Cyberscope

A TAC Security Company

Audit Report

Unipoly Contracts

October 2025

Repository
https://github.com/Mohammadali-Ghods/UNPChainBridge/tree/main/Contracts

Commit 6c14b33c7aabd6536b3348e91add0bfca405203e

Audited by © cyberscope

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

Repository	https://github.com/Mohammadali-Ghods/UNPChainBridge/tree/main/Contracts
Commit	6c14b33c7aabd6536b3348e91add0bfca405203e

Audit Updates

Initial Audit	01 Oct 2025

Source Files

Filename	SHA256
BridgeUNPSide.sol	6deccb7251e4dfd4a9341075953f98e7d9489f631c3e4022b42e0d62e38855cf
BridgeEthSide.sol	b3ee3fd02042c197140a7fb1e6b0d591d7cdfa43fd478de5b7ee2a15a012e2eb
BNBUNPCoin.sol	5da103fa08f8131f72c1eaaece21107b3ed31e85ce9c4c483354b9206c2d7a79
BNBBridge.sol	1e836807269c39e72e9f6d0a417148201aa1bba941fae40eeb9b580ba7707473

Overview

UNPToken Contract

The `UNPToken` contract implements an ERC20 token with a **maximum supply cap** and a **controlled minting and burning mechanism**.
It ensures strict issuance control by assigning a **single authorized minter/burner address** that can only be set once.
This contract leverages OpenZeppelin's secure `ERC20` and `Ownable` implementations to maintain transparency and safety.

Mint Functionality

The `mint` function allows the authorized minter/burner to create new tokens up to the defined `maxSupply`.
It is designed to facilitate controlled token creation, commonly used in bridge operations or incentive mechanisms.

Burn Functionality

The `burn` function enables the minter/burner address to destroy tokens from a specified address.
This maintains token equilibrium when tokens are bridged or withdrawn on another network.

ERC20Bridge (BNBBridge.sol)

The **mint/burn version of the `ERC20Bridge` contract** is designed for tokens that support **minting and burning**, such as `UNPToken`.
It facilitates the bridging of tokens across different blockchain environments by burning value on one chain and minting it on another.

Deposit Mechanism

When a user deposits tokens, the bridge **burns** the corresponding amount from their balance. This effectively removes tokens from circulation on the source chain while signaling an equivalent mint on the destination chain. An event `Deposited` is emitted to notify off-chain services (such as a relayer or validator).

Withdraw Mechanism

Users can withdraw tokens by providing a **validator-signed message** authorizing the withdrawal. The contract verifies the signature and **mints** tokens to the user. The system uses a **nonce-based protection mechanism** by marking each processed transaction as completed.

ERC20Bridge (BridgeEthSide.sol)

This version of the `ERC20Bridge` contract supports **standard ERC20 tokens** that do not allow minting or burning. Instead, it uses **token custody** within the bridge to lock and release tokens safely across networks.

Deposit Functionality

Users deposit tokens by transferring them to the bridge contract using `transferFrom`. These tokens remain locked within the contract until a corresponding withdrawal request is verified.

Withdrawal Functionality

Withdrawals require a **validator-signed message**.
Once validated, the bridge **transfers** tokens from its balance to the user.

NativeTokenBridge Contract (BridgeUNPSide.sol)

The `NativeTokenBridge` contract enables the secure transfer of **native blockchain tokens** (e.g., ETH, BNB, MATIC) across different networks.
Unlike ERC20 bridges, it handles native currency directly and uses a validator-based authorization system for withdrawals.

Deposit (Receiving Native Tokens)

Users can send native tokens directly to the bridge contract.
The contract accepts incoming transfers through its `receive()` function and records them as part of its balance.

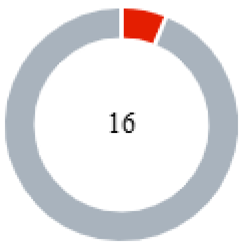
Withdraw Functionality

Withdrawals require a **validator-signed authorization message**.
After validation, the bridge sends the specified amount of native tokens directly to the user.
An event `Withdrawn` is emitted to record successful transactions.

