Bitcoin.com

Verse

Dynamic Rewards Farming

SMART CONTRACT AUDIT

22.10.2024

Made in Germany by Softstack.io

Table of contents

1. Disclaimer

The audit makes no statements or warrantees about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Saint Bitts LLC. If you are not the intended receptor of this document, remember that any disclosure, copying or dissemination of it is forbidden.

Major Versions / Date	Description
0.1 (10.10.2024)	Layout
0.4 (14.10.2024)	Automated Security Testing Manual Security Testing
0.5 (14.10.2024)	Verify Claims
0.6 (14.10.2024)	Testing SWC Checks
0.9 (14.10.2024)	Summary and Recommendation
1.0 (14.10.2024)	Final document
1.1 (22.10.2024)	Re-check

2. About the Project and Company

Company address:

Saint Bitts LLC
858 Zenway Blvd, Unit 15-203
C/O Corporate Solutions Ltd
Bayview Commercial Complex Nevis
Frigate Bay, Saint Kitts
Saint Kitts & Nevis

Website: https://www.bitcoin.com

LinkedIn: https://www.linkedin.com/company/bitcoin.com

Twitter (X): https://twitter.com/bitcoincom

Discord: https://discord.gg/qfEUDMP79w

Telegram: https://t.me/GetVerse/1

Instagram: https://www.instagram.com/bitcoin.com_official

Youtube: https://www.youtube.com/channel/UCetxkZolEBHX47BqtZktbkg/

Facebook: https://www.facebook.com/buy.bitcoin.news

TikTok: https://www.tiktok.com/@bitcoin.com_news

2.1 Project Overview

Bitcoin.com Verse is an ecosystem developed by Bitcoin.com to offer users a broad range of crypto services and financial products. At its center is the VERSE token, which rewards users for engaging in platform activities and provides access to exclusive benefits. The platform introduces decentralized finance (DeFi) opportunities, creating a seamless experience for both new and experienced cryptocurrency users.

The VERSE token, an ERC-20 utility and reward token on the Ethereum blockchain, powers the Bitcoin.com Verse platform. It is used for earning rewards, staking, accessing DeFi services, and unlocking lower trading fees on the Bitcoin.com exchange. The Bitcoin.com Wallet integrates VERSE tokens alongside other cryptocurrencies, offering a secure way to store, send, and receive crypto assets. The ecosystem encourages user participation through VERSE Rewards, where users can earn tokens for trading, staking, and more. As part of its development, Bitcoin.com distribute VERSE tokens through airdrops and sales, ensuring wide community participation. The platform will continue to expand its DeFi services, allowing users to stake and earn yield while also benefiting from lower trading fees and VIP access to certain features.

In conclusion, Bitcoin.com Verse aims to integrate decentralized finance into its existing crypto services. The VERSE token plays a key role in incentivizing user engagement and expanding the platform’s features. As the ecosystem grows, it aims to make DeFi more accessible and valuable to a global audience.

3. Vulnerability & Risk Level

Risk represents the probability that a certain source-threat will exploit vulnerability, and the impact of that event on the organization or system. Risk Level is computed based on CVSS version 3.0.

Level	Value	Vulnerability	Risk (Required Action)
Critical	9 – 10	A vulnerability that can disrupt the contract functioning in a number of scenarios, or creates a risk that the contract may be broken.	Immediate action to reduce risk level.
High	7 – 8.9	A vulnerability that affects the desired outcome when using a contract, or provides the opportunity to use a contract in an unintended way.	Implementation of corrective actions as soon as possible.
Medium	4 – 6.9	A vulnerability that could affect the desired outcome of executing the contract in a specific scenario.	Implementation of corrective actions in a certain period.
Low	2 – 3.9	A vulnerability that does not have a significant impact on possible scenarios for the use of the contract and is probably subjective.	Implementation of certain corrective actions or accepting the risk.
Informational	0 – 1.9	A vulnerability that have informational character but is not effecting any of the code.	An observation that does not determine a level of risk

4. Auditing Strategy and Techniques Applied

Throughout the review process, care was taken to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices. To do so, reviewed line-by-line by our team of expert pentesters and smart contract developers, documenting any issues as there were discovered.

4.1 Methodology

The auditing process follows a routine series of steps:

1. Code review that includes the following:
   1. Review of the specifications, sources, and instructions provided to softstack to make sure we understand the size, scope, and functionality of the smart contract.
   2. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   3. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to softstack describe.
2. Testing and automated analysis that includes the following:
   1. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   2. Symbolic execution, which is analysing a program to determine what inputs causes each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, actionable recommendations to help you take steps to secure your smart contracts.

