Privacy Ninja

SMART CONTRACT

Security Audit Report

Project:	Bit Ultra (BLT)
Platform:	Binance Smart Chain
Language:	Solidity
Date:	October 9th, 2024

Table of contents

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO PUBLIC AFTER ISSUES ARE RESOLVED.

Introduction

Privacy Ninja was contracted by the BLT team to perform the Security audit of the Bit Ultra (BLT) smart contracts code. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on October 9th, 2024.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contracts.

Project Background

- BLT Contracts handle multiple contracts, and all contracts have different functions.
- Here's a brief overview of the key components and functionalities of the provided code:
    - **BitUltra:** The contract implements an ERC20 token called BitUltra (BLT) with special features, such as a whitelist system that controls which addresses can receive tokens via transfers.
    - **BLTVaultContract:** This contract manages a token release schedule for the BLT token, governed by phases and blocks.
- There are 2 smart contracts files that were included in the audit scope.

Audit scope

Name	Code Review and Security Analysis Report for Bit Ultra (BLT)Smart Contracts
Platform	**Binance Smart Chain / Solidity**
File 1	BitUltra.sol
File 1 Smart Contract	0xB95462682257e272E7D32c4214A3197a3B7cCf5e
File 2	BLTVaultContract.sol
File 2 Smart Contract	0x99482b6fb63f6367571dbe629380de0836edbebc
Audit Date	October 9th, 2024

Claimed Smart Contracts Features

Claimed Feature Detail	Our Observation
File 1 : BitUltra.sol • Name: Bit Ultra • Ticker: BLT • Max supply: 2.1 billion BLT tokens • Decimal: 18 **Owner Specification:** • The whitelistUsers function allows the contract owner to whitelist multiple addresses at once, ensuring that only whitelisted addresses can receive tokens when the whitelist is active. • The disableWhitelist function allows the owner to permanently disable the whitelist once, locking the functionality to prevent future modifications. • The current owner can transfer the ownership. • The owner can renounce ownership.	**YES, This is valid.** **We advise renouncing ownership once the ownership functions are no longer needed. This will make the smart contract 100% decentralized.**
File 2: BLTVault.sol **Owner Specification:** • Calculate and claim tokens based on the current block and phase. • Start the release phases by setting the start block for phase 1. • The current owner can transfer the ownership. • The owner can renounce ownership.	**YES, This is valid.** **We advise renouncing ownership once the ownership functions are no longer needed. This will make the smart contract 100% decentralized.**

Audit Summary

According to the standard audit assessment, the Customer`s solidity-based smart contracts are **“Secured”**.Also, these contracts contain owner control, which does not make them fully decentralized.

We used various tools like MythX, Slither, and Remix IDE. At the same time, this finding is based on a critical analysis of the manual audit.
All issues found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the Audit Overview section. The general overview is presented in the AS-IS section and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium, 0 low, and 7 very low-level issues.

Investor Advice: A technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner-controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	**The solidity** version **is** not specified	Passed
	The solidity version **is** too old	**Passed**
	Integer overflow/underflow	Passed
	Function input parameters lack check	**Passed**
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	**Passed**
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	Passed
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	**Moderated**
	Features claimed	Passed
	Other programming issues	**Moderated**
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Other code specification issues	Passed
Gas Optimization	“Out of Gas” Issue	**Passed**
	High consumption ‘for/while’ loop	**Moderated**
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage **is** not set	**Passed**
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

Code Quality

This audit scope has 2 smart contract files. These smart contracts also contain Libraries, Smart contracts inherits, and Interfaces. These are compact and well-written contracts.

The libraries in the BLT are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address and its properties/methods can be reused many times by other contracts in the Fraxtor.

The BLT team has not provided scenario and unit test scripts, which would helped to determine the integrity of the code in an automated way.

Some code parts are **well-commented** on smart contracts.

Documentation

We were given BLT smart contract code in the form of a BitUltra and BLTVaultContract web URL.