Chart

Contract	Purpose	Token Type	Mechanism	Validator Role
UNPToken	ERC20 token with controlled minting/burning	ERC20	Mint / Burn	N/A
ERC20Bridge (Mint/Burn)	Bridge for mintable tokens (e.g., UNPToken)	Custom ERC20	Burn on deposit / Mint on withdraw	Signature verification
ERC20Bridge (Transfer)	Bridge for standard ERC20 tokens	Standard ERC20	Lock on deposit / Release on withdraw	Signature verification
NativeTokenBridge	Bridge for native blockchain tokens	Native coin	Hold and send	Signature verification

Findings Breakdown



●	Critical	1
●	Medium	0
●	Minor / Informative	15

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	1	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	15	0	0	0

Diagnostics

🔴 Critical 🟠 Medium ⚪ Minor / Informative

Severity	Code	Description	Status
🔴	SRV	Signature Replay Vulnerability	Unresolved
⚪	BT	Burns Tokens	Unresolved
⚪	CCR	Contract Centralization Risk	Unresolved
⚪	ECV	External Call Validation	Unresolved
⚪	IDI	Immutable Declaration Improvement	Unresolved
⚪	MT	Mints Tokens	Unresolved
⚪	MC	Missing Check	Unresolved
⚪	MDSU	Missing Deposit State Update	Unresolved
⚪	MEE	Missing Events Emission	Unresolved
⚪	PTAI	Potential Transfer Amount Inconsistency	Unresolved
⚪	TSI	Tokens Sufficiency Insurance	Unresolved
⚪	ZWAA	Zero Withdraw Amount Allowed	Unresolved
⚪	L04	Conformance to Solidity Naming Conventions	Unresolved
⚪	L06	Missing Events Access Control	Unresolved

●	L17	Usage of Solidity Assembly	Unresolved
●	L19	Stable Compiler Version	Unresolved

SRV - Signature Replay Vulnerability

Criticality	Critical
Location	BridgeUNPSide.sol#L12 BridgeEthSide.sol#L35 BNBBridge.sol#L12
Status	Unresolved

Description

The withdraw function allows users to claim funds by providing a `bytes32 nonce` and a `signature` from the `validator` address. The signature is generated off-chain using the `receiver` address, `amount`, and `nonce`. However, the current implementation does not include the chain id in the signed message. This omission creates a vulnerability where the same signature could be reused across multiple chains where the contracts are deployed, potentially allowing users to withdraw funds multiple times on different chains.

Shell
```
function withdraw(uint256 amount, bytes32 nonce, bytes
memory signature) external {
    require(!processed[nonce], "Already processed");
    bytes32 messageHash = getMessageHash(msg.sender,
amount, nonce);
    require(recoverSigner(messageHash, signature) ==
validator, "Invalid signature");
    ...

}
```

Additionally, if multiple bridges are deployed on the same chain, a user could potentially claim funds from each of them using the same signature and nonce. This vulnerability also extends to scenarios where multiple bridges exist on the same network for different tokens, allowing a user to withdraw funds from each token bridge with a single signature.

Recommendation

To mitigate the cross-chain and multi-bridge signature reuse vulnerabilities, the signature generation should include additional unique data to bind it to a specific context. At a minimum, the chain ID should be incorporated to prevent cross-chain replay attacks, ensuring that a signature valid on one chain cannot be reused on another. Furthermore, each bridge deployment and token type should include a unique identifier in the signed message, so that a signature can only be used for that specific bridge and token combination. Implementing these changes will prevent users from withdrawing funds multiple times across different chains, bridges, or token contracts, strengthening the overall security of the withdrawal mechanism.

BT - Burns Tokens

Criticality	Minor / Informative
Location	BNBUNPCoin.sol#L38
Status	Unresolved

Description

The owner of the contract has the authority to set the `minterBurner` address. The `minterBurner` address has the authority to burn tokens from a specific address. They may take advantage of it by calling the `burn` function. As a result, the targeted address will lose the corresponding tokens.

```
Shell
function burn(address from, uint256 amount)
external onlyMinterBurner {
    _burn(from, amount);

}
```

Recommendation

The team should carefully manage the private keys of the owner’s and `minterBurner` address's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, but this is a non-reversible action.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	BridgeUNPSide.sol#L15 BridgeEthSide.sol#L30 BNBUNPCoin.sol#L21,32,37 BNBBridge.sol#L25
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.

