Blaize.Security

BRAINSTARTER
SMART CONTRACT AUDIT

August 27th, 2024 / V. 1.0

BLAIZE.SECURITY
WE SECURE WEB3 ECOSYSTEMS

Blaize.Security is a world-class web3 security provider that works with up-to-date technologies to build a safe environment for entire ecosystems. We provide security services for over 25 chains and ecosystems, and the list grows.

Blaize.Security has over 6 years in the web3 industry and offers a dedicated team of over 25 certified Security Researchers and Auditors to cover all platform components. Therefore we have separate teams for each security direction and service.

Pavlo Horbonos, Head of Security in Blaize

Software Engineer and Security Researcher with decades of experience in decentralized technologies, fintech, low-level programming, cyber + web3 security, and AI and Data Science. Certified auditor and security researcher.

https://www.linkedin.com/in/pavelmidvel/
https://twitter.com/MidvelCorp

Check more info on
https://security.blaize.tech

X (ex-Twitter): **https://x.com/BlaizeSecurity**

Linkedin: **https://www.linkedin.com/company/blaize-security**

Contact us:
security@blaize.tech

Table of Contents

Executive Summary

During the audit, we examined the security of smart contracts for the **BrainStarter** protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the BrainStarter smart contracts conducted between **July 31st** and **August 22nd, 2024**.

Blaize.Security conducted a comprehensive audit of the BrainStarter protocol, focusing on its smart contracts for token staking and liquidity management. The audit scope included 7 separate smart contracts developed by the BrainStarter team. The protocol features a system for managing $BRAINS tokens, allowing users to stake tokens and receive either liquid or illiquid NFTs based on staking thresholds. Liquid stakes are transferable, while illiquid stakes remain locked and non-transferable. The audit also covered the mechanisms for token minting, burning, and fee calculations, ensuring that the protocol's logic is secure and functions as intended.

More information about the contract logic is detailed in the Protocol Overview section.

The security team provided the decomposition of the protocol, checked the integrity of the business logic, funds flow, access control system, and user flow, and analyzed a set of edge cases. The team also checked the system against several checklists (including all standard vulnerabilities) and conducted an intensive testing stage. The team found several issues of different levels of risk. All issues are described in the Complete Analysis section. The BrainStarter team resolved or verified most of them.

However, the security team should note several concerns.

1.Controllable backdoor, centralization, and single point of failure.
All platform contracts are upgradeable - thus, the owner can change any aspect of the logic of each platform component. While upgradeability is a regular practice, it creates a controllable backdoor with a significant risk of exploitation in case of inappropriate private key handling. As all contracts in the scope are upgradeable, such an approach leads to centralization and possible issues in case of the admin's private key leak.

2.Token upgradability.
The tokens issued by the platform are also upgradeable. Typically, tokens are made non-upgradeable because they represent users' assets, and the upgradeability can cause trust issues. The team confirmed, that token will serve for utility purposes only, still auditors see such approach as sub-standard.

3.Centralization and control over tokens.
The designated role has full control over NFTs (can mint/burn without restrictions). The same applies to Dopamine tokens, as the owner can mint tokens without restrictions and burn tokens without the user's approval. The team confirmed, that is a part of the business logic, because of pure utility of assets. Nevertheless it currently violates users' balances and is a negative centralization factor.

4.Burnable token without restrictions.
The tokens implement the public burn() / burnFrom() interface via the OZ ERC20Burnable interface. While this is standard practice, the Customer should be aware of the potential risks associated with it, listed in the Info-3 issue.

5.Tokens implement the ERC-2612 Permit interface.
While it is a regular practice, it requires additional monitoring of its usage and the cautious behavior of users. Permit functionality creates several potentially dangerous situations for users:
- the signed permit cannot be withdrawn; thus, it exists until it is either used or re-written by the alternative signed permit
– users should validate the deadline for the signature, as in the case of the unvalidated considerable number, it will exist until executed (as it cannot be withdrawn)
- several simultaneous signatures may exist for the same nonce, creating a racing condition
Thus, users and the Customer should be aware of these conditions and inform the community accordingly.

Therefore, the security team currently evaluates the project as **Highly Secure**, though auditors noted the list of areas for improvement (noted with failed standard areas' checks) and exposure to several risks described above.

Auditing strategy and Techniques applied/Procedure

Blaize.Security auditors start the audit by developing an auditing strategy - an individual plan where the team plans methods, techniques, approaches for the audited components. That includes a list of activities:

MANUAL AUDIT STAGE

- Manual line-by-line code by at least 2 security auditors with crosschecks and validation from the security lead;
- Protocol decomposition and components analysis with building an interaction scheme, depicting internal flows between the components and sequence diagrams;
- Business logic inspection for potential loopholes, deadlocks, backdoors;
- Math operations and calculations analysis, formula modeling;
- Access control review, roles structure, analysis of user and admin capabilities and behavior;
- Review of dependencies, 3rd parties, and integrations;
- Review with automated tools and static analysis;
- Vulnerabilities analysis against several checklists, including internal Blaize.Security checklist;
- Storage usage review;
- Gas (or tx weight or cross-contract calls or another analog) optimization;
- Code quality, documentation, and consistency review.

and a wide spectrum of other vulnerable areas.

FOR ADVANCED COMPONENTS:

- Cryptographical elements and keys storage/usage audit (if applicable);
- Review against OWASP recommendations (if applicable);
- Blockchain interacting components and transactions flow (if applicable);
- Review against CCSSA (C4) checklist and recommendations (if applicable);

TESTING STAGE:

- Development of edge cases based on manual stage results for false positives validation;
- Integration tests for checking connections with 3rd parties;
- Manual exploratory tests over the locally deployed protocol;
- Checking the existing set of tests and performing additional unit testing;
- Fuzzy and mutation tests (by request or necessity);
- End-to-end testing of complex systems;

In case of any issues found during audit activities, the team provides detailed recommendations for all findings.

POST-AUDIT STEPS RECOMMENDED

To ensure the security of the contract, the **Blaize.Security** team suggests that the team follow post-audit steps:

1. Request audits of other protocol components (dApp, backend, wallet, blockchain, etc) from Blaize Security
2. Request consulting and deployment overwatch services provided by Blaize Security
3. Launch active protection over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
4. Launch a **bug bounty program** to encourage further active analysis of the smart contracts.
5. Request post-deployment assessment service provided by Blaize Security to ensure the correctness of the configuration, settings, cross-connections of deployed entities and live functioning

Audit Rating

Score: **9.73**/10

	RATING
Security	9.73
Logic optimization	10
Code quality	9.9
Testing suite	7.3
Documentation	9.5

Security: General mark for the security of the protocol.
The main mark for the audit qualification.

Logic optimization: Evaluation of how optimal the implementation is, including presence of extra/unused code, uncovered/extra cases, gas (or its analog) optimization, memory management optimization, etc
Code quality: Evaluation of best practices followed, code readability, structure and convenience of further development
Testing suite: Availability of the native tests suite, level of logic coverage, checks of critical areas being covered.
Documentation: Availability and quality of the documentation, coverage of core functionality and user flows: whitepaper, gitbook, readme, specs, natspec, comments in the code and other possible forms of documentation.

SECURITY RATING CALCULATION

Approximate weight of unresolved issues.
Critical: -3 points
High: -2 points
Medium: -0.5 points
Low: -0.1 points
Informational: -0.1 point (in general, depends on the context)

Note: additional concerns, violated checklist items (including standard vulnerabilities), and verified backdoors may influence the final mark and weight of certain issues.

Starting with a perfect score of 10:

Critical issues: 3 issues (3 resolved): 0 points deducted

High issues: 1 issue (1 resolved): 0 points deducted. While the issue is verified, it still raises concerns from auditor's side regarding the absence of the reliable oracle

Medium issues: 1 issue (1 resolved): 0 points deducted

Low issues: 4 issues (4 resolved): 0 points deducted

Informational issues: 6 issues (6 verified): -0.05 points deducted based on the concern regarding locked tokens (Info-2)

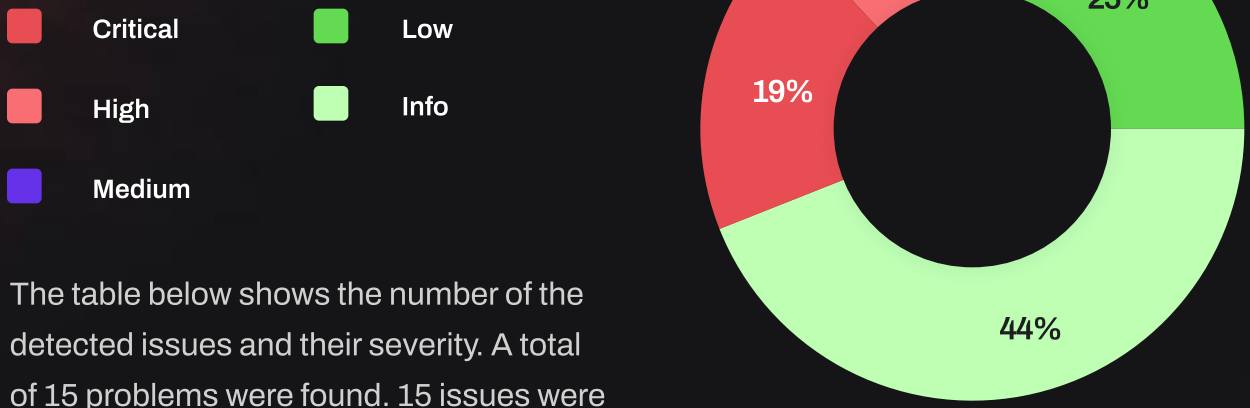
Best practices: partly resolved, -0.02 points deducted.