5. Metrics

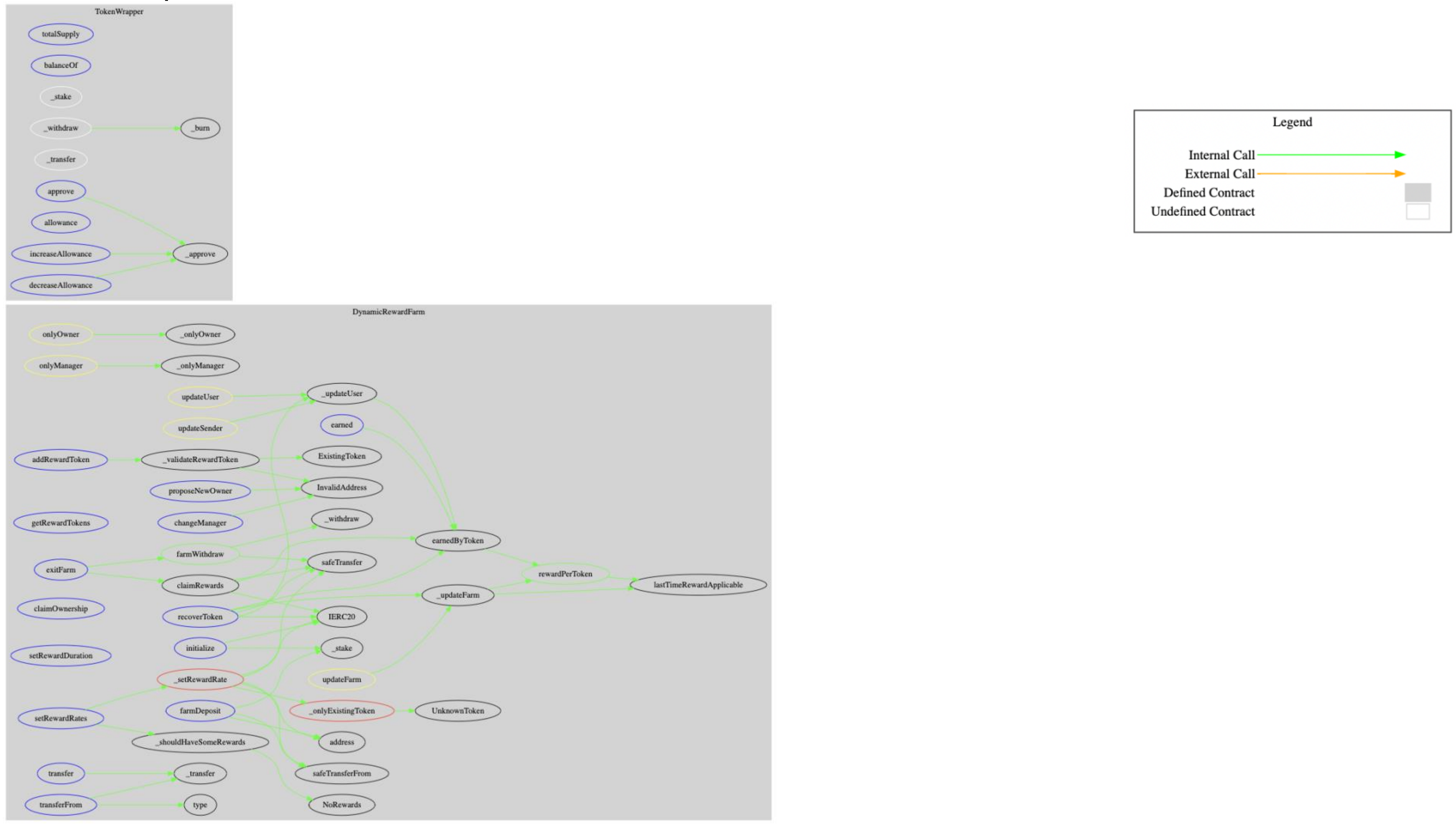
The metrics section should give the reader an overview on the size, quality, flows and capabilities of the codebase, without the knowledge to understand the actual code.

5.1 Tested Contract Files

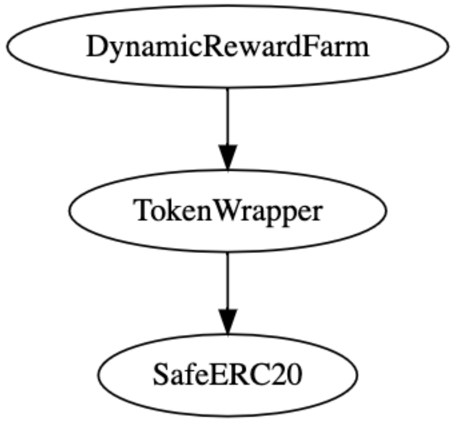
The following are the MD5 hashes of the reviewed files. A file with a different MD5 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different MD5 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

File	Fingerprint (MD5)
./contracts/TokenWrapper.sol	215bcb4176a7a4bb26205384519f268e
./contracts/DynamicRewardFarm.sol	89d299549767dae05558b020a2183246

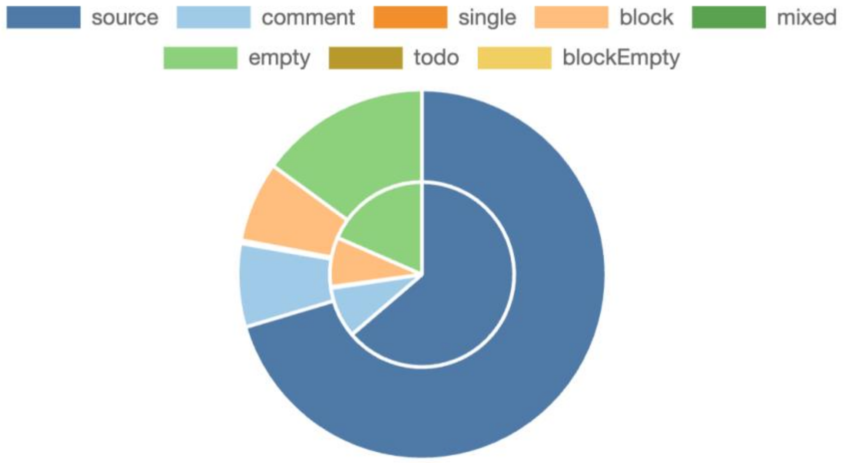
5.2 CallGraph

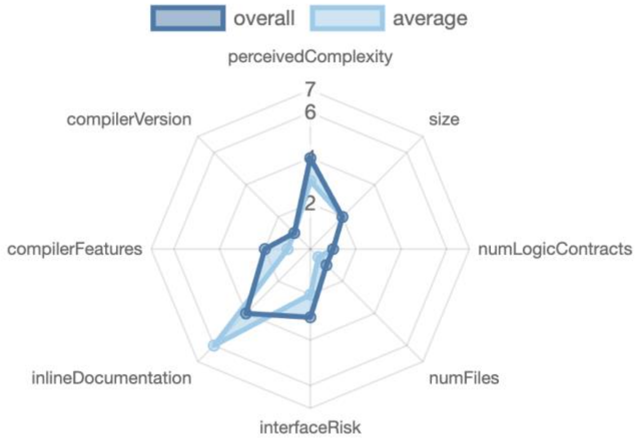


5.3 Inheritance Graph



5.4 Source Lines & Risk





5.5 Capabilities

Solidity Versions observed	🧪 Experimental Features	💰 Can Receive Funds	🖥 Uses Assembly	💣 Has Destroyable Contracts
=0.8.26				

📤 Transfers ETH	⚡ Low-Level Calls	👥 DelegateCall	🧮 Uses Hash Functions	🔖 ECRecover	🌀 New/Create/Create2

Exposed Functions

This section lists functions that are explicitly declared public or payable. Please note that getter methods for public stateVars are not included.

🌐Public	💰Payable
25	0

External	Internal	Private	Pure	View
20	30	8	1	12

StateVariables

Total	🌐Public
20	13

5.6 Source Unites in Scope

File	Logic Contracts	Interfaces	Lines	nLines	nSLOC	Comment Lines	Complex. Score
contracts/DynamicRewardFarm.sol	1		725	584	418	45	227
contracts/TokenWrapper.sol	1		277	215	141	34	32
Totals	**2**		**1002**	**799**	**559**	**79**	**259**

- **Lines**: total lines of the source unit
- **nLines**: normalized lines of the source unit (e.g. normalizes functions spanning multiple lines)
- **nSLOC**: normalized source lines of code (only source-code lines; no comments, no blank lines)
- **Comment Lines**: lines containing single or block comments
- **Complexity Score**: a custom complexity score derived from code statements that are known to introduce code complexity (branches, loops, calls, external interfaces, ...

