Smart Contract Audit Report for Possum

Testers

1. Or Duan
2. Avigdor Sason Cohen

Table of Contents

Management Summary

Possum contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for Possum's smart contracts.

Over the research period of 40 research hours, we discovered 7 vulnerabilities in the contract.

After a review by the Sayfer team, we certify that all the security issues mentioned in this report have been addressed or acknowledged by the Possum team.

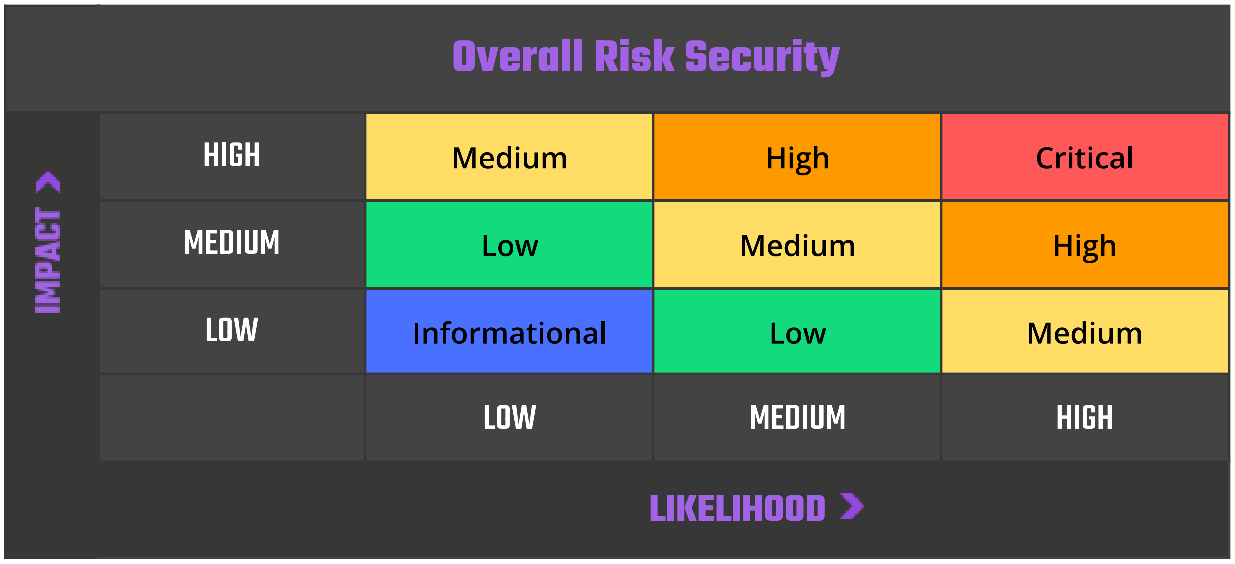
Risk Methodology

At Sayfer, we are committed to delivering the highest quality smart contract audits to our clients. That's why we have implemented a comprehensive risk assessment model to evaluate the severity of our findings and provide our clients with the best possible recommendations for mitigation.

Our risk assessment model is based on two key factors: **IMPACT** and **LIKELIHOOD**. Impact refers to the potential harm that could result from an issue, such as financial loss, reputational damage, or a non-operational system. Likelihood refers to the probability that an issue will occur, taking into account factors such as the complexity of the contract and the number of potential attackers.

By combining these two factors, we can create a comprehensive understanding of the risk posed by a particular issue and provide our clients with a clear and actionable assessment of the severity of the issue. This approach allows us to prioritize our recommendations and ensure that our clients receive the best possible advice on how to protect their smart contracts.