Other: -0.2 points deducted for the failed check against the backdoor (upgradeability of contracts), and centralization risk.

Security rating = 10 - 0.05 - 0.02 - 0.2 = 9.73

Technical Summary

THE GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 15 problems were found. 15 issues were fixed or verified by the Customer's team.

	FOUND	FIXED/VERIFIED
Critical	3	3
High	1	1
Medium	1	1
Low	4	4
Info	6	6

Best practices and optimizations

- 4 items resolved
- 4 items unresolved

Section is marked as **unresolved**, as half of recommendations are unaddressed

SEVERITY DEFINITION

CRITICAL

The system contains several issues ranked as very seriousand dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

HIGH

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

MEDIUM

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

LOW

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

INFO

The issue has no impact on the contract's ability to operate, yet is relevant for best practices. Or this status can be assigned to the issues related to the suspicious activity or substandard business logic decisions which cannot be classified without the comments from the team (and can be re-classified on the later audit stages).

Issues reviewed by the team can get the next statuses:
Resolved: issue is resolved by an appropriate patch or changes in the business logic
Verified: the team provided sufficient evidences that the issue describes desired behavior
Unresolved: neither path nor comments provided by the team, or they are not sufficient to resolve the issue
Acknowledged: the team accepts the misbehavior and connected risks

Audit Scope

Language/Technology: **Solidity**

Blockchain: **Polygon**

The scope of the project includes:

- contracts\Brains.sol
- contracts\BrainsReceiptLocker.sol
- contracts\BrainsStaking.sol
- contracts\Dopamine.sol
- contracts\IlliquidStake.sol
- contracts\LiquidStake.sol
- contracts\UnlockFeeCalculator.sol

Repository: https://github.com/codefunded/p-brainstarter-contracts

The source code of the smart contract was taken from the branch: main.

Initial commit:

- a80efe9ae1705e31adf46e340c0c6d03ece89002

Final commit:

- ab203b656f996b7ff80f025a41305227d8e5a211

Protocol overview

DESCRIPTION

Brains.sol:
A token contract implementing ERC20, ERC20Burnable, ERC20Permit, and Ownable features. It includes a minting mechanism with a yearly limit. Batch transfer functionality is provided. The contract is upgradable via UUPS.

BrainsReceiptLocker.sol:
Manages the exchange of receipt tokens from various sale phases (pre-sale, strategic, seed) into actual $BRAINS tokens and stakes them. It integrates with the BrainsStaking contract for staking operations.

BrainsStaking.sol:
The BrainsStaking contract manages the staking process within the BrainStarter protocol. It allows users to stake their $BRAINS tokens and receive either liquid or illiquid stake NFTs based on the staking amount and configured thresholds. Liquid stakes are represented by ERC721 tokens that are transferable and tradeable. Liquid stakes are minted when the staking amount exceeds the configured liquidStakeThreshold. Illiquid stakes are represented by non-transferable ERC721 tokens. Illiquid stakes are minted when the staking amount does not exceed the liquidStakeThreshold or the remaining amount after minting liquid stakes is less than the threshold.

The proportion of liquid to illiquid stakes is determined by the liquidStakeThreshold parameter. Here's how it works:

1. Stake Amount Below Threshold:
If the staking amount is less than the liquidStakeThreshold, the entire amount is converted into an illiquid stake.

2. Stake Amount Equal to or Above Threshold:
If the staking amount is equal to or greater than the liquidStakeThreshold, the following steps occur:
The maximum possible number of liquid stakes are minted, each representing an amount equal to the liquidStakeThreshold. Any remaining amount that is less than the liquidStakeThreshold is converted into an illiquid stake.

It appears that the BrainsStaking contract itself does not include a reward system. However, there seems to be a mechanism in place for distributing rewards using the Dopamine ($DPM) token based on the amounts staked in the BrainsStaking contract. A script for calculating airdrop was found within the protocol, suggesting that rewards are distributed later in the form of Dopamine tokens instead of embedding a reward system directly in the staking contract.

Dopamine.sol:
Another token contract implementing ERC20, ERC20Burnable, and Ownable features, with additional minting and burning controls. Supports batch transfers. The contract is upgradeable via UUPS. The owner has the unique capability to burn tokens from any account without needing approval.

IlliquidStake.sol:
An ERC721 contract representing non-transferable stakes in the BrainStarter ecosystem. Only mintable by managers and serves to lock tokens in a non-transferable state.

LiquidStake.sol:
An ERC721 contract representing transferable stakes., allowing trading of $BRAINS stakes within the ecosystem. Manages the minting and burning of these stakes.

UnlockFeeCalculator.sol:
This is a utility library for calculating unlock fees based on the type of lock and the time elapsed since staking. It handles different fee structures for various lock types, ensuring proper compliance with staking rules.

ROLES AND RESPONSIBILITIES

1. Brains.sol

Owner

• Initialize the contract and set key parameters such as the initial supply and yearly mint limit.
• Mint tokens.
• Authorize upgrades of the contract.

General Users

• Transfer tokens.
• Burn their tokens.

2. BrainsReceiptLocker.sol

Owner

• Initialize the contract with the necessary parameters and addresses.
• Authorize upgrades of the contract.

General Users

• Exchange their receipt tokens for $BRAINS and stake them.

3. BrainsStaking.sol

Owner

• Initialize the contract and set parameters like the staking token and thresholds.
• Authorize upgrades of the contract.
• Withdraw tokens from the contract.
• Set the liquid stake threshold.

Stakers

• Stake their $BRAINS tokens.
• Unstake their tokens (both liquid and illiquid stakes) according to the rules.
• Check their stake information and rewards.

4. Dopamine.sol

Owner

- Initialize the contract and set the initial owner.
- Mint and burn tokens.
- Authorize upgrades of the contract.
- Execute batch transfers.

General Users

- Transfer tokens.
- Burn their tokens.

5. IlliquidStake.sol

Manager

- Mint and burn illiquid stakes (NFTs) for specific addresses.

UPGRADER

- Authorize upgrades of the contract.

Stakers

- Hold non-transferable illiquid stakes.

Default Admin

- Initial role assignment and overall contract management.

6. LiquidStake.sol

Manager

- Mint and burn liquid stakes (NFTs).

UPGRADER

- Authorize upgrades of the contract.

Stakers

- Hold and transfer liquid stakes.

Default Admin

- Initial role assignment and overall contract management.

CONFIGURATION AND SETTINGS

1. Initial Supply and Minting Limits

- **Contracts: Brains.sol**
- **Importance:**

Initial Supply: Determines the starting amount of $BRAINS tokens in circulation, impacting the initial market dynamics and token distribution.
Yearly Minting Limits: Caps the number of new tokens minted yearly, preventing inflation and maintaining token value.

2. Stake Thresholds and Lock Types

- **Setting:** Minimum amount required for staking and the lock type associated with stakes (liquid or illiquid).
- **Contracts:**

BrainsStaking.sol:
The setLiquidStakeThreshold function sets the minimum staking amount for liquid stakes.
The stakeFor function manages the staking process, including lock types defined in the UnlockFeeCalculator library.
UnlockFeeCalculator.sol (Library):
Defines the lock types and associated unlock fees.

- **Importance:**

Stake Thresholds: Ensure that only committed users can participate in staking, providing stability to the staking pool.
Lock Types: Different lock types offer various levels of liquidity and reward structures.

LIST OF VALUABLE ASSETS

1. $BRAINS Token

• **Type:** ERC20

• **Description:** In the BrainsReceiptLocker.sol contract, users can exchange their receipt tokens, which represent entitlements from pre-sale, strategic, and seed sales, for $BRAINS tokens. This process integrates users into the ecosystem by converting their initial investments into the primary token. In the BrainsStaking.sol contract, $BRAINS tokens can be staked in both liquid and illiquid forms.

Location in Protocol:

• **BrainsReceiptLocker.sol:** Receives $BRAINS tokens in exchange for receipt tokens from various sales phases.

• **BrainsStaking.sol:** $BRAINS tokens are staked here, and the contract keeps track of the staked amounts. Additionally, $BRAINS tokens remain on the contract as unlock fees during unstake operations when a fee is applicable.

2. Dopamine (DPM) Token

• **Type:** ERC20

• **Description:** Another token within the ecosystem, similar in structure to $BRAINS, with functionalities for minting, burning, and batch transfers.

3. Illiquid Stakes

• **Type:** ERC721 (NFT)

• **Description:** Non-transferable NFT stakes representing locked $BRAINS tokens. These stakes are illiquid and cannot be traded, and they are designed to lock tokens for certain periods or conditions.

4. Liquid Stakes

• **Type:** ERC721 (NFT)

• **Description:** Transferable NFT stakes representing $BRAINS tokens that can be traded or transferred. They provide liquidity and flexibility within the ecosystem, allowing users to trade their stakes.