6. Scope of Work

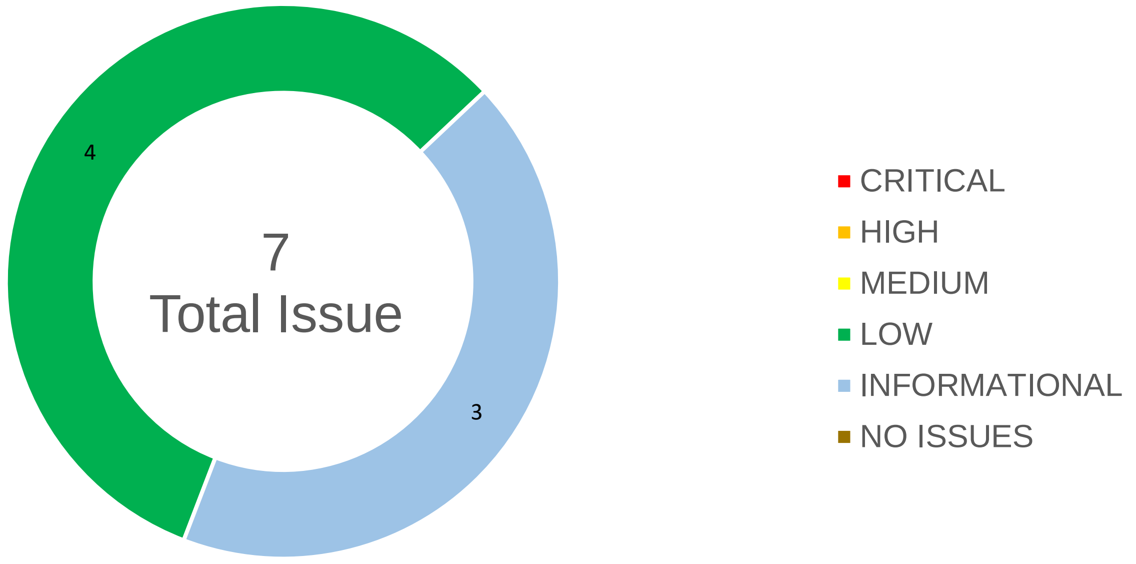
The Bitcoin.com Team provided us with the files that needs to be tested. The scope of the audit are the Dynamic Reward Farm contracts.

The team put forward the following assumptions regarding the security, usage of the contracts:

1. **Compliance with Smart Contract Best Practices**: The audit will ensure that the contracts adhere to established smart contract best practices, including identifying vulnerabilities such as reentrancy, overflow, and underflow issues. The goal is to confirm the contracts are robust against these common threats.
2. **Access Control and Permissions**: The contracts should incorporate strict access control mechanisms to ensure only authorized entities can access or modify sensitive functions. This will prevent unauthorized actions that could compromise the system.
3. **Gas Efficiency**: The contracts will be reviewed for optimization in terms of gas usage. Efficient gas usage will minimize unnecessary computational costs and storage operations, contributing to lower transaction fees for users.
4. **Data Integrity and State Consistency**: The audit will ensure that data integrity and consistency are maintained across all operations. All state changes should accurately reflect the intended actions, and data loss or corruption should be prevented.
5. **Security of Reward Distribution**: The contracts should ensure that reward calculations and distributions are handled accurately and fairly. This includes verifying that rewards are allocated correctly based on staking activity, without any potential for exploitation or manipulation of the reward system.

The main goal of this audit will be to verify these claims and ensure that the contracts are secure, efficient, and reliable. Upon the client’s request, the audit team can provide further feedback on specific areas of the contract.

6.1 Findings Overview



No	Title	Severity	Status
6.2.1	Missing SafeApprove Pattern in the approve Function	LOW	ACKNOWLEDGED
6.2.2	Insufficient Validation of Reward Token Addresses	LOW	ACKNOWLEDGED
6.2.3	Inflexible Reward Token Management Due to Fixed Token Limit	LOW	ACKNOWLEDGED
6.2.4	Insufficient Token Validation in setRewardRates Function	LOW	FIXED
6.2.5	Duplicate Token Validation Logic in DynamicRewardFarm Contract	INFORMATIONAL	ACKNOWLEDGED
6.2.6	Incorrect Import of SafeERC20 in TokenWrapper Contract	INFORMATIONAL	ACKNOWLEDGED

6.2.7	Missing Event Emission for New Reward Token Addition	INFORMATIONAL	ACKNOWLEDGED

6.2 Manual and Automated Vulnerability Test

CRITICAL ISSUES

During the audit, softstack‘s experts found **no Critical issues** in the code of the smart contract.

HIGH ISSUES

During the audit, softstack‘s experts found **no High issues** in the code of the smart contract.

MEDIUM ISSUES

During the audit, softstack‘s experts found no Medium issues in the code of the smart contract.

LOW ISSUES

During the audit, softstack‘s experts found **4 Low issues** in the code of the smart contract

6.2.1 Missing SafeApprove Pattern in the approve Function

Severity: LOW
Status: ACKNOWLEDGED
File(s) affected: TokenWrapper.sol

Attack / Description	While the increaseAllowance and decreaseAllowance functions are already implemented in the contract, the approve function lacks the necessary validation to prevent potential race conditions or re-entrancy issues. The issue here is that the approve function should include a check to ensure the current allowance is either 0 or being set to 0 before updating it. This pattern, known as

	SafeApprove, prevents situations where an attacker could exploit the allowance system by submitting multiple transactions.

Code	Line 191 – 205 (TokenWrapper.sol):
	
	`function approve(`
	`        address _spender,`
	`        uint256 _amount`
	`    )`
	`        external`
	`        returns (bool)`
	`    {`
	`        _approve(`
	`            msg.sender,`
	`            _spender,`
	`            _amount`
	`        );`
	
	`        return true;`
	`    }`

Result/Recommendation	To mitigate this issue, the approve function should include the following validation: require(_amount == 0 || _allowances[msg.sender][_spender] == 0, "SafeApprove: Current allowance must be 0"); This ensures that the allowance can only be updated if: 1. The current allowance is set to 0, or 2. The new allowance being set is 0. Update the approve function to include the validation check:

	function approve(address _spender, uint256 _amount) external returns (bool) {
	    require(_amount == 0 || _allowances[msg.sender][_spender] == 0, "SafeApprove: Current allowance must be 0");
	    _approve(msg.sender, _spender, _amount);
	    return true;
	}
	
	Impact:
	• Prevents Front-Running: This change will protect against potential front-running attacks by ensuring the current allowance is fully consumed or set to 0 before updating it.
	• Ensures Proper Allowance Management: By enforcing this pattern, you prevent incorrect or insecure usage of allowances, improving the robustness of the contract.