Shell
```
function setMinterBurner(address _addr) external
onlyOwner
function mint(address to, uint256 amount) external
onlyMinterBurner

function burn(address from, uint256 amount)

external onlyMinterBurner
```

Additionally, the contracts' `withdraw` operations highly depend on the validator's action. If the validator acts maliciously or their keys get compromised, users will not be able to receive their tokens or the pool will be depleted of its funds.

Shell
```
function withdraw(uint256 amount, bytes32 nonce,
bytes memory signature) external {
    ...
    require(recoverSigner(messageHash, signature)
== validator, "Invalid signature");
    ...
}
```

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.

ECV - External Call Validation

Criticality	Minor / Informative
Location	BridgeUNPSide.sol#L17
Status	Unresolved

Description

The `NativeTokenBridge` contract sends funds to the recipient via an external call during the `withdraw` operation. The recipient can be either an externally owned account or a smart contract. If the recipient is a contract, the external call may trigger its external functions, potentially disrupting the bridge's operation or leading to unintended interactions with third-party contracts.

Shell
```
(bool sent, ) = msg.sender.call{value: amount}("");
```

Recommendation

The team could limit the recipient's ability to execute complex code during the transfer by restricting the amount of available `gas` that can be used during the external call or by using the `transfer` or `send` methods. In addition the team is advised to consider reentrancy guard measures to prevent unintended interactions and improve the overall security and reliability of the bridge.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	BridgeUNPSide.sol#L9 BridgeEthSide.sol#L17 BNBBridge.sol#L17
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.

```
Shell
validator
```

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.

MT - Mints Tokens

Criticality	Minor / Informative
Location	BNBUNPCoin.sol#L32
Status	Unresolved

Description

The owner of the contract has the authority to set the `minterBurner` address. The `minterBurner` address has the authority to mint tokens. They may take advantage of it by calling the `mint` function. As a result, the contract tokens will be highly inflated.

```
Shell
function mint(address to, uint256 amount) external
onlyMinterBurner {
    require(totalSupply() + amount <= maxSupply,
"Max supply exceeded");
    _mint(to, amount);

}
```

Recommendation

The team should carefully manage the private keys of the owner's and `minterBurner` address's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership but this is a non-reversible action.

MC - Missing Check

Criticality	Minor / Informative
Location	BNBUNPCoin.sol#L18
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, `maxSupply` should have a reasonable value that accounts for the contract's decimals.

```
Shell
maxSupply = maxSupply_;
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MDSU - Missing Deposit State Update

Criticality	Minor / Informative
Location	BridgeUNPSide.sol#L40 BridgeEthSide.sol#L19 BNBBridge.sol#L19
Status	Unresolved

Description

The `deposit` functions in the contracts do not update the contracts' state when called. This is because the contracts lack state variables or mappings to record and track user deposits.

```
Shell
receive() external payable {}
```

```
Shell
function deposit(uint256 amount) external {
    require(amount > 0, "Invalid amount");
    require(token.transferFrom(msg.sender,
address(this), amount), "Transfer failed");
    emit Deposited(msg.sender, amount);

}
```

Shell
```
function deposit(uint256 amount) external {
    require(amount > 0, "Invalid amount");
    token.burn(msg.sender, amount);
    emit Deposited(msg.sender, amount);

}
```

Recommendation

Tracking the deposits of users would make the contracts more scalable and transparent. The team is advised to update the contracts so that they keep the state of the deposits of users.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	BNBUNPCoin.sol#L21 BridgeUNPSide.sol#L40
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.

Shell

```
function setMinterBurner(address _addr) external
onlyOwner
```

Additionally, the `receive` function of the `NativeTokenBridge` should also emit an event to ensure that external applications are updated for potential deposits.

Shell
```
receive() external payable {}
```

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.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	BridgeEthSide.sol#L22
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

Shell
```
function deposit(uint256 amount) external {
    require(amount > 0, "Invalid amount");
    require(token.transferFrom(msg.sender,
address(this), amount),"Transfer failed");
    emit Deposited(msg.sender, amount);
}
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
 Actual Transferred Amount = Balance After Transfer - Balance
Before Transfer
```

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	BridgeUNPSide.sol#L12 BridgeEthSide.sol#L27
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external source. While external sources can provide flexibility, it introduces a dependency on the sources' actions, which can lead to various issues and centralization risks.

Shell
```
function withdraw(uint256 amount, bytes32 nonce,
bytes memory signature) external {
     ...
     require(token.transfer(msg.sender, amount),
"Transfer failed");