5. Receipt Tokens

• **Type:** ERC20 (Specific to pre-sale, strategic, and seed sales)

• **Description:** These tokens represent the entitlement to $BRAINS tokens obtained through various sale phases. Users exchange these tokens for actual $BRAINS tokens using the BrainsReceiptLocker contract.

Location in Protocol:

• **BrainsReceiptLocker.sol**: Holds receipt tokens and handles their exchange for $BRAINS tokens. It then stakes the $BRAINS tokens on behalf of the user.

Note: Contracts do not support any other assets except those listed above. Nevertheless, rescue functionality was added to the BrainsReceiptLocker contract.

DEPLOYMENT

1. deployBrains.ts

Description:

• Deploys the Brains token contract using a UUPS proxy.

• Initializes the contract with the deployer address, initial supply, and yearly mint limit from deploymentConfig.

• Saves the contract's ABI and address for future reference.

Critical Settings and Considerations:

• Initial Supply and Mint Limits: It's crucial to set these values accurately during deployment as they impact the initial token distribution and inflation control.

2. deployStakeNFTs.ts

Description:

• Deploys the IlliquidStake and LiquidStake contracts using UUPS proxies.

• Initializes both contracts with the deployer's address.

• Saves the contracts' ABI and address.

Critical Settings and Considerations:

• Role Assignments: Ensure roles like Manager and UPGRADER are properly assigned to manage staking operations and contract upgrades.

3. deployDopamine.ts

Description:

• Deploys the Dopamine token contract using a UUPS proxy.

• Initializes the contract with the deployer as the initial owner.

• Logs the deployment address.

4. deployStakingAndLocker.ts

Description:

• Deploys the BrainsStaking with references to the Brains, IlliquidStake, LiquidStake

• Grants Manager roles to the BrainsStaking for both the IlliquidStake and LiquidStake.

• Deploys the BrainsReceiptLocker contract, linking it with the BrainsStaking contract and token addresses from deploymentConfig.

Critical Settings and Considerations:

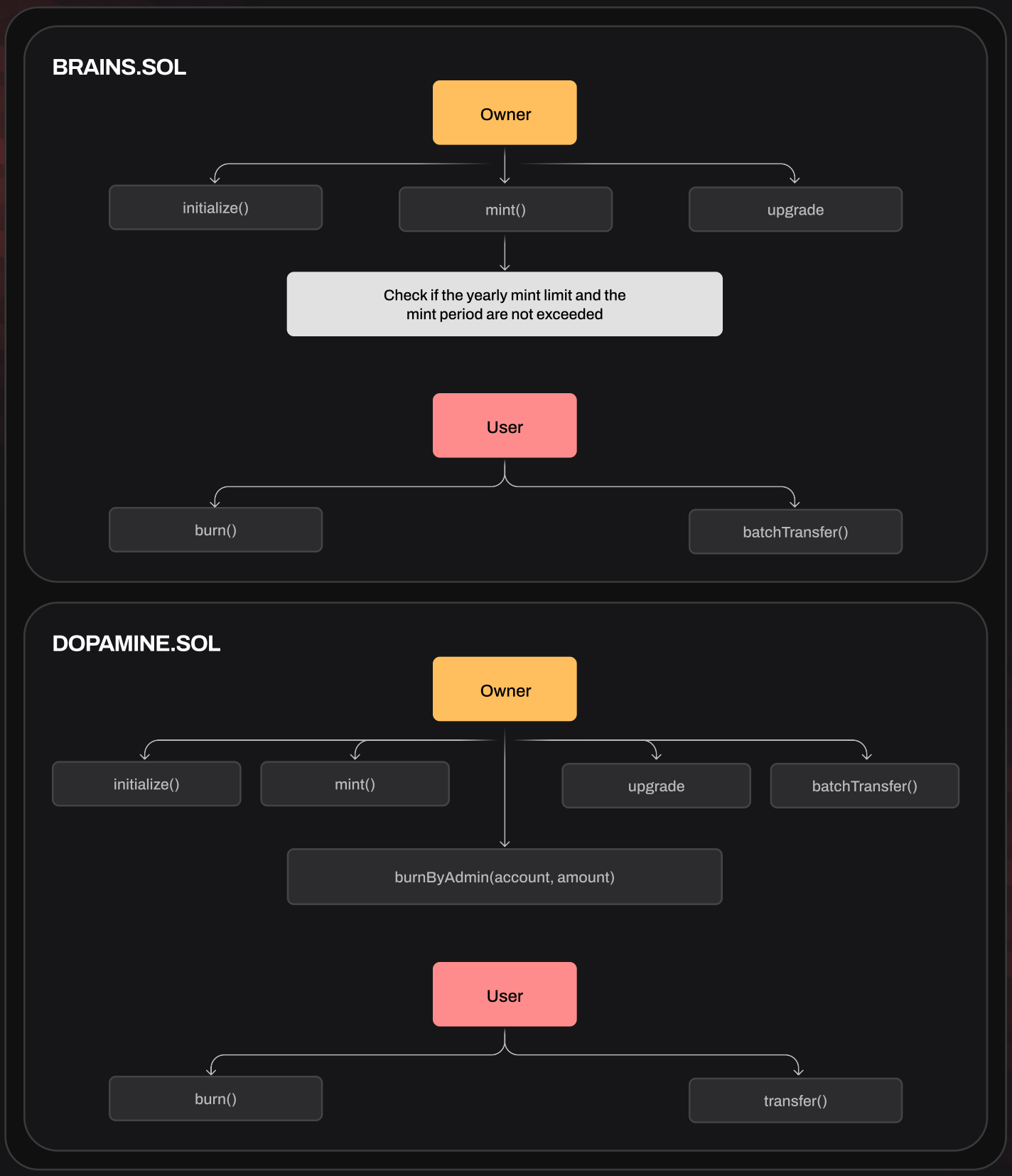
• setLiquidStakeThreshold: Set this parameter carefully to control liquidity in the system. If set to zero, it means all stakes are locked, which affects protocol's flexibility.

Good Practices in Using Upgrades

• Using upgrades: The upgrades.deployProxy function from the OpenZeppelin Hardhat Upgrades plugin simplifies the deployment process by handling the proxy setup and contract initialization. This method provides built-in safety checks and standards compliance, making it more secure and reliable compared to manual deployment.

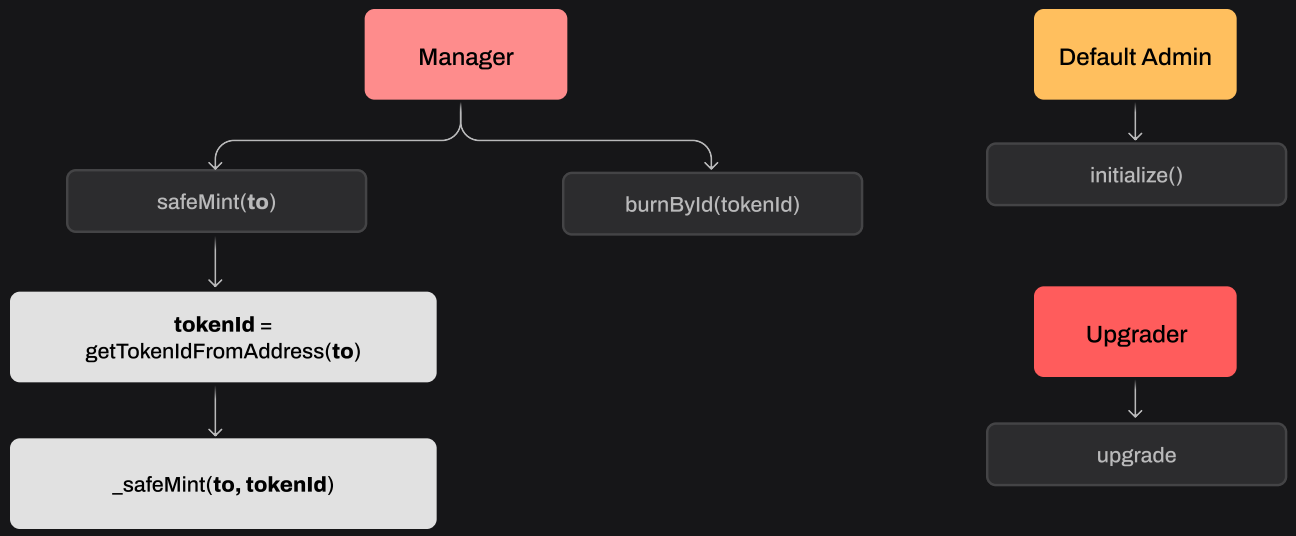
• Explicit Role Management: Always ensure that the UPGRADER role is properly managed. This role is critical for authorizing upgrades, and restricting it to trusted accounts or governance mechanisms is vital for protocol security.