6.2.2 Insufficient Validation of Reward Token Addresses

Severity: LOW
Status: ACKNOWLEDGED
File(s) affected: DynamicRewardFarm.sol

Attack / Description	The _validateRewardToken function only checks if the token address is non-zero and not already added. It doesn't verify if the address actually represents a valid ERC20 token contract. Even without malicious intent, adding a non-ERC20 contract by mistake could cause the farm to malfunction, potentially locking user funds or causing incorrect reward distributions.

Code	Line 139 - 154 (DynamicRewardFarm.sol):
	
	function _validateRewardToken(
	        address _tokenAddress

	```
	)
	    private
	    view
	{
	    if (_tokenAddress == ZERO_ADDRESS) {
	        revert InvalidAddress();
	    }

	    for (uint256 i; i < rewardTokens.length; i++) {
	        if (_tokenAddress == rewardTokens[i]) {
	            revert ExistingToken();
	        }
	    }
	}
	```

Result/Recommendation	Implement a more comprehensive validation process.
	```
	function _validateRewardToken(address _tokenAddress) private view {
	   if (_tokenAddress == ZERO_ADDRESS) {
	       revert InvalidAddress();
	   }
	   if (isRewardToken[_tokenAddress]) {
	       revert ExistingToken();
	}
	   // Basic ERC20 interface check
	   try IERC20(_tokenAddress).totalSupply() returns (uint256) {
	       // Additional checks could be added here
	   } catch {
	       revert InvalidERC20Token();
	} }
	```
	This change adds a basic check to ensure the address implements the ERC20 totalSupply

	function. While not foolproof, it provides an additional layer of security against non-ERC20 addresses being added as reward tokens.

6.2.3 Inflexible Reward Token Management Due to Fixed Token Limit

Severity: LOW
Status: ACKNOWLEDGED
File(s) affected: DynamicRewardFarm.sol
Update: No further action is required as this is a wished design choice and not present a vulnerability in this case.

Attack / Description	The DynamicRewardFarm contract has a hardcoded limit of 10 reward tokens (MAX_TOKENS = 10). Once this limit is reached, no new reward tokens can be added, and there is no mechanism to remove existing tokens. This design significantly limits the contract's flexibility to adapt to changing market conditions or project requirements. **Impact:** 1. Limited Scalability: The contract cannot support more than 10 reward tokens, which could hinder the project's growth and its ability to offer a diverse range of rewards. 2. Permanent Token Lock-In: Once a reward token is added, it cannot be removed, even if it becomes obsolete or undesirable. 3. Forced Contract Redeployment: To add new tokens beyond the limit or remove existing ones, a new contract would need to be deployed. This leads to: ○ Increased gas costs and operational complexity ○ Potential loss of user trust during the migration process ○ Risk of fragmenting the user base across multiple contracts 4. Missed Opportunities: The project may miss out on new token partnerships or innovative reward strategies once the 10-token limit is reached.

Code	Line 20 (DynamicRewardFarm.sol):
	
	`uint256 constant MAX_TOKENS = 10;`
	
	Line 114 - 134 (DynamicRewardFarm.sol):
	`/**`
	`     * @dev Adds a new reward token to the farm`
	`     */`
	`    function addRewardToken(`
	`        address _rewardToken`
	`    )`
	`        external`
	`        onlyOwner`
	`    {`
	`        require(`
	`            tokenCount < MAX_TOKENS,`
	`            "DynamicRewardFarm: MAX_TOKENS_REACHED"`
	`        );`
	
	`        _validateRewardToken(`
	`            _rewardToken`
	`        );`
	
	`        rewardTokens.push(`
	`            _rewardToken`
	`        );`
	
	`        tokenCount = tokenCount + 1;`
	`    }`

Result/Recommendation	Recommendations:

	1. Dynamic Token Limit: Replace the hardcoded MAX_TOKENS with a variable that can be adjusted by the contract owner or through a governance mechanism. 2. Implement Token Removal: Add a function to safely remove reward tokens, ensuring proper handling of accrued rewards and user balances. 3. Token Rotation Mechanism: Implement a system to rotate or replace existing tokens with new ones, maintaining the 10-token limit but allowing for flexibility. 4. Upgradeable Contract Design: Consider implementing the contract as upgradeable, allowing for future modifications without the need for full redeployment. 5. Multi-Contract Architecture: Design a system where multiple farm contracts can be managed under a single interface. This would allow for the creation of new farms when needed, without being constrained by the limitations of a single contract.

6.2.4 Insufficient Token Validation in setRewardRates Function

Severity: LOW
Status: FIXED
File(s) affected: DynamicRewardFarm.sol
Update: https://github.com/bitcoin-portal/sol-farms/commit/e5c23237c65d58eb33e14bc984cf99c810a8d1da

Attack / Description	The setRewardRates function in the DynamicRewardFarm contract lacks proper validation of the reward tokens passed as parameters. While the function checks for array length mismatches, it does not verify if the tokens in the _rewardTokens array are actually the reward tokens stored in the contract. This oversight could allow an attacker with manager privileges to set reward rates for non-existent or incorrect tokens.

Code	Line 571 - 602 (DynamicRewardFarm.sol):
	
	`function setRewardRates(`
	`        address[] calldata _rewardTokens,`
	`        uint256[] calldata _newRewardRates`

	```
	)
	    external
	    onlyManager
	    updateFarm()
	{
	    require(
	        _rewardTokens.length == _newRewardRates.length,
	        "DynamicRewardFarm: ARRAY_LENGTH_MISMATCH"
	    );

	    require(
	        _rewardTokens.length == rewardTokens.length,
	        "DynamicRewardFarm: TOKEN_LENGTH_MISMATCH"
	    );

	    _shouldHaveSomeRewards(
	        _newRewardRates
	    );

	    for (uint256 i; i < _rewardTokens.length; i++) {
	        _setRewardRate(
	            _rewardTokens[i],
	            _newRewardRates[i]
	        );
	    }

	    lastUpdateTime = block.timestamp;
	    periodFinished = block.timestamp + rewardDuration;
	}
	```