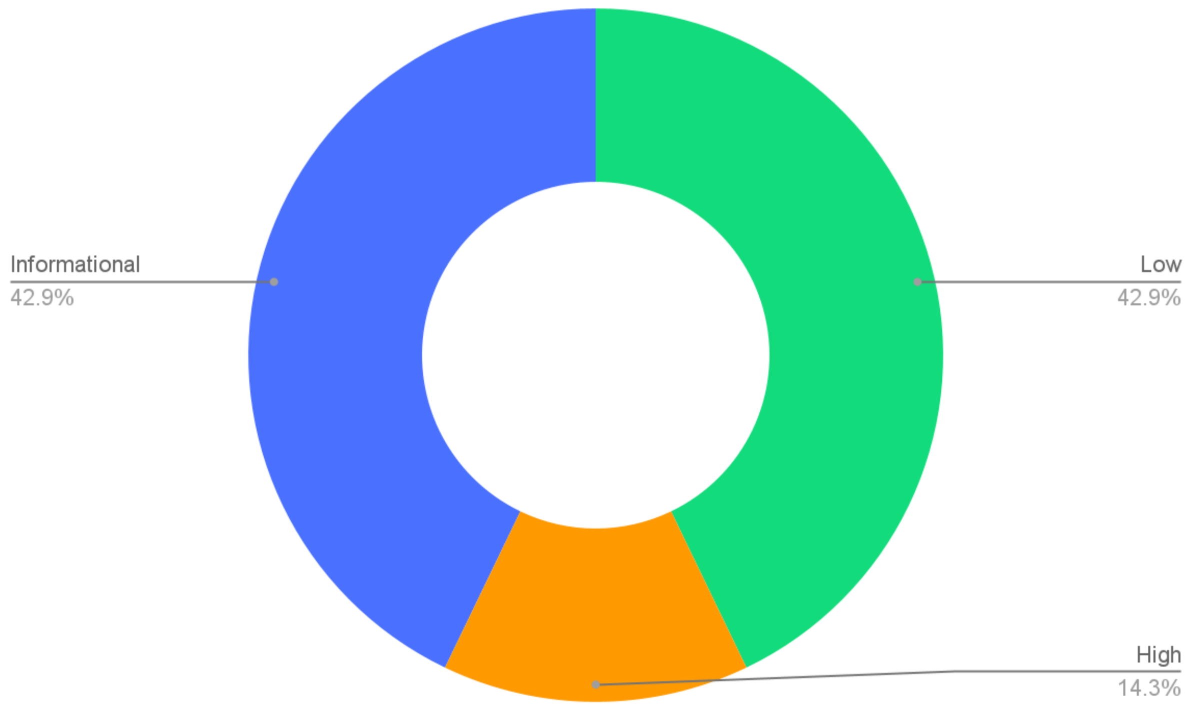
Risk is defined as follows:



Overall Risk Security

IMPACT				
	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Informational	Low	Medium
		LOW	MEDIUM	HIGH
			LIKELIHOOD	

Vulnerabilities by Risk



Risk	Low	Medium	High	Critical	Informational
# of issues	3	0	1	0	3

Approach

Introduction

Possum contacted Sayfer to perform a security audit on their smart contracts.

This report documents the research carried out by Sayfer targeting the selected resources defined under the research scope. Particularly, this report displays the security posture review for the aforementioned contracts.

Scope Overview

Together with the client team we defined the following contract as the scope of the project. Commit hash:

Contract	Sha-256
`MintBurnToken.sol`	`5dfa18f86d7f1e431e7e881d785aac6b69f58d3538f508701112f9b15aa55f21`
`Portal.sol`	`1947471ac662a38490be080aebe2ce25e86582fa0684a3d56a8e6c9848c217e5`
`Possum.sol`	`0270a5a56adf3a254ddf7cfa9e8642fdcb8b06367258262179e2ce5218dd909f`

Our tests were performed between October to November 2023.

Scope Validation

We began by ensuring that the scope defined to us by the client was technically logical. Deciding what scope is right for a given system is part of the initial discussion.

Threat Model

We defined that the largest current threat to the system is the ability of malicious users to steal funds from the contract.

Protocol Overview

Protocol Introduction

Possum Labs is a novel DeFi concept focused on building a diverse range of self-regulating financial products. It is named 'Possum' to reflect the goal of creating a system where everyone is fairly rewarded. Possum Labs vision is centered on addressing longstanding issues in the current financial system, particularly the excessive control wielded by a few major players over global assets. This dominance often leads to misaligned incentives, like asset managers prioritizing their own fees over their clients' investments. To make this vision a reality, Possum Labs is developing innovative financial tools, all governed by an on-chain system. The first of these products is called Possum Portals.