ERC20

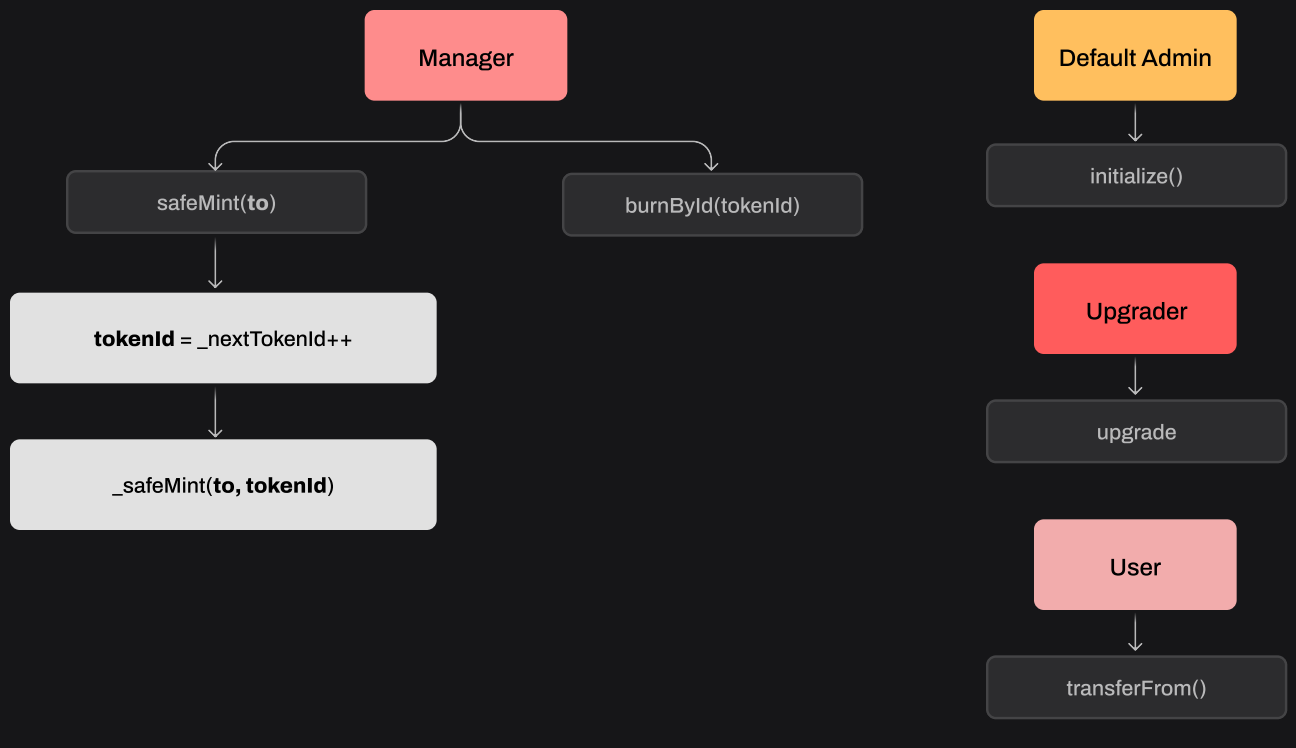

BRAINS.SOL
Owner
initialize()
mint()
upgrade
Check if the yearly mint limit and the mint period are not exceeded
User
burn()
batchTransfer()
DOPAMINE.SOL
Owner
initialize()
mint()
upgrade
batchTransfer()
burnByAdmin(account, amount)
User
burn()
transfer()