As mentioned above, some code parts are **well-commented**. So, it is difficult to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries used in these smart contracts infrastructures are based on well-known industry-standard open-source projects. And their core code blocks are written well.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

BitUltra.sol

(1) Interface

(a) IERC20

(b) IERC20Metadata

(c) IERC20Errors

(d) IERC721Errors

(e) IERC1155Errors

(2) Inherited Contracts

(a) ERC20

(b) Ownable

(3) Events

(a) event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

(b) event Transfer(address indexed from, address indexed to, uint256 value);

(c) event Approval(address indexed owner, address indexed spender, uint256 value);

(4) Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	whitelistUsers	external	Infinite loops possibility, Lack of Functionality to Remove Whitelisted Users	Refer Audit Findings
3	disableWhitelist	external	access only owner	No Issue
4	transfer	write	Passed	No Issue
5	onlyOwner	modifier	Passed	No Issue
6	owner	read	Passed	No Issue
7	_checkOwner	internal	Passed	No Issue

8	renounceOwnership	write	access only owner	No Issue
9	transferOwnership	write	access only owner	No Issue
10	_transferOwnership	internal	Passed	No Issue
11	name	read	Passed	No Issue
12	symbol	read	Passed	No Issue
13	decimals	read	Passed	No Issue
14	totalSupply	read	Passed	No Issue
15	balanceOf	read	Passed	No Issue
16	transfer	write	Passed	No Issue
17	allowance	read	Passed	No Issue
18	approve	write	Passed	No Issue
19	transferFrom	write	Passed	No Issue
20	_transfer	internal	Passed	No Issue
21	_update	internal	Passed	No Issue
22	_mint	internal	Passed	No Issue
23	_burn	internal	Passed	No Issue
24	_approve	internal	Passed	No Issue
25	_approve	internal	Passed	No Issue
26	_spendAllowance	internal	Passed	No Issue

BLTVault.sol

(1) Interface

(a) IERC20

(2) Inherited Contracts

(a) Ownable

(3) Events

(a) event TokensReleased(uint256 amount, uint256 currentBlock);

(b) event PhaseStarted(uint256 phaseIndex, uint256 startBlock);

(c) event AllPhasesCompleted();

(d) event Transfer(address indexed from, address indexed to, uint256 value);

(e) event Approval(address indexed owner, address indexed spender, uint256 value);

(f) event OwnershipTransferred(address indexed previousOwner, address indexed newOwner);

(4) Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	startPhases	external	access only owner	No Issue
3	claimTokens	external	Potential High Gas Usage in claimTokens, Inefficient Phase Completion Logic	Refer Audit Findings
4	getCurrentPhase	read	Inefficient Phase Completion Logic	Refer Audit Findings
5	getLastClaimedBlock	read	Passed	No Issue
6	getReleaseSchedule	external	Compile time warnings	Refer Audit Findings
7	getCurrentSubpart	read	Passed	No Issue
8	onlyOwner	modifier	Passed	No Issue
9	owner	read	Passed	No Issue
10	_checkOwner	internal	Passed	No Issue
11	renounceOwnership	write	access only owner	No Issue
12	transferOwnership	write	access only owner	No Issue
13	_transferOwnership	internal	Passed	No Issue

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical

No critical severity vulnerabilities were found.

High

No high-severity vulnerabilities were found.

Medium

No medium-severity vulnerabilities were found.

Low

No low-severity vulnerabilities were found.

Very Low / Discussion / Best practices:

(1) Infinite loops possibility: **BitUltra.sol**

As array elements will increase, then it will cost more and more gas. And eventually, it will stop all the functionality. After several hundreds of transactions, all those functions depending on it will stop. We suggest avoiding loops. For example, use mapping to store the array index. And query that data directly, instead of looping through all the elements to find an element.

Resolution: Adjust logic to replace loops with mapping or other code structures.

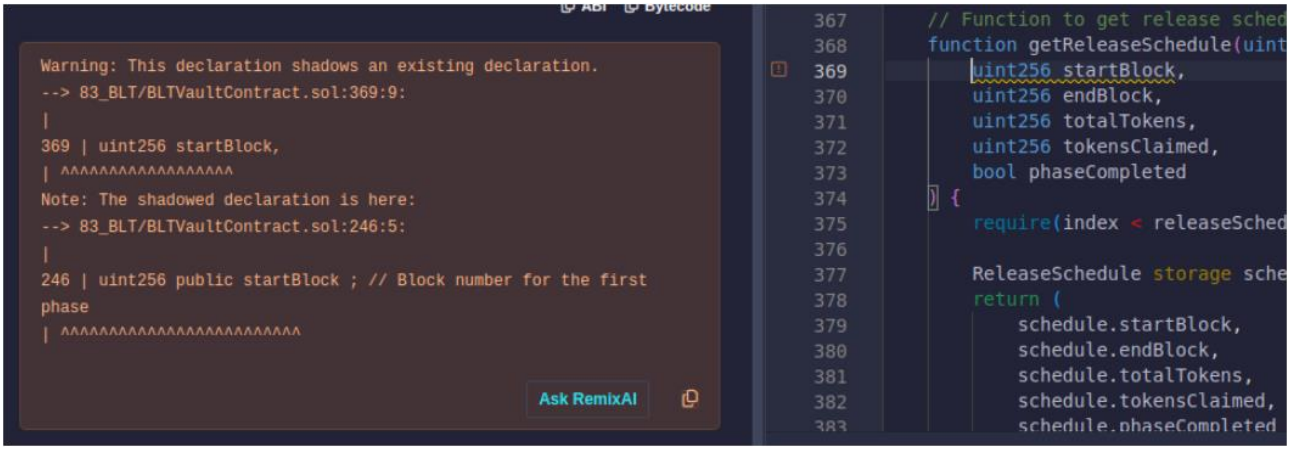
- whitelistUsers() - users.length.