Result/Recommendation	Implement strict validation of the _rewardTokens array against the stored rewardTokens:
	
	function setRewardRates(
	  address[] calldata _rewardTokens,
	  uint256[] calldata _newRewardRates
	)
	  updateFarm()
	{
	  require(
	    _rewardTokens.length == _newRewardRates.length,
	    "DynamicRewardFarm: ARRAY_LENGTH_MISMATCH"
	);
	  require(
	    _rewardTokens.length == rewardTokens.length,
	    "DynamicRewardFarm: TOKEN_LENGTH_MISMATCH"
	);
	external
	onlyManager
	_shouldHaveSomeRewards(_newRewardRates);
	  for (uint256 i; i < _rewardTokens.length; i++) {
	    require(
	      _rewardTokens[i] == rewardTokens[i],
	      "DynamicRewardFarm: INVALID_TOKEN_ORDER"
	);
	    _setRewardRate(_rewardTokens[i], _newRewardRates[i]);
	  }
	  lastUpdateTime = block.timestamp;
	  periodFinished = block.timestamp + rewardDuration;
	}

INFORMATIONAL ISSUES

During the audit, softstack‘s experts found **3 Informational issue** in the code of the smart contract.

6.2.5 Duplicate Token Validation Logic in DynamicRewardFarm Contract

Severity: INFORMATIONAL
Status: ACKNOWLEDGED
File(s) affected: DynamicRewardFarm.sol

Attack / Description	The DynamicRewardFarm contract contains two separate functions, _validateRewardToken and _onlyExistingToken, which perform similar token validation checks. This duplication of functionality creates unnecessary code redundancy and increases the risk of inconsistencies if one function is updated without the other.

Code	Line 139 - 154 (DynamicRewardFarm.sol):
	```
	function _validateRewardToken(
	        address _tokenAddress
	    )
	        private
	        view
	    {
	        if (_tokenAddress == ZERO_ADDRESS) {
	            revert InvalidAddress();
	        }

	        for (uint256 i; i < rewardTokens.length; i++) {
	            if (_tokenAddress == rewardTokens[i]) {
	                revert ExistingToken();
	```

	```
	            }
	        }
	    }
	```
	Line 493 - 506 (DynamicRewardFarm.sol):
	```
	function _onlyExistingToken(
	        address _tokenAddress
	    )
	        private
	        view
	    {
	        for (uint256 i; i < rewardTokens.length; i++) {
	            if (_tokenAddress == rewardTokens[i]) {
	                return;
	            }
	        }

	        revert UnknownToken();
	    }
	```

Result/Recommendation	The token validation logic can be simplified and made more flexible by consolidating it into a single function, as shown below:
	```
	function _validateRewardToken(address _tokenAddress, bool _shouldExist)
	    private
	    view
	{
	    if (_tokenAddress == ZERO_ADDRESS) {
	        revert InvalidAddress();
	    }
	```

```
    bool exists = false;

    for (uint256 i = 0; i < rewardTokens.length; i++) {
        if (_tokenAddress == rewardTokens[i]) {
            exists = true;
            break;
        }
    }

    if (_shouldExist && !exists) {
        revert UnknownToken();
    }

    if (!_shouldExist && exists) {
        revert ExistingToken();
    }
}
```

Update the Calling Functions to Use the New Consolidated Function:

For adding new tokens:

```
_validateRewardToken(_tokenAddress, false);
```

For checking existing tokens:

```
_validateRewardToken(_tokenAddress, true);
```

This approach reduces redundancy in the code by combining the token validation logic into a single function that can handle both adding and checking tokens. Depending on the scenario, the bool

	_shouldExist parameter determines whether the token is expected to exist or not, making the function more versatile and efficient.

6.2.6 Incorrect Import of SafeERC20 in TokenWrapper Contract

Severity: INFORMATIONAL
Status: ACKNOWLEDGED
File(s) affected: TokenWrapper.sol

Attack / Description	The TokenWrapper contract currently imports a non-standard version of SafeERC20.sol. It should import the well-established and widely-used version provided by OpenZeppelin for security and best practices in handling ERC-20 tokens.

Code	Line 5 (TokenWrapper.sol):
	
	pragma solidity =0.8.26;
	
	import "./SafeERC20.sol";
	
	contract TokenWrapper is SafeERC20 {

Result/Recommendation	Replace the current import statement with: import "@openzeppelin/contracts/token/ERC20/utils/SafeERC20.sol"; This will ensure that the contract is using the trusted OpenZeppelin implementation, which includes security enhancements and thorough testing. **Impact:**

	• Security: OpenZeppelin’s implementation is widely trusted and has undergone rigorous security audits, which reduces the risk of vulnerabilities. • Best Practices: Using a standard library ensures compatibility with other contracts and systems in the Ethereum ecosystem. • Community Trust: Developers and auditors will have greater confidence in the contract, knowing it relies on the battle-tested OpenZeppelin libraries. By updating the import to OpenZeppelin's library, the contract will adhere to industry best practices and improve its overall security.

6.2.7 Missing Event Emission for New Reward Token Addition

Severity: INFORMATIONAL
Status: ACKNOWLEDGED
File(s) affected: DynamicRewardFarm.sol

Attack / Description	The addRewardToken function in the DynamicRewardFarm contract adds a new reward token to the farm but fails to emit an event to signal this significant state change. This omission makes it challenging for off-chain systems and users to track when new reward tokens are added to the farm. This function modifies the contract state by adding a new token, but provides no mechanism for external systems to easily detect this change.

Code	Line 114 - 134 (DynamicRewardFarm.sol):
	
	`function addRewardToken(`
	`        address _rewardToken`
	`    )`
	`        external`
	`        onlyOwner`
	`    {`

	```
	        require(
	            tokenCount < MAX_TOKENS,
	            "DynamicRewardFarm: MAX_TOKENS_REACHED"
	        );

	        _validateRewardToken(
	            _rewardToken
	        );

	        rewardTokens.push(
	            _rewardToken
	        );

	        tokenCount = tokenCount + 1;
	    }
	```

Result/Recommendation	Implement an event emission when a new reward token is added. This can be achieved by adding an event declaration and emitting it within the addRewardToken function:
	```
	/ Add this event declaration to the contract
	event RewardTokenAdded(address indexed token, uint256 tokenCount);
	function addRewardToken(address _rewardToken) external onlyOwner {
	   require(tokenCount < MAX_TOKENS, "DynamicRewardFarm:
	MAX_TOKENS_REACHED");
	   _validateRewardToken(_rewardToken);
	   rewardTokens.push(_rewardToken);
	   tokenCount = tokenCount + 1;
	   // Emit the event
	   emit RewardTokenAdded(_rewardToken, tokenCount);
	}
	```