ERC721

ILLIQUIDSTAKE.SOL

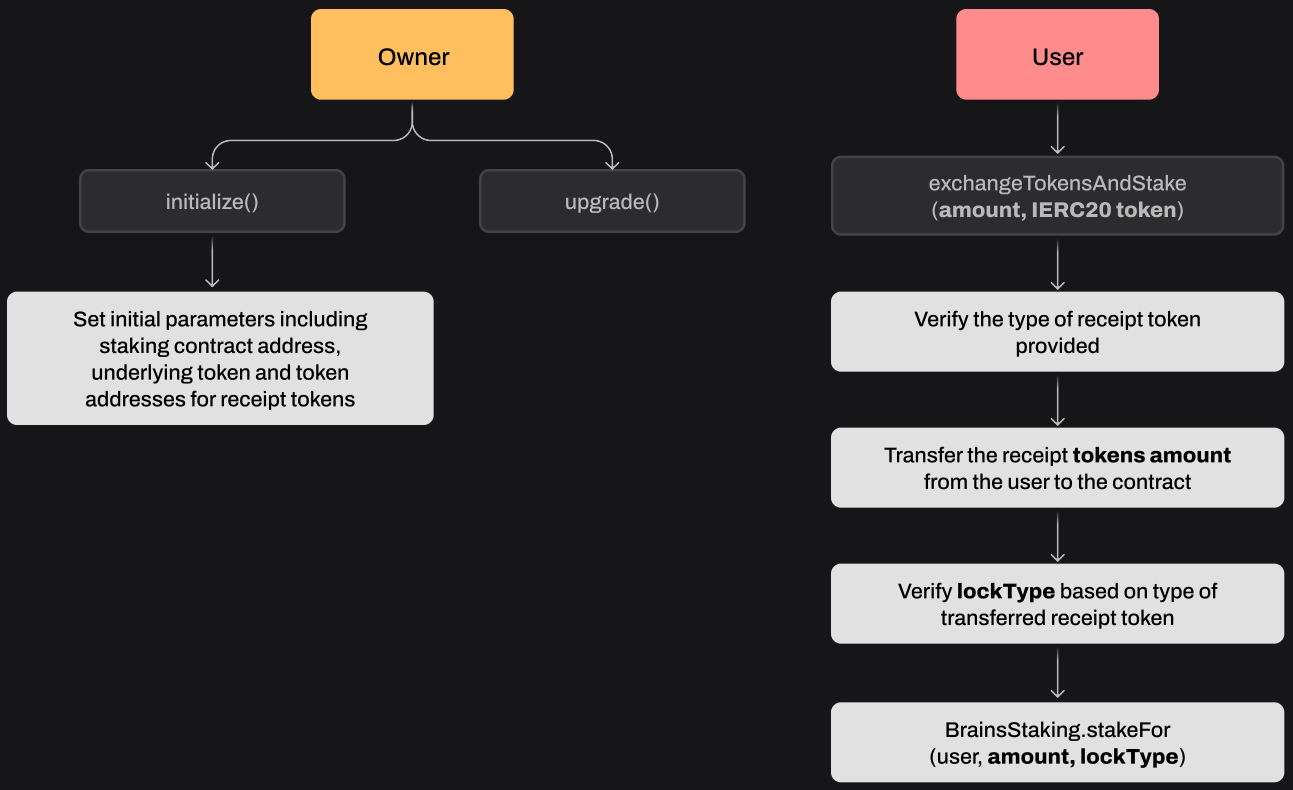


LIQUIDSTAKE.SOL

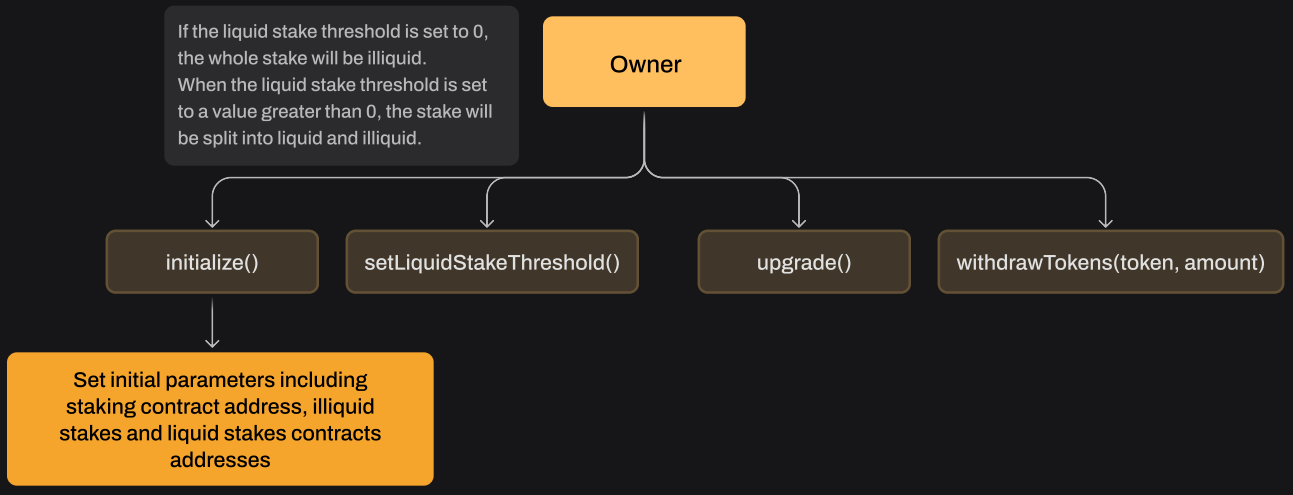


CONTRACTS

BRAINSRECEIPTLOCKER.SOL

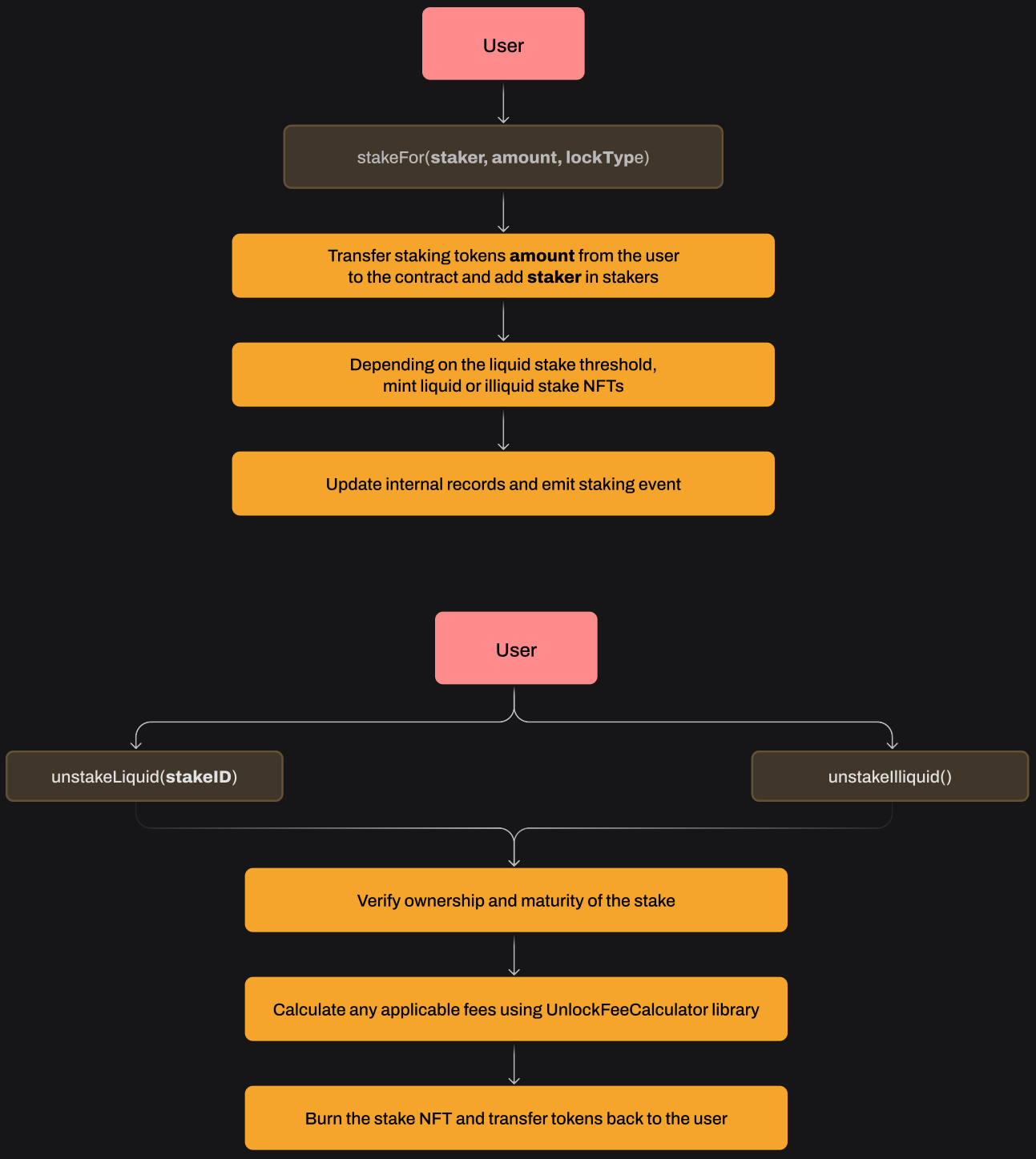


BRAINSTAKING.SOL



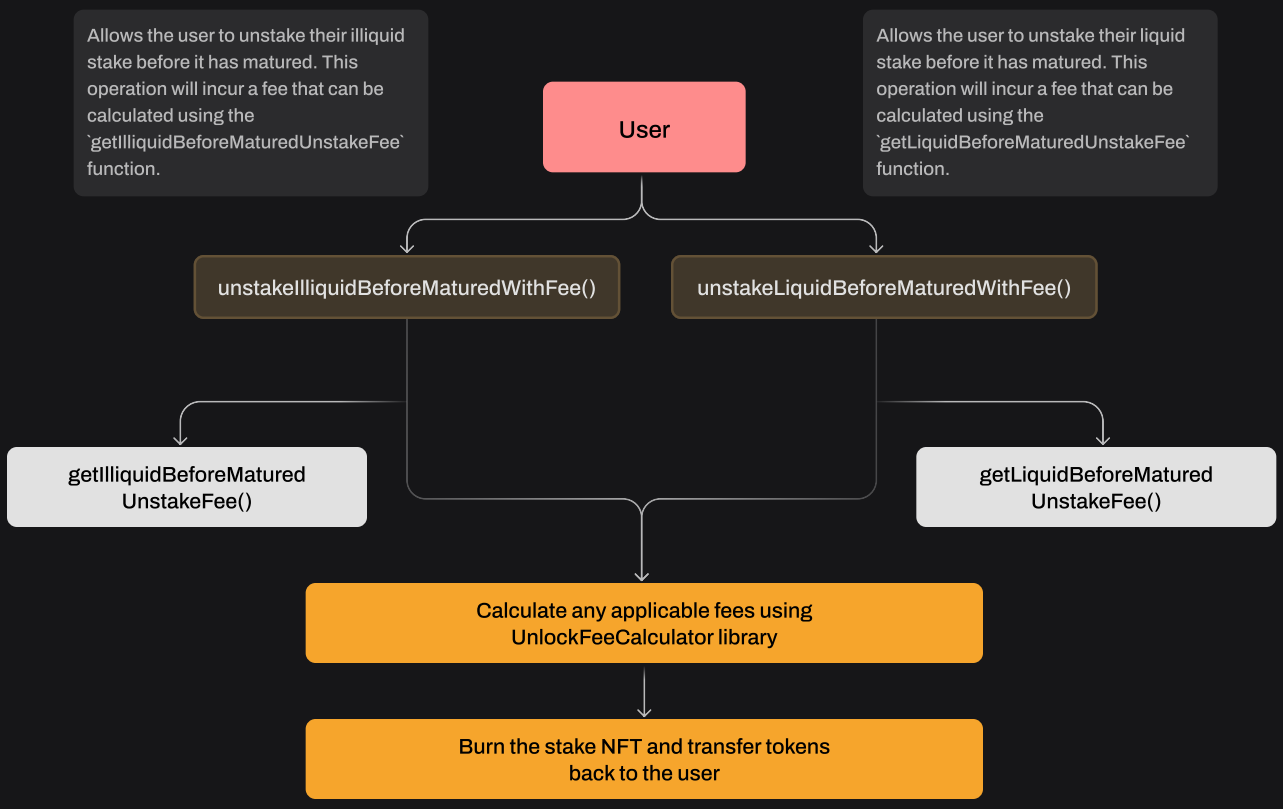
CONTRACTS

BRAINSRECEIPTLOCKER.SOL



CONTRACTS

BRAINSRECEIPTLOCKER.SOL



INTERACTION FLOW

Complete Analysis

STANDARD CHECKLIST / VULNERABLE AREAS

Storage structure and data modification flow	Pass
Access control structure, roles existing in the system	Pass
Public interface and restrictions based on the roles system	Pass
General Denial Of Service (DOS)	Pass
Entropy Illusion (Lack of Randomness)	N/A
Order-dependency and time-dependency of operations	Pass
Accuracy loss, incorrect math/formulas other violated operations with numbers	Pass
Validation of function parameters, inputs validation	Pass
Asset management, funds flow and assets conversions	Pass
Signatures replay and multisig schemes security	N/A
Asset Security (backdoors connected to underlying assets)	Fail
Incorrect minting, initial supply or other conditions for assets issuance	Pass
Global settings mis-using, incorrect default values	Pass
Violated communication between components/modules, broken co-dependencies	N/A
3rd party dependencies, used libraries and packages structure	N/A
Single point of failure	Pass
Centralization risk	Fail
General code structure checks and correspondence to best practices	Pass
Language-specific checks	Pass

DISCOVERED ISSUES

CRITICAL-1	Resolved

THE PREVIOUS STAKE AMOUNT MIGHT BE OVERWRITTEN.

BrainsStaking.sol: stakeFor(), lines 213-215.
In case liquidStakeThreshold equals 0, a special case in the code is invoked during the staking of tokens where an internal function _mintOrSetIlliquidStakeData() is executed, passing the passed parameter `_amount.` The function _mintOrSetIlliquidStakeData() is supposed to store the total staked amount of tokens staked as illiquid stakes. However, the passed amount won't be added to the value in `s.illiquidStakeIdToInfo[_staker].amount` but will overwrite it. As a result, users might lose their previous stakes if liquidStakeThreshold equals 0.

This issue is marked as Critical because the incorrect handling of staking amounts when liquidStakeThreshold equals 0 can result in users losing their previously staked tokens. Overwriting the stake amount instead of accumulating it could lead to a significant loss of staked tokens, impacting the users' balances and trust in the system.

RECOMMENDATION:

Pass `s.illiquidStakeIdToInfo[_staker].amount + _amount` when calling _mintOrSetIlliquidStakeData() for case when liquidStakeThreshold equals to 0.

POST-AUDIT.

Resolved with the code proposed in the recommendation.

CRITICAL-2 **Resolved**

LOSS OF ILLIQUID STAKES IS POSSIBLE.