Security Evaluation

The following test cases were the guideline while auditing the system. This checklist is a modified version of the SCSVS v1.2, with improved grammar, clarity, conciseness, and additional criteria. Where there is a gap in the numbering, an original criterion was removed. Criteria that are marked with an asterisk were added by us.

Architecture, Design and Threat Modeling	Test Name
G1.2	Every introduced design change is preceded by threat modeling.
G1.3	The documentation clearly and precisely defines all trust boundaries in the contract (trusted relations with other contracts and significant data flows).
G1.4	The SCSVS, security requirements or policy is available to all developers and testers.
G1.5	The events for the (state changing/crucial for business) operations are defined.
G1.6	The project includes a mechanism that can temporarily stop sensitive functionalities in case of an attack. This mechanism should not block users' access to their assets (e.g. tokens).
G1.7	The amount of unused cryptocurrencies kept on the contract is controlled and at the minimum acceptable level so as not to become a potential target of an attack.
G1.8	If the fallback function can be called by anyone, it is included in the threat model.
G1.9	Business logic is consistent. Important changes in the logic should be applied in all contracts.
G1.10	Automatic code analysis tools are employed to detect vulnerabilities.
G1.11	The latest major release of Solidity is used.
G1.12	When using an external implementation of a contract, the most recent version is used.
G1.13	When functions are overridden to extend functionality, the super keyword is used to maintain previous functionality.
G1.14	The order of inheritance is carefully specified.
G1.15	There is a component that monitors contract activity using events.
G1.16	The threat model includes whale transactions.
G1.17	The leakage of one private key does not compromise the security of the entire project.

Policies and Procedures	Test Name

G2.2	The system's security is under constant monitoring (e.g. the expected level of funds).
G2.3	There is a policy to track new security vulnerabilities and to update libraries to the latest secure version.
G2.4	The security department can be publicly contacted and that the procedure for handling reported bugs (e.g., thorough bug bounty) is well-defined.
G2.5	The process of adding new components to the system is well defined.
G2.6	The process of major system changes involves threat modeling by an external company.
G2.7	The process of adding and updating components to the system includes a security audit by an external company.
G2.8	In the event of a hack, there's a clear and well known mitigation procedure in place.
G2.9	The procedure in the event of a hack clearly defines which persons are to execute the required actions.
G2.10	The procedure includes alarming other projects about the hack through trusted channels.
G2.11	A private key leak mitigation procedure is defined.

Upgradability	**Test Name**
G2.2	Before upgrading, an emulation is made in a fork of the main network and everything works as expected on the local copy.
G2.3	The upgrade process is executed by a multisig contract where more than one person must approve the operation.
G2.4	Timelocks are used for important operations so that the users have time to observe upcoming changes (please note that removing potential vulnerabilities in this case may be more difficult).
G2.5	*initialize()* can only be called once.
G2.6	*initialize()* can only be called by an authorized role through appropriate modifiers (e.g. *initializer*, *onlyOwner*).
G2.7	The update process is done in a single transaction so that no one can front-run it.
G2.8	Upgradeable contracts have reserved gap on slots to prevent overwriting.
G2.9	The number of reserved (as a gap) slots has been reduced appropriately if new variables have been added.
G2.10	There are no changes in the order in which the contract state variables are declared, nor their types.
G2.11	New values returned by the functions are the same as in previous versions of the contract (e.g. *owner()*, *balanceOf(address)*).
G2.12	The implementation is initialized.
G2.13	The implementation can't be destroyed.