}
```

Shell
```
function withdraw(uint256 amount, bytes32 nonce,
bytes memory signature) external {
     ...
     (bool sent, ) = msg.sender.call{value:
amount}("");
     require(sent, "Native transfer failed");
```

```
  }
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

ZWAA - Zero Withdraw Amount Allowed

Criticality	Minor / Informative
Location	BridgeUNPSide.sol#L12 BridgeEthSide.sol#L27 BNBBridge.sol#L25
Status	Unresolved

Description

The contracts' `withdraw` function allows users to withdraw an amount of tokens or currency from the contract by providing a `nonce` and a `signature` that matches the validator's address. However the withdraw function allows the transfer of zero amounts of tokens or currency.

Shell
```
function withdraw(uint256 amount, bytes32 nonce,
bytes memory signature) external
```

Recommendation

The team is advised to only allow the success of meaningful withdrawals in order to optimize the contracts' usefulness and efficiency.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	BNBUNPCoin.sol#L21
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
```
address _addr
```

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.

L06 - Missing Events Access Control

Criticality	Minor / Informative
Location	BNBUNPCoin.sol#L24
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task. There are functions that have no event emitted, so it is difficult to track off-chain changes.

Shell
```
minterBurner = _addr
```

Recommendation

To avoid this issue, it's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	BridgeUNPSide.sol#L33 BridgeEthSide.sol#L47 BNBBridge.sol#L45
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 {
r := mload(add(sig, 32))
s := mload(add(sig, 64))
v := byte(0, mload(add(sig, 96)))
}
```

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	BridgeUNPSide.sol#L2 BridgeEthSide.sol#L2 BNBBridge.sol#L2
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 ^0.8.18;
```

Recommendation

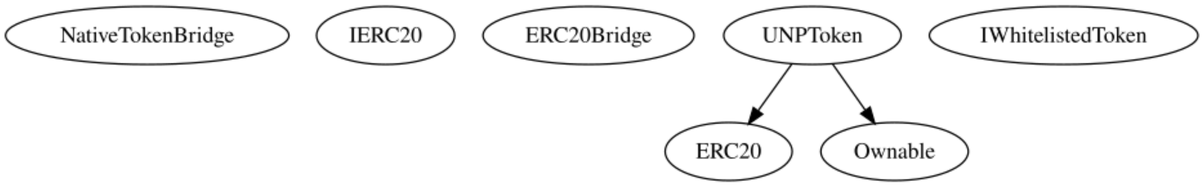
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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