6.3 SWC Attacks

ID	Title	Relationships	Test Result
SWC-131	Presence of unused variables	CWE-1164: Irrelevant Code	✓
SWC-130	Right-To-Left-Override control character (U+202E)	CWE-451: User Interface (UI) Misrepresentation of Critical Information	✓
SWC-129	Typographical Error	CWE-480: Use of Incorrect Operator	✓
SWC-128	DoS With Block Gas Limit	CWE-400: Uncontrolled Resource Consumption	✓
SWC-127	Arbitrary Jump with Function Type Variable	CWE-695: Use of Low-Level Functionality	✓
SWC-125	Incorrect Inheritance Order	CWE-696: Incorrect Behavior Order	✓

ID	Title	Relationships	Test Result
SWC-124	Write to Arbitrary Storage Location	CWE-123: Write-what-where Condition	✓
SWC-123	Requirement Violation	CWE-573: Improper Following of Specification by Caller	✓
SWC-122	Lack of Proper Signature Verification	CWE-345: Insufficient Verification of Data Authenticity	✓
SWC-121	Missing Protection against Signature Replay Attacks	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-120	Weak Sources of Randomness from Chain Attributes	CWE-330: Use of Insufficiently Random Values	✓
SWC-119	Shadowing State Variables	CWE-710: Improper Adherence to Coding Standards	✓
SWC-118	Incorrect Constructor Name	CWE-665: Improper Initialization	✓
SWC-117	Signature Malleability	CWE-347: Improper Verification of Cryptographic Signature	✓
SWC-116	Timestamp Dependence	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-115	Authorization through tx.origin	CWE-477: Use of Obsolete Function	✓

ID	Title	Relationships	Test Result
SWC-114	Transaction Order Dependence	CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	✓
SWC-113	DoS with Failed Call	CWE-703: Improper Check or Handling of Exceptional Conditions	✓
SWC-112	Delegatecall to Untrusted Callee	CWE-829: Inclusion of Functionality from Untrusted Control Sphere	✓
SWC-111	Use of Deprecated Solidity Functions	CWE-477: Use of Obsolete Function	✓
SWC-110	Assert Violation	CWE-670: Always-Incorrect Control Flow Implementation	✓
SWC-109	Uninitialized Storage Pointer	CWE-824: Access of Uninitialized Pointer	✓
SWC-108	State Variable Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓
SWC-107	Reentrancy	CWE-841: Improper Enforcement of Behavioral Workflow	✓
SWC-106	Unprotected SELFDESTRUCT Instruction	CWE-284: Improper Access Control	✓
SWC-105	Unprotected Ether Withdrawal	CWE-284: Improper Access Control	✓

ID	Title	Relationships	Test Result
SWC-104	Unchecked Call Return Value	CWE-252: Unchecked Return Value	✓
SWC-103	Floating Pragma	CWE-664: Improper Control of a Resource Through its Lifetime	✓
SWC-102	Outdated Compiler Version	CWE-937: Using Components with Known Vulnerabilities	✓
SWC-101	Integer Overflow and Underflow	CWE-682: Incorrect Calculation	✓
SWC-100	Function Default Visibility	CWE-710: Improper Adherence to Coding Standards	✓

6.4 Unit Tests

DynamicRewardFarm.t.sol

1. Initialization and Basic Functionality:

- testStakeAndEarnRewards: Verifies the basic staking and reward earning mechanism.
- testMultipleUsersStaking: Ensures correct reward distribution among multiple users.
- testWithdrawStake: Checks the withdrawal functionality and subsequent reward calculations.

2. Reward Token Management:

- testAddNewRewardToken: Validates the process of adding a new reward token.
- testAddRewardTokenDuplicate: Ensures that duplicate reward tokens cannot be added.
- testAddRewardTokenLimit: Verifies the maximum limit of reward tokens (10) is enforced.
- testAddTokenZeroAddress: Checks that zero address cannot be added as a reward token.

3. Ownership and Management:

- testOwnershipTransfer: Verifies the ownership transfer process.
- testChangeManager: Ensures the manager can be changed correctly.
- testClaimOwnership: Checks the ownership claiming process.
- testInvalidNewOwner: Verifies that invalid new owners are rejected.
- testNewOwnerWrongAddress: Ensures zero address cannot be set as a new owner.
- testOnlyOwnerCanAddRewardToken: Confirms only the owner can add reward tokens.
- testOnlyOwnerFunctions: Verifies functions restricted to the owner.

4. Reward Calculations and Distribution:

- testEarnedFunction: Checks the calculation of earned rewards.
- testRewardPerToken: Verifies the reward per token calculation.
- testLastTimeRewardApplicable: Ensures correct timestamp for reward applicability.

5. Farm Operations:

- testExitFarm: Verifies the process of exiting the farm completely.
- testFarmWithdraw: Checks the partial withdrawal process.

6. Reward Rate Management:

- testManagerFunctions: Verifies functions restricted to the manager role.
- testSetRewardRatesInvalidLength: Ensures reward rates cannot be set with mismatched array lengths.
- testSetRewardRatesInvalidRate: Checks that invalid reward rates are rejected.
- testSetRewardRatesInvalidToken: Verifies that reward rates cannot be set for invalid tokens.
- testSetRewardRatesNoTokens: Ensures reward rates cannot be set without any tokens.

7. Token Recovery:

- testRecoverToken: Verifies the token recovery process for non-reward tokens.
- testCannotRecoverStakeOrRewardTokens: Ensures stake tokens and reward tokens cannot be recovered.
- testRecoverRewardTokenFromDeadAddress: Checks recovery of rewards from the dead address.

8. Transfer Functionality:

- testTransfer: Verifies the transfer of receipt tokens.
- testTransferFrom: Checks the transferFrom functionality for receipt tokens.

- testTransferReceiptTokens: Ensures correct transfer of receipt tokens and subsequent reward calculations.

9. Error Handling and Edge Cases:

- testInvalidDuration: Checks that invalid durations are rejected.
- testInvalidManager: Verifies that non-managers cannot perform manager-only actions.
- testInvalidOwner: Ensures non-owners cannot perform owner-only actions.
- testChangeDurationDuringRewardPeriod: Checks that reward duration cannot be changed during an active period.