Business Logic	Test Name
G4.2	The contract logic and protocol parameters implementation corresponds to the documentation.
G4.3	The business logic proceeds in a sequential step order and it is not possible to skip steps or to do it in a different order than designed.
G4.4	The contract has correctly enforced business limits.
G4.5	The business logic does not rely on the values retrieved from untrusted contracts (especially when there are multiple calls to the same contract in a single flow).
G4.6	The business logic does not rely on the contract’s balance (e.g., *balance == 0*).
G4.7	Sensitive operations do not depend on block data (e.g., *block hash*, *timestamp*).
G4.8	The contract uses mechanisms that mitigate transaction-ordering (front-running) attacks (e.g. pre-commit schemes).
G4.9	The contract does not send funds automatically, but lets users withdraw funds in separate transactions instead.

Access Control	Test Name
G5.2	The principle of the least privilege is upheld. Other contracts should only be able to access functions and data for which they possess specific authorization.
G5.3	New contracts with access to the audited contract adhere to the principle of minimum rights by default. Contracts should have a minimal or no permissions until access to the new features is explicitly granted.
G5.4	The creator of the contract complies with the principle of the least privilege and their rights strictly follow those outlined in the documentation.
G5.5	The contract enforces the access control rules specified in a trusted contract, especially if the dApp client-side access control is present and could be bypassed.
G5.6	Calls to external contracts are only allowed if necessary.
G5.7	Modifier code is clear and simple. The logic should not contain external calls to untrusted contracts.
G5.8	All user and data attributes used by access controls are kept in trusted contracts and cannot be manipulated by other contracts unless specifically authorized.
G5.9	the access controls fail securely, including when a revert occurs.
G5.10	If the input (function parameters) is validated, the positive validation approach (whitelisting) is used where possible.

Communication	Test Name
G6.2	Libraries that are not part of the application (but the smart contract relies on to operate) are identified.

G6.3	Delegate call is not used with untrusted contracts.
G6.4	Third party contracts do not shadow special functions (e.g. revert).
G6.5	The contract does not check whether the address is a contract using *extcodesize* opcode.
G6.6	Re-entrancy attacks are mitigated by blocking recursive calls from other contracts and following the Check-Effects-Interactions pattern. Do not use the *send* function unless it is a must.
G6.7	The result of low-level function calls (e.g. *send*, *delegatecall*, *call*) from other contracts is checked.
G6.8	Contract relies on the data provided by the right sender and does not rely on tx.origin value.

Arithmetic	Test Name
G7.2	The values and math operations are resistant to integer overflows. Use SafeMath library for arithmetic operations before solidity 0.8.*.
G7.3	the unchecked code snippets from Solidity ≥ 0.8.* do not introduce integer under/overflows.
G7.4	Extreme values (e.g. maximum and minimum values of the variable type) are considered and do not change the logic flow of the contract.
G7.5	Non-strict inequality is used for balance equality.
G7.6	Correct orders of magnitude are used in the calculations.
G7.7	In calculations, multiplication is performed before division for accuracy.
G7.8	The contract does not assume fixed-point precision and uses a multiplier or store both the numerator and denominator.

Denial of Service	Test Name
G8.2	The contract does not iterate over unbound loops.
G8.3	Self-destruct functionality is used only if necessary. If it is included in the contract, it should be clearly described in the documentation.
G8.4	The business logic isn't blocked if an actor (e.g. contract, account, oracle) is absent.
G8.5	The business logic does not disincentivize users to use contracts (e.g. the cost of transaction is higher than the profit).
G8.6	Expressions of functions assert or require have a passing variant.
G8.7	If the fallback function is not callable by anyone, it is not blocking contract functionalities.
G8.8	There are no costly operations in a loop.
G8.9	There are no calls to untrusted contracts in a loop.
G8.10	If there is a possibility of suspending the operation of the contract, it is also

	possible to resume it.
G8.11	If whitelists and blacklists are used, they do not interfere with normal operation of the system.
G8.12	There is no DoS caused by overflows and underflows.

Blockchain Data	Test Name
G9.2	Any saved data in contracts is not considered secure or private (even private variables).
G9.3	No confidential data is stored in the blockchain (passwords, personal data, token etc.).
G9.4	Contracts do not use string literals as keys for mappings. Global constants are used instead to prevent Homoglyph attack.
G9.5	Contract does not trivially generate pseudorandom numbers based on the information from blockchain (e.g. seeding with the block number).