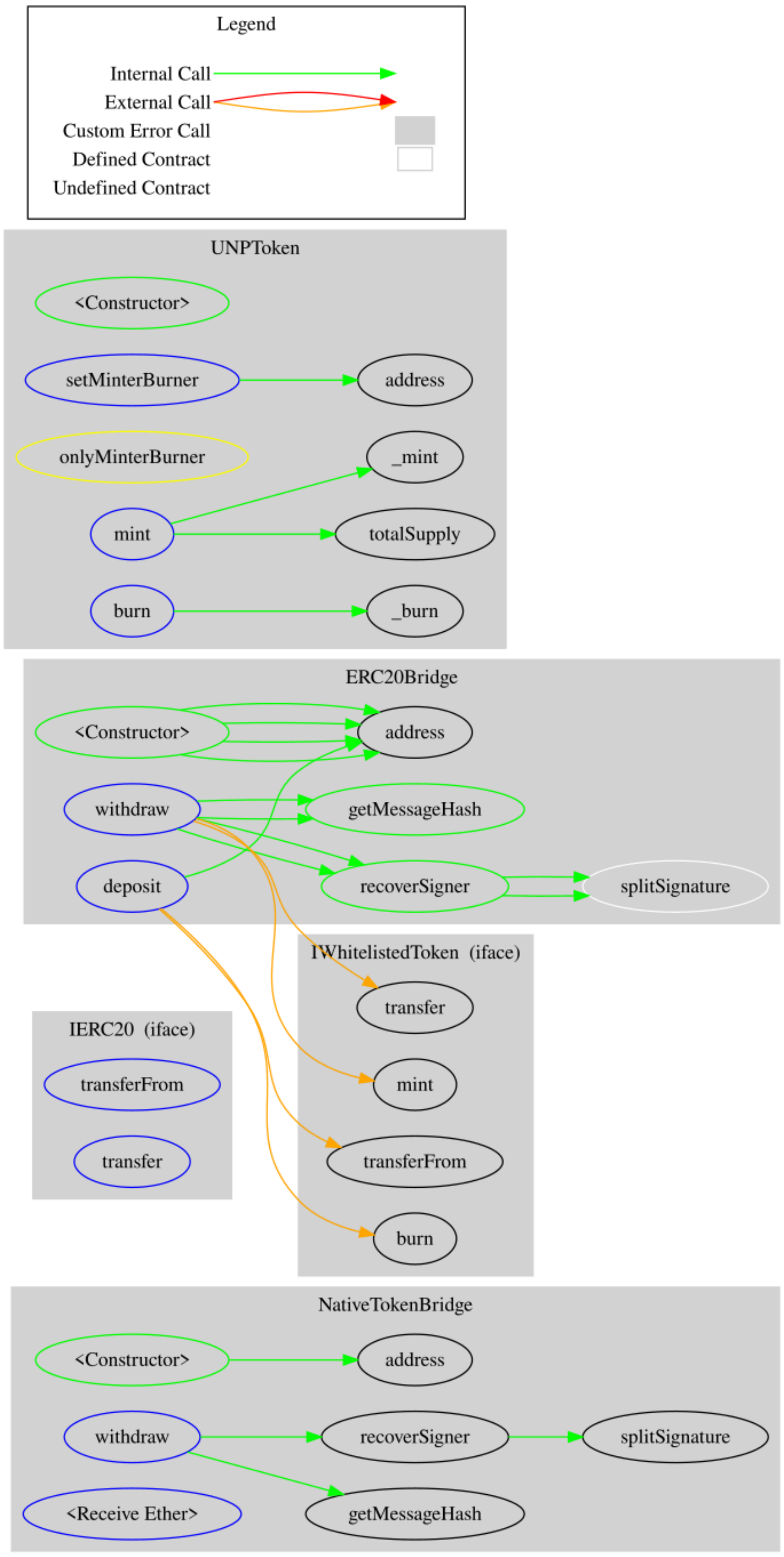
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
NativeTokenBridge	Implementation			
		Public	✓	-
	withdraw	External	✓	-
	getMessageHash	Public		-
	recoverSigner	Public		-
	splitSignature	Internal		
		External	Payable	-
IERC20	Interface			
	transferFrom	External	✓	-
	transfer	External	✓	-
ERC20Bridge	Implementation			
		Public	✓	-
	deposit	External	✓	-
	withdraw	External	✓	-
	getMessageHash	Public		-
	recoverSigner	Public		-
	splitSignature	Internal		

UNPToken	Implementation	ERC20, Ownable		
		Public	✓	ERC20 Ownable
	setMinterBurner	External	✓	onlyOwner
	mint	External	✓	onlyMinterBurner
	burn	External	✓	onlyMinterBurner
IWhitelistedToken	Interface			
	burn	External	✓	-
	mint	External	✓	-
ERC20Bridge	Implementation			
		Public	✓	-
	deposit	External	✓	-
	withdraw	External	✓	-
	getMessageHash	Public		-
	recoverSigner	Public		-
	splitSignature	Internal		

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
UNPToken
<Constructor>
setMinterBurner
address
onlyMinterBurner
_mint
mint
totalSupply
burn
_burn
ERC20Bridge
<Constructor>
address
withdraw
getMessageHash
deposit
recoverSigner
splitSignature
IWhitelistedToken (iface)
transfer
mint
transferFrom
burn
IERC20 (iface)
transferFrom
transfer
NativeTokenBridge
<Constructor>
address
withdraw
recoverSigner
splitSignature
<Receive Ether>
getMessageHash

Summary

Unipoly contract implements a token, bridge and utility mechanism. 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 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