(2) Lack of Functionality to Remove Whitelisted Users: **BitUltra.sol**

The BitUltra contract currently lacks a mechanism for the owner to remove addresses from the whitelist. This limitation can lead to security concerns, as it prevents the owner from revoking access from users who may no longer need it or may have been compromised.

Resolution: We suggest implementing a remove whitelist user functionality to enhance security and flexibility. Ensuring it includes checks for whitelistEnabled and whitelistLocked to maintain the integrity of the whitelist system. If this is a part of the plan then disregard this issue.

(3) Compile time warnings: **BLTVaultContract.sol**



Warning: This declaration shadows an existing declaration.

Resolution: The warning "This declaration shadows an existing declaration" indicates that you have a variable or function name that conflicts with another one in the same scope or a parent scope. To resolve this, you should rename the variable or function to something unique. Check the surrounding code for existing declarations with the same name and choose a distinct name to avoid ambiguity.

(4) Unused Mapping for Last Claimed Block: **BLTVaultContract.sol**

```
mapping(address => uint256) private lastClaimedBlock;
```

The lastClaimedBlock mapping is declared but never updated or used in the contract, making it redundant.

Resolution: Update the lastClaimedBlock mapping whenever a user successfully claims tokens.

For example, after transferring tokens in the claimTokens function, set lastClaimedBlock[msg.sender] = block.number;

(5) Potential High Gas Usage in claimTokens: **BLTVaultContract.sol**

The claimTokens function iterates over all phases, which can consume a lot of gas if there are many phases or if called frequently.

Resolution: Optimize the function by adding a mechanism to skip already claimed phases or breaking the claiming process into smaller chunks. Consider allowing users to claim tokens for specific phases instead of all at once.

(6) Missing Specific User Claim Events: **BLTVaultContract.sol**

The TokensReleased event indicates that tokens have been released but does not specify which user claimed them. This can make it challenging to track individual claims, especially in cases where multiple users are interacting with the contract.

Resolution: To enhance transparency and accountability, you should emit a separate event specifically for user claims. This could look like:

event UserTokensClaimed(address indexed user, uint256 amount);

Then, within the claimTokens function, emit this event right after transferring the tokens:

emit UserTokensClaimed(msg.sender, totalToClaim);

(7) Inefficient Phase Completion Logic: **BLTVaultContract.sol**

```
 // Check if all phases are completed
        if (getCurrentPhase() >= releaseSchedules.length) {
            allPhasesCompleted = true;
            emit AllPhasesCompleted();
        }


// Helper function to get the current phase based on the block
number
    function getCurrentPhase() public view returns (uint256) {
        require(releaseSchedules[0].startBlock > 0, "Phases have
not started");
        uint256 currentBlock = block.number;
```

```
        for (uint256 i = 0; i < releaseSchedules.length; i++) {
            ReleaseSchedule storage schedule =
releaseSchedules[i];
            if (currentBlock >= schedule.startBlock &&
currentBlock <= schedule.endBlock) {
                return i;
            }
        }

        return releaseSchedules.length; // If beyond all phases
    }
```

The logic for checking if all phases are completed within claimTokens can be inefficient and could lead to unnecessary calculations.

Resolution: Maintain a counter or boolean that tracks the number of completed phases. Update this counter whenever a phase is marked as complete, rather than recalculating each time in the claim function.