Result

```
Ran 34 tests for contracts/DynamicRewardFarm.t.sol: DynamicRewardFarmTest
[PASS] testAddNewRewardToken() (gas: 622930)
[PASS] testAddRewardTokenDuplicate() (gas: 22606)
[PASS] testAddRewardTokenLimit() (gas: 9473127)
[PASS] testAddTokenZeroAddress() (gas: 15987)
[PASS] testCannotRecoverStakeOrRewardTokens() (gas: 52263)
[PASS] testChangeDurationDuringRewardPeriod() (gas: 16378)
[PASS] testChangeManager() (gas: 32698)
[PASS] testClaimOwnership() (gas: 46653)
[PASS] testEarnedFunction() (gas: 172456)
[PASS] testExitFarm() (gas: 366251)
[PASS] testFarmWithdraw() (gas: 337741)
[PASS] testInvalidDuration() (gas: 14232)
[PASS] testInvalidManager() (gas: 14247)
[PASS] testInvalidNewOwner() (gas: 41264)
[PASS] testInvalidOwner() (gas: 16380)
[PASS] testLastTimeRewardApplicable() (gas: 18055)
[PASS] testManagerFunctions() (gas: 390955)
[PASS] testMultipleUsersStaking() (gas: 503785)
[PASS] testNewOwnerWrongAddress() (gas: 13889)
```

```
[PASS] testOnlyOwnerCanAddRewardToken() (gas: 66382)
[PASS] testOnlyOwnerFunctions() (gas: 31889)
[PASS] testOwnershipTransfer() (gas: 88703)
[PASS] testRecoverRewardTokenFromDeadAddress() (gas: 341910)
[PASS] testRecoverToken() (gas: 1083760)
[PASS] testRewardPerToken() (gas: 148915)
[PASS] testSetRewardRatesInvalidLength() (gas: 84327)
[PASS] testSetRewardRatesInvalidRate() (gas: 81313)
[PASS] testSetRewardRatesInvalidToken() (gas: 108439)
[PASS] testSetRewardRatesNoTokens() (gas: 137182)
[PASS] testStakeAndEarnRewards() (gas: 337840)
[PASS] testTransfer() (gas: 497922)
[PASS] testTransferFrom() (gas: 506332)
[PASS] testTransferReceiptTokens() (gas: 497946)
[PASS] testWithdrawStake() (gas: 337907)
Suite result: ok. 34 passed; 0 failed; 0 skipped; finished in 296.71ms (168.56ms CPU time)
```

2. dynamic-reward-farm.test.js

1. Farm Initial Values:

- Correct farm name
- Correct farm symbol
- Correct farm decimals
- Correct farm supply
- Correct receipt balance for a given account
- Correct allowance for a given spender
- Correct staking token address

- Correct owner address
- Correct manager address
- Correct lastUpdateTime value
- Correct duration value
- Initialization with invalid duration is prevented

2. Duration Functionality:

- Ability to change farm duration value
- Only manager can change farm duration value
- Prevention of setting duration to 0
- Prevention of changing duration during active distribution

3. Reward Allocation Functionality:

- Prevention of setting reward rate to 0
- Correct setting of periodFinished date value
- Correct emission of RewardsAdded event
- Manager can set reward rates only if stakers exist
- Manager funds the farm during reward rate announcement
- Manager can increase rate at any time
- Manager can decrease rate only after distribution finished

4. User Reward Earning:

- Users earn rewards after depositing and setting reward rate

- Rewards are earned proportionally to stake time for all reward tokens
- Users can earn rewards from newly added reward tokens after interaction

5. Claiming Rewards:

- Correct transfer of reward amounts to user upon claiming
- (Note: There's a failing test for resetting userRewards mapping after claim)

6. Transfer Functionality:

- Correct transfer of staked tokens and reward updates
- Correct handling of transfers when users haven't interacted with new reward tokens

7. Withdrawing Functionality:

- Users can withdraw staked tokens
- Rewards are updated upon withdrawal

8. Adding New Reward Tokens:

- Manager can add new reward tokens
- Prevention of adding the same reward token twice
- New reward tokens start distributing after setting rate

9. Recover Token Functionality:

- Recovery of accidentally sent tokens
- Prevention of recovering stake tokens
- Prevention of recovering reward tokens

10. Edge Cases and Additional Scenarios:

- Correct handling of zero staked balance when calculating rewards
- Prevention of setting reward rates to zero when there are stakers

11. Adding New Reward Tokens After Cycles:

- Correct distribution of multiple reward tokens added after cycles

12. Transfer of Receipt Tokens and Reward Adjustment:

- Correct adjustment of rewards when users transfer receipt tokens, accounting for DEAD_ADDRESS

13. Multiple Users Staking and Reward Adjustment:

- Correct adjustment of rewards when new users join, accounting for DEAD_ADDRESS

Result
Contract: DynamicRewardFarm

Farm initial values
✓ should have correct farm name
✓ should have correct farm symbol
✓ should have correct farm decimals
✓ should have correct farm supply
✓ should return receipt balance for the given account
✓ should return the correct allowance for the given spender
✓ should have correct staking token address
✓ should have correct owner address
✓ should have correct manager address
✓ should have correct lastUpdateTime value
✓ should have correct duration value
✓ should not be able to initialize with wrong default duration value

Duration initial functionality
✓ should be able to change farm duration value
✓ should be able to change farm duration value only by manager
✓ should not be able to change farm duration value to 0
✓ should not be able to change farm duration during distribution

Reward allocation initial functionality by manager
✓ should not be able to set rate to 0
✓ should correctly set the periodFinished date value
✓ should emit correct RewardsAdded event
✓ manager should be able to set rewards rate only if stakers exist
✓ manager should fund the farm during reward rate announcement
✓ manager should be able to increase rate any time
✓ manager should be able to decrease rate only after distribution finished

User starts earning rewards after depositing and setting reward rate
✓ should allow user to earn rewards after depositing and setting reward rate

Earning functionality
✓ should earn rewards proportionally to stake time for all reward tokens
✓ should allow users to earn rewards from newly added reward tokens after interaction

Claiming Rewards
✓ should reset userRewards mapping after claim to 0 for all tokens
✓ should transfer correct reward amounts to user upon claiming

Transfer Functionality
✓ should transfer correct amount of staked tokens and update rewards accordingly
✓ should correctly handle transfers when users have not interacted with new reward tokens

Withdrawing Functionality
✓ should allow users to withdraw staked tokens
✓ should update rewards upon withdrawal

Adding New Reward Tokens
✓ should allow manager to add new reward tokens
✓ should prevent adding the same reward token twice
✓ should start distributing new reward token after setting its rate

Recover Token Functionality
✓ should be able to recover accidentally sent tokens from the contract
✓ should not be able to recover stakeTokens from the contract
✓ should not be able to recover rewardTokens from the contract

