeApprove
IERC20 (iface)
totalSupply
decimals
balanceOf
transfer
allowance
approve
transferFrom

Summary

The AnyswapV6ERC20 contract implements advanced token functionalities, integrating cross-chain interoperability and enhanced transfer mechanisms. This audit investigates security issues, business logic concerns and potential improvements.

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