Centralization

This smart contract has some functions that can be executed by the Admin (Owner) only. If the admin wallet's private key is compromised, then it would create trouble. The following are Admin functions:

BitUltra.sol

- whitelistUsers: The owner can whitelist multiple addresses.
- disableWhitelist: The owner can disable the whitelist mechanism, but can only be called once.

BLTVaultContract.sol

- startPhases: Start the release phases by setting the start block for phase 1 by the owner.
- claimTokens: The owner can calculate and claim tokens based on the current block and phase.

To make the smart contract 100% decentralized, we suggest renouncing ownership in the smart contract once its function is completed.

Conclusion

We were given a contract code in the form a BitUltra and BLTVaultContract web URL. And we have used all possible tests based on given objects as files. We observed 7 very low issues in the smart contracts. But those are not critical. **So, it's good to go for the mainnet deployment.**

Since possible test cases can be unlimited for such smart contract, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed contract, based on standard audit procedure scope, is **"Secured".**

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of the systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and white box penetration testing. We look at the project's website to get a high-level understanding of the functionality of the software under review. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

Privacy Ninja Pte. Ltd. Disclaimer

The smart contract given for audit has been analyzed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Because the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

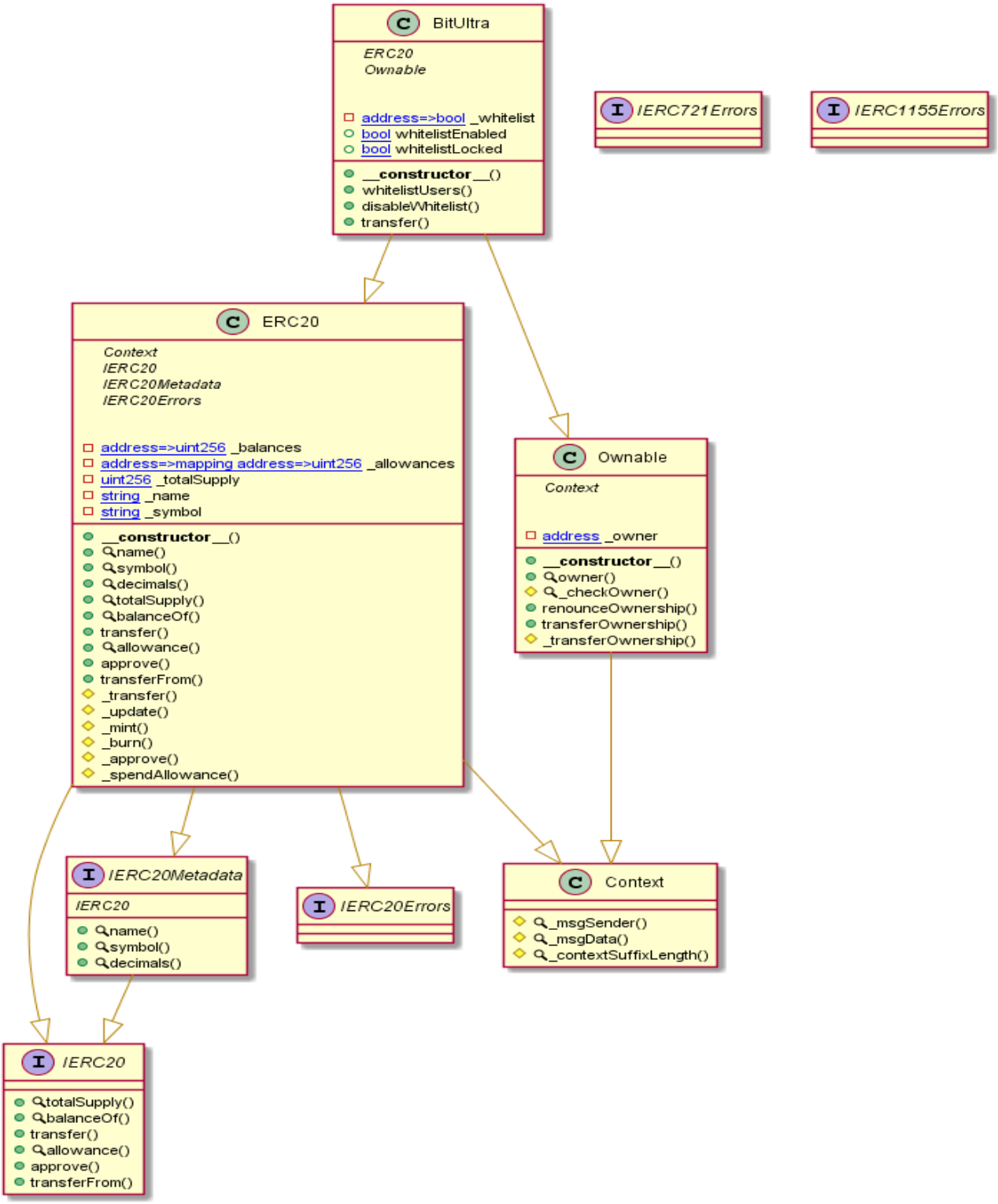
Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee the explicit security of the audited smart contracts.