BrainsStaking.sol: stakeFor(), lines 213-215.
BrainsStaking.sol: stakeFor(), lines 227-228.
When a user stakes tokens, part of them can become liquid and part of them illiquid. This ratio is defined by the parameter liquidStakeThreshold. One of the conditions in the function covers cases where tokens are divided into liquid and illiquid, and the user already has illiquid tokens stored in `s.illiquidStakeIdToInfo[_staker].amount.`

The new amount is calculated as `newTotalAmount = existingIlliquidStakeAmount + remainingAmountToAddToStake.`

If `newTotalAmount` is greater than liquidStakeThreshold, the function increases the number of liquid NFT tokens to mint by 1, and the remainder `newTotalAmount % s.liquidStakeThreshold` is written in the `s.illiquidStakeIdToInfo[_staker].amount`.
However, in case s.liquidStakeThreshold was changed between the user's previous stakes and current stake, adding only 1 token to the number of liquid NFT tokens might not be enough. As a result, the user will lose his tokens because they were subtracted from `s.illiquidStakeIdToInfo[_staker].amount,` and a sufficient number of liquid tokens weren't minted.

Consider the following example:
s.illiquidStakeIdToInfo[_staker].amount = 60.
liquidStakeThreshold was changed from 100 to 50, thus = 50.

1. The user stakes 140 tokens.
2. `howManyLiquidStakesToMint` = 140 / 50 = 2.
3. `newTotalAmount` = 60 + 40 = 100.
4. `s.illiquidStakeIdToInfo[_staker].amount` = 100 % 50 = 0.

As a result, the user's illiquid balance is equal to 0. However, only 1 liquid token was added, whereas he should have received 2 liquid tokens.

This issue is marked as Critical because changes in the liquidStakeThreshold parameter between staking operations can lead to an incorrect calculation of liquid and illiquid token balances. If the threshold changes, the logic for converting illiquid stakes to liquid tokens may not account for the new threshold correctly, causing users to lose tokens. This can result in discrepancies in token balances, undermining the integrity and reliability of the staking mechanism.

RECOMMENDATION:

Take into account possible changes of liquidStakeThreshold and recalculate the number of liquid tokens to stake in cases where `newTotalAmount` is greater than liquidStakeThreshold. For example, instead of
howManyLiquidStakesToMint++
the code can look like
howManyLiquidStakesToMint += newTotalAmount / s.liquidStakeThreshold.

POST-AUDIT.

Resolved with the code proposed in the recommendation.

CRITICAL-3 Resolved

LACK OF INFORMATION REMOVAL AFTER UNSTAKING.

BrainsStaking.sol: _unstakeLiquid(), _unstakeIlliquid().
The BrainsStaking contract does not appear to remove information about withdrawn stakes in the _unstakeLiquid() and _unstakeIlliquid() functions. Specifically, the amount in s.illiquidStakeIdToInfo[_staker].amount and in s.liquidStakeIdToInfo[liquidStakeIds[i]].amount is not cleared after an illiquid stake is withdrawn. This oversight could allow multiple withdrawals of the same stake.

This issue is marked as Critical because the potential for multiple withdrawals can lead to significant financial losses and exploitation of the staking system.

RECOMMENDATION:

Implement logic to remove or clear stake information after the stake has been successfully withdrawn to ensure accurate tracking of staked tokens and prevent multiple withdrawals.

POST-AUDIT.

Information removal now exists for both liquid and illiquid stakes.

HIGH-1 Resolved

POTENTIAL BYPASS OF COMMISSION SYSTEM.

BrainsStaking.sol.
The issue arises when a user takes the following steps:

1. The user stakes tokens via BrainsReceiptLocker with a lock type that requires a time-based lock period and associated fees. The staked amount in this case is less than the liquidStakeThreshold.
2. The user then stakes additional tokens directly via BrainsStaking using the Public lock type, which does not have any lock period or fees. The staked amount, in this case, is also less than the liquidStakeThreshold.
3. The user then can unstakes all their tokens via unstakeIlliquid(). Due to the mechanics of the BrainsStaking contract, the user can withdraw both the original locked stake and the newly staked tokens without incurring any fees, effectively bypassing the commission system.

Note: This issue is marked as high severity because it presents a significant risk to the integrity of the commission system. While it doesn't directly compromise the security of the contract (which would warrant a critical classification), it does allow users to exploit the system and avoid paying fees.

RECOMMENDATION:

Implement checks and restrictions to prevent this bypass strategy.

POST-AUDIT.

The necessary checks have been added to the function.

MEDIUM-1 Resolved

UNFINISHED TODO AND VERIFICATION OF FEE WITHDRAWAL LOGIC.

BrainsStaking.sol: withdrawTokens(), _unstakeIlliquid(), _unstakeLiquid().
The owner's withdraw function contains a TODO comment (line 308) regarding the restriction of withdrawing staking tokens, but this restriction has not been implemented. Also, it is unclear how the unlock fees, subtracted during unstake operations (lines 369, 399), will be withdrawn if such a restriction is applied.
Unfinished Implementation: Leaving TODO comments without implementation can lead to incomplete functionalities and unexpected behavior.
Ambiguity in Fee Management: The current implementation does not clarify how the unlock fees, subtracted during unstake operations, will be managed and withdrawn from the contract if the staking token withdrawal is restricted.

This issue is marked as Medium because commissions could potentially get stuck in the contract if a restriction is added. If there is no restriction, then the admin will have to calculate how much commissions he can withdraw.

RECOMMENDATION:

1. Verify Fee System: Confirm how the fee system is intended to work, especially in relation to the management and withdrawal of unlock fees.
2. Clarify Restriction Intent: Provide clarification on the intended restriction for withdrawing staking tokens. Explain why the restriction has not been implemented yet and whether it will be implemented in the future.
3. Update Documentation: Ensure that the documentation clearly outlines the process for managing and withdrawing unlock fees, including any restrictions.
E.g.: add the variable to track currently accumulated fees (unlockFee amounts deducted from unstakes), and withdraw fees in case the variable is not 0.

POST-AUDIT.

The Customer's team verified the implemented logic. The logic tracks how many tokens are stored in the contract as fees, and the contract allows the admin to withdraw only the collected fees, not tokens staked by users. For any other token, there are no restrictions on withdrawing (intended to be used as a rescue token function by admins).

LOW-1 Verified

VERIFICATION OF TOKEN PRESENCE IN THE CONTRACT.

BrainsReceiptLocker.sol: exchangeTokensAndStake().
The exchangeTokensAndStake function in the BrainsReceiptLocker contract allows users to exchange their receipt tokens for $BRAINS tokens and stake them immediately. However, the function does not mint $BRAINS tokens; instead, it approves the underlying tokens for the BrainsStaking contract. The assumption is that the underlying tokens will be transferred to BrainsStaking, where the actual staking occurs. The BrainsStaking contract then transfers the staking tokens from the sender, which in this case is BrainsReceiptLocker.
This chain of operations implies that the underlying tokens must already be present in the BrainsReceiptLocker contract before the function is called. The absence of verification steps or documentation regarding this requirement can lead to potential issues if the tokens are not present, resulting in failed transactions or unexpected behavior.

RECOMMENDATION:

1. Verify Token Presence: Add explicit verification steps to ensure the underlying tokens are present in the contract before proceeding with staking.
2. Add Documentation: Document the requirement for tokens to be present in the BrainsReceiptLocker contract before it starts functioning, so developers are aware of this prerequisite.

POST-AUDIT:

The Customer's team verified the logic and added a comment to the BrainsReceiptLocker that an admin has to initially send the tokens into that contract.

LOW-2 Resolved

CONFUSING STAKE ID LOGIC.

BrainsStaking.sol: _mintLiquidStakes().
In the _mintLiquidStakes() function, the s.illiquidStakeIdToInfo[_staker].stakeId is set to the ID of the last liquid stake minted for the user. This could be confusing since users can withdraw both liquid and illiquid tokens, leading to unclear logic for dApp integration.

RECOMMENDATION:

Simplify or clarify the logic in _mintLiquidStakes() by ensuring that s.illiquidStakeIdToInfo[_staker].stakeId is used consistently and does not mix liquid and illiquid stake IDs.

POST-AUDIT:

The assignment of stakeId was corrected for liquid stakes minting, as this was a typo.

LOW-3 Resolved

UNNECESSARY MINTING OF ILLIQUID STAKE.

BrainsStaking.sol: stakeFor().
In the stakeFor() function, an illiquid stake is minted even when the user only intends to stake liquid tokens. This occurs regardless of whether the remainingAmountToAddToStake is zero, leading to the creation of an unnecessary illiquid stake. While this does not critically impact the staking functionality, it can create confusion for users, especially when interacting with getters and dApps that differentiate between liquid and illiquid stakes.

RECOMMENDATION:

Adjust the logic to avoid minting an illiquid stake when the remainingAmountToAddToStake is zero. This will prevent unnecessary token minting and reduce potential confusion for users and developers working with the staking contract.

POST-AUDIT:

The project team added a check if remainingAmountToAddToStake is equal to 0, to short circuit to minting liquid stakes and skip locked (illiquid) stake logic.

LOW-4 Resolved

INCORRECT STAKER LIST MANAGEMENT ON LIQUID STAKE TRANSFER.