Edge Cases and Additional Scenarios
✓ should handle zero staked balance correctly when calculating rewards
✓ should prevent setting reward rates to zero when there are stakers

Adding New Reward Tokens After Cycles
✓ should distribute multiple reward tokens added after cycles

Transfer of Receipt Tokens and Reward Adjustment
✓ should adjust rewards when user transfers receipt tokens to another user, accounting for DEAD_ADDRESS

Multiple Users Staking and Reward Adjustment
✓ should adjust rewards when another user joins, accounting for DEAD_ADDRESS

Coverage
```
| File | % Lines | % Statements | % Branches | % Funcs |
| contracts/DynamicRewardFarm.sol | 100.00% (131/131) | 100.00% (169/169) | 100.00% (35/35) | 100.00% (32/32) |
```

6.5 Verify Claims

6.5.1 **Compliance with Smart Contract Best Practices**: The audit will ensure that the contracts adhere to established smart contract best practices, including identifying vulnerabilities such as reentrancy, overflow, and underflow issues. The goal is to confirm the contracts are robust against these common threats.
Status: tested and verified ✅

6.5.2 **Access Control and Permissions**: The contracts should incorporate strict access control mechanisms to ensure only authorized entities can access or modify sensitive functions. This will prevent unauthorized actions that could compromise the system.
Status: tested and verified ✅

6.5.3 **Gas Efficiency**: The contracts will be reviewed for optimization in terms of gas usage. Efficient gas usage will minimize unnecessary computational costs and storage operations, contributing to lower transaction fees for users.
Status: tested and verified ✅

6.5.4 **Data Integrity and State Consistency**: The audit will ensure that data integrity and consistency are maintained across all operations. All state changes should accurately reflect the intended actions, and data loss or corruption should be prevented.
Status: tested and verified ✅

6.5.5 **Security of Reward Distribution**: The contracts should ensure that reward calculations and distributions are handled accurately and fairly. This includes verifying that rewards are allocated correctly based on staking activity, without any potential for exploitation or manipulation of the reward system.
Status: tested and verified ✅

7. Executive Summary

Two independent softstack experts performed an unbiased and isolated audit of the smart contract codebase provided by the Bitcoin.com team. The main objective of the audit was to verify the security and functionality claims of the smart contract. The audit process involved a thorough manual code review and automated security testing.

Overall, the audit identified a total of one issue, classified as follows:
- No critical issues were found.
- No high severity issues were found.
- No medium severity issues were found.
- 4 low severity issues were discovered
- 3 informational issues were identified

The audit report provides detailed descriptions of each identified issue, including severity levels, CWE classifications, and recommendations for mitigation. It also includes code snippets, where applicable, to demonstrate the issues and suggest possible fixes. We recommend that the Bitcoin.com team review the suggestions.

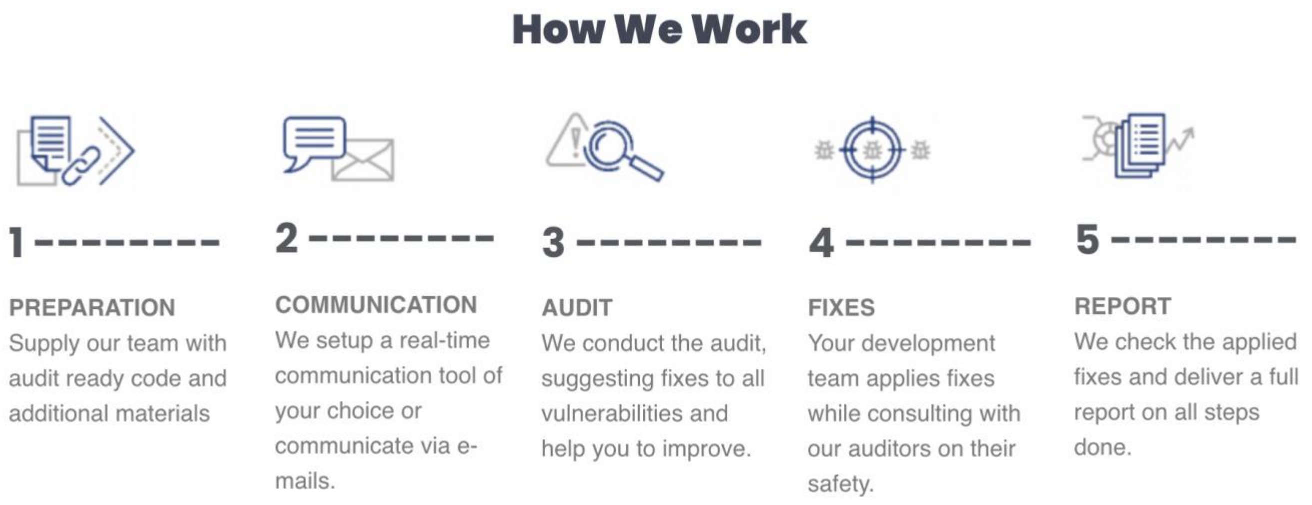
8. About the Auditor

Established in 2017 under the name Chainsulting, and rebranded as softstack GmbH in 2023, softstack has been a trusted name in Web3 Security space. Within the rapidly growing Web3 industry, softstack provides a comprehensive range of offerings that include software development, cybersecurity, and consulting services. Softstack's competency extends across the security landscape of prominent blockchains like Solana, Tezos, TON, Ethereum and Polygon. The company is widely recognized for conducting thorough code audits aimed at mitigating risk and promoting transparency.

The firm's proficiency lies particularly in assessing and fortifying smart contracts of leading DeFi projects, a testament to their commitment to maintaining the integrity of these innovative financial platforms. To date, softstack plays a crucial role in safeguarding over $100 billion worth of user funds in various DeFi protocols.

Underpinned by a team of industry veterans possessing robust technical knowledge in the Web3 domain, softstack offers industry-leading smart contract audit services. Committed to evolving with their clients' ever-changing business needs, softstack's approach is as dynamic and innovative as the industry it serves.

Check our website for further information: https://softstack.io



How We Work

1 --------

PREPARATION
Supply our team with audit ready code and additional materials

2 --------

COMMUNICATION
We setup a real-time communication tool of your choice or communicate via e-mails.

3 --------

AUDIT
We conduct the audit, suggesting fixes to all vulnerabilities and help you to improve.

4 --------

FIXES
Your development team applies fixes while consulting with our auditors on their safety.

5 --------

REPORT
We check the applied fixes and deliver a full report on all steps done.