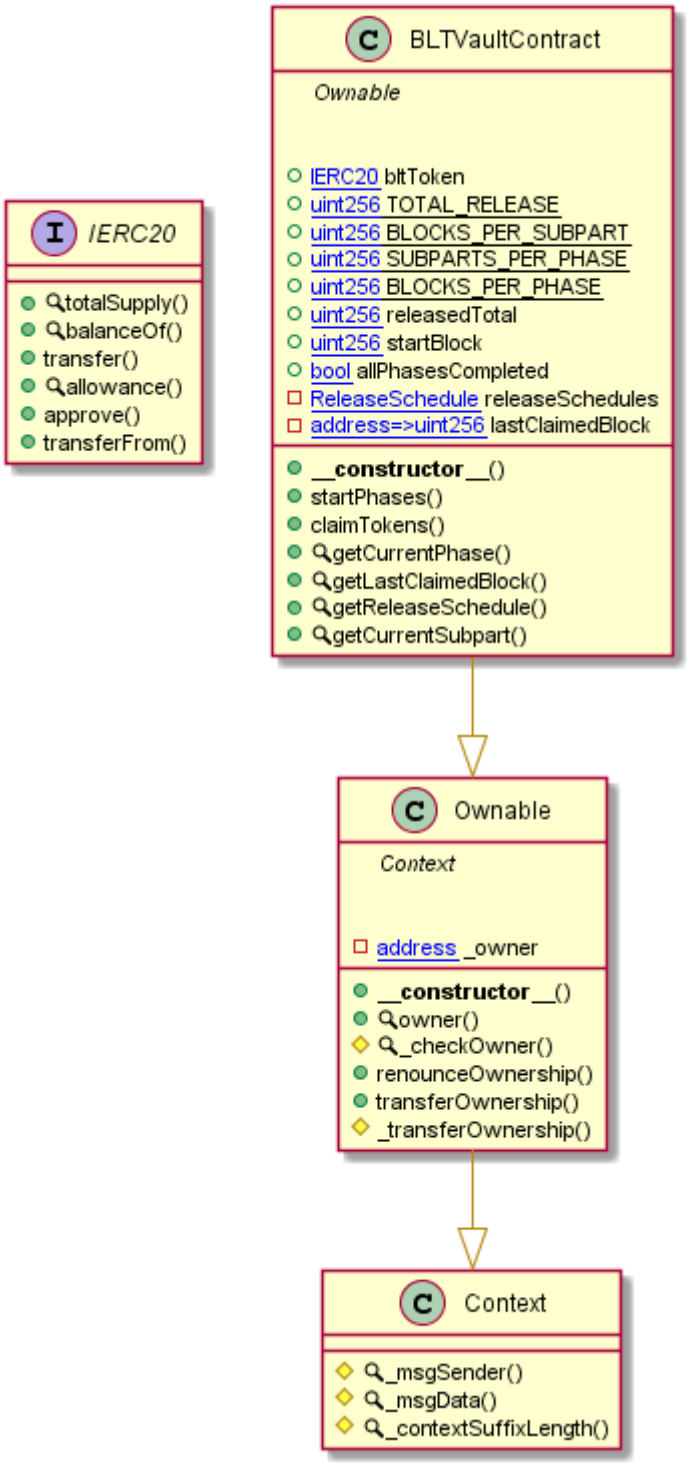
Appendix

Code Flow Diagram - Bit Ultra (BLT)