BrainsStaking.sol: stakeFor().
When a user stakes tokens and is added to the stakers list, they remain on this list even if they transfer their liquid stake to another user. If the new owner of the stake later unstakes the tokens, the original staker is not removed from the stakers list. This could cause confusion and potential issues, such as incorrect tracking of active stakers.
Note: The issue is marked as low severity because it does not directly impact the core functionality or security of the staking process. The primary risk is related to inaccurate tracking of the stakers list, which could lead to minor confusion or inefficiencies in off-chain or on-chain systems that rely on this list.

RECOMMENDATION:

Ensure that when a liquid stake is transferred and then unstaked by the new owner, the original staker is appropriately removed from the stakers list if they no longer hold any stakes.

POST-AUDIT:

The client's team has successfully implemented the recommended changes. The staker list management logic has been corrected and moved from BrainsStaking.sol to the appropriate LiquidStake.sol and LockedStake.sol contracts.

INFO-1 Verified

ADMIN CAN BURN TOKENS FROM ANY USER WITHOUT APPROVAL.

Dopamine.sol, LiquidStake.sol, IlliquidStake.sol.
The current implementation allows the admin (designated role for NFTs) to burn tokens and NFTs from any user without requiring approval from the user. This functionality provides the admin with significant control over the tokens, which could be used to enforce certain policies or penalties but also poses a risk of misuse. The ability to burn tokens without user approval can be seen as a centralization risk, as it gives the admin significant power over user assets.

The issue is marked as Info, as while it violates the user's balances, it can be part of the planned business logic (especially if the token is planned as pure utility). However, the issue may be reclassified if the token is used as an exchangeable value.

RECOMMENDATION:

Verify the intended functionality and provide clear documentation on the rationale behind this design decision. Implement safeguards or checks to prevent potential misuse of this capability.

POST-AUDIT:

The Customer verified that tokens are not supposed to be financial assets but purely internal utility assets.

INFO-2 Verified

UPGRADABLE CONTRACT AND POTENTIAL TOKEN WITHDRAWAL.

The BrainsReceiptLocker contract, which is responsible for locking a significant amount of tokens, currently lacks a function to withdraw receipt tokens. Given that the contract is upgradeable, there is a potential risk that an admin could update the contract to add a withdrawal function and subsequently withdraw the locked tokens.

RECOMMENDATION:

Confirm whether the tokens are intended to be locked permanently or if there should be a mechanism to withdraw them in the future. If maintaining upgradeability is necessary, implement safeguards to prevent unauthorized withdrawals. One approach could be to transfer tokens to a zero address if they are meant to be locked permanently.

POST-AUDIT:

The Customer verified that it is an intended logic, as the team will operate this contract in further development stages. However the security team still leaves the concern about clarity of the logic (clear marking that assets may be operated lately) and about the upgradeability of the contract in general.

INFO-3 Verified

NO RESTRICTIONS ON THE TOKEN BURNING.

Dopamine.sol, Brains.sol
The contracts inherit the default ERC20Burnable interface, which contains public burn() and burnFrom() functions.
While it is a regular functionality that allows users to burn tokens from their accounts, it still possesses several risks to the tokens economy:

- intentional decreasing of the supply by malicious actors
- secondary threats from the protocols holding the token - in case of their exploit
- secondary threats from burning of tokens via “dangling approves.”

The issue is marked as Info—while such functionality creates certain risks, it is a regular token extension. However, the security team requires additional verification from the Customer about the awareness of existing risks connected to this business logic decision.

RECOMMENDATION:

Verify the protocol's unrestricted burn() functionality is desired and potential risks are acknowledged.

POST-AUDIT:

The Customer’s team verified that they are aware of potential risks and find them acceptable. They confirmed that tokens burning is not financially feasible. Even if someone tries to buy as many tokens as possible and burn them, Tokenomics won't suffer.

INFO-4 Verified

UNRESTRICTED TOKEN MINTING.

Dopamine.sol.
The token implements the mint() interface controlled by the owner.
However, there are no restrictions on the minting process:

- the owner can mint any amount of tokens, at any moment in time, for any user
- there are no restrictions on the total supply of the token; thus, it can grow to any value

While the flexible supply and controlled minting are regular models, the absence of control measures creates several risks:

- centralization risk, as the Owner has unrestricted access to the token's supply
- secondary risks for the token economy in case of uncontrolled emission
- human factor risk - tokens can be minted to unvalidated addresses.

The issue is marked as Info—while such functionality creates certain risks, it is a regular token extension. However, the security team requires additional verification from the Customer about the awareness of existing risks connected to this business logic decision.

RECOMMENDATION:

1) Verify that the protocol's unrestricted minting functionality is desired and potential risks are acknowledged - risks of minting to wrong addresses, risks of minting incorrect amounts, risk of increasing the inflation speed, risk of causing price volatility after certain amounts minting, etc.
2) Provide the sanitizing measures for the minting process: validation of the Owner's assigned, validation of minted token receivers, validation of the minting schedule, control over the emission rate, and supply growth speed.
3) The better way is to encode restrictions into the smart contract: limits on one-time minting, supply cap or restrictions on the emission rate, and restrictions on the receiving addresses. Though the auditors recognize that it will require token re-deployment, these recommendations are targeted at the Customer's awareness of potential risks.

POST-AUDIT.

The Customer verified that this is a desired behavior, as admin is supposed to arbitrarily mint or burn dopamine from any wallet. Dopamine is not a financial asset, it's just a marker that will be used in off chain computation. It was a business requirement to make it an erc20 token so it can be transparently tracked on chain.

INFO-5 Verified

FUNCTION PARAMETER MISUSE RISK.

BrainsStaking.sol, BrainsReceiptLocker.sol.
In the stakeFor() function, the description for the _lockType parameter suggests that users should use the Public type since it has no lock period or fees, and other types will be handled through BrainsReceiptLocker. However, a user might mistakenly specify a different type. Additionally, the purpose of the Founder type is unclear if users can simply use Public.
The issue is marked as Info, as while regular users will not benefit from choosing another option, the tokensale participants may have such an option. However, since the presale contract and logic are out of the scope of the audit, auditors require comments from the project's team. Thus, verification is required so that tokensale participants will not be able to use the contract directly - otherwise, the criticality of the issue will be increased.

RECOMMENDATION:

It is recommended to add the following checks:

1. Users directly calling the staking function can only specify Public.
2. BrainsReceiptLocker can only specify PreSale, StrategicOrPrivate, or Seed.
3. Ensure that founders can only stake with the Founder type (e.g., maintain a list of founders on the contract and enforce that users in this list must use the Founder type).

POST-AUDIT:

The Customer acknowledged potential risks and verified that users will only use these contracts through a frontend hosted by the team. If someone tries to use contracts directly through the blockchain, they do so on their own responsibility.

INFO-6 Verified

ABSENT URI ADDRESS FOR NFTS.

LiquidStake.sol, IlliquidStake.sol
Both NFTs implement the default ERC721 functionality inherited from the OpenZeppelin. However, the default implementation expects the overload of the _baseUri() and/or tokenUri() methods, as by default, the NFTs uri is empty and contains only NFT id.
Since both liquid and illiquid NFTs have consequent IDs starting from 0, they cannot be distinguished without URIs, thus there is no place for metadata.

However, auditors recognize that these are utility NFTs and may not require the metadata features. Though it will be a substandard behavior, it requires confirmation from the team.

RECOMMENDATION:

Verify that liquid and illiquid utility NFTs require no metadata features so the URI can be empty or consist of only the token ID. Or override _baseUri()/tokenUri() methods and set the URI during the initialization.

POST-AUDIT:

The Customer's team verified that since NFTs are meant to serve for utility purposes only, there is no need for metadata.

Unresolved

BEST PRACTICES AND CODE STYLE VIOLATIONS, OPTIMIZATIONS

1) Inconsistent Use of Require-Revert and If-Revert Patterns.
The protocol employs different styles for error handling: some contracts use require statements with an error instead of string messages for validation, while others use if statements followed by a reverting call. Although both approaches are valid and can be used, this inconsistency can lead to confusion and make the codebase harder to read and maintain.

Adopt a single style for error handling across the entire codebase.

2) Redundant Public Burn Function.
Brains.sol.
The contract defines a burn function that triggers the internal _burn() to remove tokens from the sender's balance. However, this function is redundant because the contract inherits from ERC20Burnable, which already provides a public burn function. Having redundant functions increases the code size unnecessarily.

Remove the redundant burn function from the contract, as ERC20Burnable already provides the functionality.

3) Redundant imports.
UnlockFeeCalculator.sol.
The contract includes an import statement for hardhat/console.sol, which is used for testing and debugging purposes. This import should be removed before deploying the contract to the mainnet to ensure the production code is clean and free from unnecessary dependencies.

4) Use of Magic Numbers.
UnlockFeeCalculator.sol: getUnlockFeeAmount().
The function uses several magic numbers without explanations or descriptive variable names. Magic numbers can make the code difficult to read, understand, and maintain, as the purpose of these numbers is not immediately clear.

Replace magic numbers with named constants or variables, and provide explanations or comments to clarify their purpose.