Gas Usage and Limitations	Test Name
G10.2	Gas usage is anticipated, defined and has clear limitations that cannot be exceeded. Both code structure and malicious input should not cause gas exhaustion.
G10.3	Function execution and functionality does not depend on hard-coded gas fees (they are bound to vary).

Clarity and Readability	Test Name
G11.2	The logic is clear and modularized in multiple simple contracts and functions.
G11.3	Each contract has a short 1-2 sentence comment that explains its purpose and functionality.
G11.4	Off-the-shelf implementations are used, this is made clear in comment. If these implementations have been modified, the modifications are noted throughout the contract.
G11.5	The inheritance order is taken into account in contracts that use multiple inheritance and shadow functions.
G11.6	Where possible, contracts use existing tested code (e.g. token contracts or mechanisms like *ownable*) instead of implementing their own.
G11.7	Consistent naming patterns are followed throughout the project.
G11.8	Variables have distinctive names.
G11.9	All storage variables are initialized.
G11.10	Functions with specified return type return a value of that type.

G11.11	All functions and variables are used.
G11.12	*require* is used instead of *revert* in *if* statements.
G11.13	The *assert* function is used to test for internal errors and the *require* function is used to ensure a valid condition in input from users and external contracts.
G11.14	Assembly code is only used if necessary.

Test Coverage	**Test Name**
G12.2	Abuse narratives detailed in the threat model are covered by unit tests.
G12.3	Sensitive functions in verified contracts are covered with tests in the development phase.
G12.4	Implementation of verified contracts has been checked for security vulnerabilities using both static and dynamic analysis.
G12.5	Contract specification has been formally verified.
G12.6	The specification and results of the formal verification is included in the documentation.

Decentralized Finance	**Test Name**
G14.1	The lender's contract does not assume its balance (used to confirm loan repayment) to be changed only with its own functions.
G14.2	Functions that change lenders' balance and/or lend cryptocurrency are non-re-entrant if the smart contract allows borrowing the main platform's cryptocurrency (e.g. Ethereum). It blocks the attacks that update the borrower's balance during the flash loan execution.
G14.3	Flash loan functions can only call predefined functions on the receiving contract. If it is possible, define a trusted subset of contracts to be called. Usually, the sending (borrowing) contract is the one to be called back.
G14.4	If it includes potentially dangerous operations (e.g. sending back more ETH/tokens than borrowed), the receiver's function that handles borrowed ETH or tokens can be called only by the pool and within a process initiated by the receiving contract's owner or another trusted source (e.g. multisig).
G14.5	Calculations of liquidity pool share are performed with the highest possible precision (e.g. if the contribution is calculated for ETH it should be done with 18 digit precision - for Wei, not Ether). The dividend must be multiplied by the 10 to the power of the number of decimal digits (e.g. dividend * 10^18 / divisor).
G14.6	Rewards cannot be calculated and distributed within the same function call that deposits tokens (it should also be defined as non-re-entrant). This protects from momentary fluctuations in shares.
G14.7	Governance contracts are protected from flash loan attacks. One possible

	mitigation technique is to require the process of depositing governance tokens and proposing a change to be executed in different transactions included in different blocks.
G14.8	When using on-chain oracles, contracts are able to pause operations based on the oracles’ result (in case of a compromised oracle).
G14.9	External contracts (even trusted ones) that are allowed to change the attributes of a project contract (e.g. token price) have the following limitations implemented: thresholds for the change (e.g. no more/less than 5%) and a limit of updates (e.g. one update per day).
G14.10	Contract attributes that can be updated by the external contracts (even trusted ones) are monitored (e.g. using events) and an incident response procedure is implemented (e.g. during an ongoing attack).
G14.11	Complex math operations that consist of both multiplication and division operations first perform multiplications and then division.
G14.12	When calculating exchange prices (e.g. ETH to token or vice versa), the numerator and denominator are multiplied by the reserves (see the *getInputPrice* function in the *UniswapExchange* contract).