BitUltra Diagram



BLTVault Diagram



Slither Results Log

Slither log >> BitUltra.sol

```
INFO:Detectors:
Context._contextSuffixLength() (BitUltra.sol#142-144) is never used and should be removed
Context._msgData() (BitUltra.sol#138-140) is never used and should be removed
ERC20._burn(address,uint256) (BitUltra.sol#552-557) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version^0.8.20 (BitUltra.sol#11) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.20 (BitUltra.sol#93) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.20 (BitUltra.sol#121) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.20 (BitUltra.sol#151) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.20 (BitUltra.sol#316) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.20 (BitUltra.sol#634) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
Pragma version^0.8.20 (BitUltra.sol#734) necessitates a version too recent to be trusted. Consider deploying with 0.8.18.
solc-0.8.20 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
BitUltra.constructor(address) (BitUltra.sol#743-748) uses literals with too many digits:
	- _mint(msg.sender,2100000000 * 10 ** decimals()) (BitUltra.sol#747)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
INFO:Slither:BitUltra.sol analyzed (9 contracts with 93 detectors), 12 result(s) found
```

Slither log >> BLTVault.sol

```
INFO:Detectors:
BLTVaultContract.claimTokens() (BLTVault.sol#286-345) ignores return value by bltToken.transfer(msg.sender,totalToClaim) (BLTVault.sol#336)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
BLTVaultContract.lastClaimedBlock (BLTVault.sol#250) is never initialized. It is used in:
	- BLTVaultContract.getLastClaimedBlock(address) (BLTVault.sol#363-365)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
INFO:Detectors:
BLTVaultContract.claimTokens() (BLTVault.sol#286-345) performs a multiplication on the result of a division:
	- claimableSubParts = claimableBlocks / BLOCKS_PER_SUBPART (BLTVault.sol#303)
	- claimableTokens = claimableSubParts * schedule.tokensPerSubPart (BLTVault.sol#304)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
BLTVaultContract.claimTokens() (BLTVault.sol#286-345) uses a dangerous strict equality:
	- lastBlockToClaim == schedule.endBlock (BLTVault.sol#312)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities
INFO:Detectors:
Reentrancy in BLTVaultContract.claimTokens() (BLTVault.sol#286-345):
	External calls:
	- bltToken.transfer(msg.sender,totalToClaim) (BLTVault.sol#336)
```

```
        State variables written after the call(s):
        - allPhasesCompleted = true (BLTVault.sol#342)
        BLTVaultContract.allPhasesCompleted (BLTVault.sol#247) can be used in cross function reentrancies:
        - BLTVaultContract.allPhasesCompleted (BLTVault.sol#247)
        - BLTVaultContract.claimTokens() (BLTVault.sol#286-345)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
BLTVaultContract.getReleaseSchedule(uint256).startBlock (BLTVault.sol#369) shadows:
        - BLTVaultContract.startBlock (BLTVault.sol#246) (state variable)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing
INFO:Detectors:
Reentrancy in BLTVaultContract.claimTokens() (BLTVault.sol#286-345):
        External calls:
        - bltToken.transfer(msg.sender,totalToClaim) (BLTVault.sol#336)
        Event emitted after the call(s):
        - AllPhasesCompleted() (BLTVault.sol#343)
        - TokensReleased(totalToClaim,currentBlock) (BLTVault.sol#338)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
Different versions of Solidity are used:
        - Version used: ['^0.8.0', '^0.8.20']
        - ^0.8.0 (BLTVault.sol#224)
        - ^0.8.20 (BLTVault.sol#11)
        - ^0.8.20 (BLTVault.sol#93)
        - ^0.8.20 (BLTVault.sol#124)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
INFO:Detectors:
Context._contextSuffixLength() (BLTVault.sol#114-116) is never used and should be removed
Context._msgData() (BLTVault.sol#110-112) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version^0.8.20 (BLTVault.sol#11) necessitates a version too recent to be trusted. Consider deploying with
0.8.18.
Pragma version^0.8.20 (BLTVault.sol#93) necessitates a version too recent to be trusted. Consider deploying with
0.8.18.
Pragma version^0.8.20 (BLTVault.sol#124) necessitates a version too recent to be trusted. Consider deploying with
0.8.18.
Pragma version^0.8.0 (BLTVault.sol#224) allows old versions
solc-0.8.20 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Loop condition i < releaseSchedules.length (BLTVault.sol#394) should use cached array length instead of referencing
`length` member of the storage array.
 Loop condition i < releaseSchedules.length (BLTVault.sol#294) should use cached array length instead of
referencing `length` member of the storage array.
 Loop condition i < releaseSchedules.length (BLTVault.sol#352) should use cached array length instead of
referencing `length` member of the storage array.
 Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cache-array-length
INFO:Slither:BLTVault.sol analyzed (4 contracts with 93 detectors), 18 result(s) found
```

Privacy Ninja