5) Missing Events for State Changes
BrainsStaking.sol: setLiquidStakeThreshold();
BrainsReceiptLocker.sol: exchangeTokensAndStake()
Certain state-changing functions in the contract do not emit events to log the changes. Events are crucial for tracking state changes on the blockchain, providing transparency, and allowing off-chain applications to respond to these changes.
Note: While the stakeFor function includes events that track staking, these events do not indicate that the staking occurred as a result of a token exchange from the exchangeTokensAndStake function. That's why exchangeTokensAndStake needs additional events.

Add events to log state changes.

6) Lack of Validation.
6.1) Brains.sol: initialize().
In the initialize function, the _yearlyMintLimit parameter is assigned to the main storage without any validation. This could potentially lead to an incorrect or undesirable mint limit being set, which might impact the functionality and security of the contract.

Add appropriate validation checks for _yearlyMintLimit during the initialization process. This could include checks to ensure the value is within a reasonable range or meets certain criteria.

6.2) IlliquidStake.sol, LiquidStake.sol: initialize().
The initialize function grants the DEFAULT_ADMIN_ROLE to the provided defaultAdmin address without any validation.

Implement validation checks for the defaultAdmin address in the initialize function to ensure it is not a zero address.

6.3) BrainsStaking.sol: setLiquidStakeThreshold().
The setLiquidStakeThreshold() function lacks validation for the _threshold parameter.

Adding validation checks will ensure that the parameter meets the expected criteria and prevent potential issues from invalid values.

7) Optimization Opportunity.
BrainsStaking.sol: getUserTotalStakedAmount().
The function retrieves the tokenId variable within a loop inside the getter function. If this getter is called by other contracts, it would be more gas-efficient to move the tokenId variable declaration outside of the loop. This optimization reduces gas usage and enhances performance when the function is called multiple times, particularly by other contracts.

8) Inconsistent Return Values in Getter Functions.
BrainsStaking.sol: getIlliquidStakeInfo(), getLiquidStakeInfo().
The functions have different return values, leading to inconsistencies in how stake information is retrieved. The first getter does not return stakeType and lockType, while the second getter includes these fields. Additionally, the second getter does not handle cases where the amount is zero or returns an `exists` flag like the first getter. This inconsistency can lead to confusion and inefficient use of the functions.

Considering modifying the getIlliquidStakeInfo() function to also return stakeType and lockType for consistency with getLiquidStakeInfo() and adjust the getLiquidStakeInfo() function to return an exists flag and handle cases where the amount is zero by returning all zero values, similar to how getIlliquidStakeInfo() handles it.

RECOMMENDATION:

Consider correcting all listed issues to increase code readability and logic optimization.

POST-AUDIT:

Fixed Issues:

1. Redundant Burn Function: Removed in Brains.sol.
2. Redundant Imports: Cleaned up in UnlockFeeCalculator.sol.
3. Missing Events: Added in BrainsStaking.sol and BrainsReceiptLocker.sol.
4. Inconsistent Return Values: Standardized in BrainsStaking.sol.

Unresolved Issues:

Inconsistent Error Handling, Lack of Validation, Optimization Opportunity.

Magic Numbers: UnlockFeeCalculator.sol still uses the magic number 100; wasn't replaced with a named constant.

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

Brains

Deployment

- ✓ Should init correctly (305ms)

Upgrading

- ✓ Should upgrade the contract
- ✓ Should upgrade the contract only by owner

Operations

- ✓ Should allow everyone to burn tokens
- ✓ Should allow batch transfer
- ✓ Shouldn`t allow batch transfer with invalid arrays length
- ✓ Should only allow the owner to mint tokens
- ✓ Should not allow minting after 5 years since deployment
- ✓ Should not allow minting more than the yearly limit
- ✓ Should allow minting up to the yearly limit
- ✓ Should allow minting next year after the first year`s limit is reached
- ✓ Should not allow minting more than the yearly limit

BrainsReceiptLocker

Exchange and stake

- ✓ Should exchange tokens and stake (93ms)

BrainsStaking

Illiquid stakes

- ✓ Should allow to stake tokens (for other address)
- ✓ Should allow to unstake tokens staked in illiquid type stake
- ✓ Shouldn`t allow to burn and mint illiquid stakes by everyone
- ✓ Shouldn`t allow to transfer illiquid stakes
- ✓ Should allow to burn illiquid stakes by manager
- ✓ Should not allow to unstake tokens staked in illiquid type stake when invested in StrategicOrPrivate round because of months difference
- ✓ Should not allow to unstake tokens staked in illiquid type stake when invested in Seed round because of months difference
- ✓ Should not allow to unstake tokens staked in illiquid type stake when invested in Founder round because of months difference
- ✓ Should only allow to unstake tokens with fee when staked in illiquid type stake and invested in 1 round (44ms)

Liquid stakes

- ✓ Should allow to burn liquid stakes by manager
- ✓ Should not allow to unstake tokens staked in liquid type stake because of months difference
- ✓ Shouldn`t allow to mint and burn liquid tokens by everyone
- ✓ If staked more than threshold, should create illiquid stake with the remainder
- ✓ Should update the illiquid stake if staked less than threshold and adds a new stake still below threshold
- ✓ When user already has existing illiquid stake and now stakes more than threshold, should create new liquid stakes and should keep the illiquid stake if there is remainder (41ms)
- ✓ Should allow to stake tokens when liquid threshold is set

- ✓ When user already has existing illiquid stake and now stakes more than threshold, should create new liquid stakes and burn the illiquid stake if there is no remainder (52ms)
- ✓ Should allow to unstake liquid before matured with fee

Operations

- ✓ Should withdraw different tokens from staking by owner (38ms)

Scenarios

- ✓ Unstake flow (90ms)
- ✓ Shouldn`t mint illiquid stake by default (53ms)
- ✓ Staking with threshold changes to bigger value (71ms)
- ✓ Staking with threshold changes to smaller value (liquid) (65ms)
- ✓ Staking with threshold changes to smaller value (illiquid) (60ms)
- ✓ Lock type staking check (illiquid, threshold 0)
- ✓ Lock type staking check (liquid) (60ms)
- ✓ Flow with 3 users (82ms)

Dopamine

Deployment

- ✓ Should init correctly

Operations

- ✓ Should allow everyone to burn tokens
- ✓ Should allow owner burn tokens for everyone by burnByAdmin
- ✓ Should allow batch transfer
- ✓ Shouldn`t allow batch transfer with invalid arrays length
- ✓ Should only allow the owner to mint tokens
- ✓ Should only allow the owner to transfer tokens

UnlockFeeCalculator

Seed

- ✓ Should not allow to withdraw at all for first 12 months
- ✓ Should allow to withdraw after 12 months with a fee (57ms)
- ✓ Should allow to withdraw after 25 months without the fee

Strategic/Private

- ✓ Should not allow to withdraw at all for first 12 months
- ✓ Should allow to withdraw after 12 months but with a fee
- ✓ Should allow to withdraw after 25 months without the fee

Founder

- ✓ Should not allow to withdraw for first 24 months

PreSale

- ✓ Should allow to withdraw after 1 months with 85% fee
- ✓ Should allow to withdraw after 2 months with 85% fee
- ✓ Should allow to withdraw after 3 months with 80% fee
- ✓ Should allow to withdraw after 5 months with 80% fee
- ✓ Should allow to withdraw after 6 months with 75% fee
- ✓ Should allow to withdraw after 8 months with 75% fee
- ✓ Should allow to withdraw after 9 months with 70% fee
- ✓ Should allow to withdraw after 11 months with 70% fee
- ✓ Should allow to withdraw after 12 months with 65% fee
- ✓ Should allow to withdraw after 13 months with 60% fee
- ✓ Should allow to withdraw after 14 months with 55% fee

- ✓ Should allow to withdraw after 15 months with 50% fee
- ✓ Should allow to withdraw after 16 months with 45% fee
- ✓ Should allow to withdraw after 17 months with 40% fee
- ✓ Should allow to withdraw after 18 months with 35% fee
- ✓ Should allow to withdraw after 19 months with 30% fee
- ✓ Should allow to withdraw after 20 months with 25% fee
- ✓ Should allow to withdraw after 21 months with 20% fee
- ✓ Should allow to withdraw after 22 months with 15% fee
- ✓ Should allow to withdraw after 23 months with 10% fee
- ✓ Should allow to withdraw after 24 months with 5% fee
- ✓ Should allow to withdraw after 25 months with 0% fee
- ✓ Should allow to withdraw but with a fee
- ✓ Should allow to withdraw without a fee when matured

RESULTING COVERAGE

FILE	% STMTS	% BRANCH	% FUNCS
Brains.sol	100	91.67	100
BrainsReceiptLocker.sol	100	78.57	80
BrainsStaking.sol	98.81	92.5	96.15
Dopamine.sol	100	83.33	85.71
LiquidStake.sol	85.71	62.5	62.5
LockedStake.sol	87.5	70	66.67
UnlockFeeCalculator.sol	95.83	95.45	100

NATIVE TESTS OVERVIEW

The BrainStarter team supplied native tests for the audited contracts. While tests have good quality, the test coverage is partial with some contracts on 85%+ or even 63%+, leaving certain areas of the provided contracts uncovered. It is highly recommended to support the test coverage on 90%+ level, aiming the industry standard of 95%. It is also recommended to have not only unit tests, but main scenarios and edge-cases as well.

Disclaimer

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.