Audit Findings

Debt Deduction When Unstaking Can Underflow

ID	SAY-01
Status	Fixed
Risk	High
Business Impact	Users may not be able to unstake their tokens for a significant amount of time.
Location	- `Portal.sol:242; unstake(uint256)` - `Portal.sol:279; forceUnstakeAll()`

Description	*unstake(uint256)* deducts `(_amount * maxLockDuration) / secondsPerYear` from *maxStakeDebt* and the function *forceUnstakeAll()* does the same for *portalEnergy*.
	
	● `Portal.sol:242; unstake(uint256)`
	`accounts[msg.sender].maxStakeDebt -= (_amount * maxLockDuration) / secondsPerYear;`
	● `Portal.sol:279; forceUnstakeAll()`
	`uint256 remainingDebt = accounts[msg.sender].maxStakeDebt - accounts[msg.sender].portalEnergy;`
	
	As long as *maxLockDuration* does not change, this calculation can never underflow and the subtraction is fine. However, the value can be increased by calling *updateMaxLockDuration()*. In such scenarios, unstaking can fail, although a user's position would be eligible for it.
	
	For instance, consider the example where maxLockDuration is 91.25 days when a user creates a position with 100 tokens. *maxStakeDebt* will be set to 25. Directly afterwards, *maxStakeDebt* is increased to 100 days. In theory, the user should be able to unstake the whole position, but this request would try to decrease maxStakeDebt by ~27.4, which would underflow. In this case, the transaction will always revert and the user's funds will remain locked until the *maxLockDuration* is changed again.

Mitigation	Set the value to 0 if the amount to deduct is larger than the position's *maxStakeDebt*.

Positions Cannot Be Deleted or Recreated

ID	SAY-02
Status	Fixed
Risk	Low
Business Impact	A user that has not interacted with the contract in a long time has an incentive to use a different address for the next stake.
Location	- `Portal.sol; stake(uint256)` - `Portal.sol; _updateAccount(address, uint256)`
Description	In *stake(uint256)*, the logic for creating new positions and updating existing ones is different.

```
/// @dev Check if the user has a staking position, else initialize a new
stake
if(accounts[msg.sender].isExist == true){
   /// @dev Update the user's stake info
   _updateAccount(msg.sender, _amount);
}
else {
                                  [ … ]
}
```

For new positions, the user can directly withdraw the staked tokens and get portal energy based on the current maximum lock duration.

```
else {
   uint256 maxStakeDebt = (_amount * maxLockDuration) / secondsPerYear;
   uint256 availableToWithdraw = _amount;
   uint256 portalEnergy = maxStakeDebt;

   accounts[msg.sender] = Account(true,
       block.timestamp,
       _amount,
       maxStakeDebt,
       portalEnergy,
       availableToWithdraw);
}
```

But when a position is updated, the portal energy is determined based on the staked balance.

- `Portal.sol; _updateAccount(address, uint256)`

```
function _updateAccount(address _user, uint256 _amount) private {
   /// @dev Calculate the accrued portalEnergy since the last update
   uint256 portalEnergyEarned = (accounts[_user].stakedBalance *
   (block.timestamp - accounts[_user].lastUpdateTime)) / secondsPerYear;

                                  [ … ]

   /// @dev Update the user's portalEnergy
   accounts[_user].portalEnergy += portalEnergyEarned;

                                  [ … ]
}
```

After a position is created for a user (and *isExist* is set to true), there is no way to delete it, even if all balances are withdrawn and no more portal energy is withdrawn.

This may lead to undesired behavior in users who haven’t interacted with the system for a long time. Consider a user that has withdrawn all tokens two years ago and has no portal energy left. If they decide to stake again, the update logic would be applied and the portal energy would not be increased based on the staked amount, although this is conceptually a completely new position. If they created the position from a different address instead, they would get the portal energy based on the staked amount.

Mitigation	Create a way for users to delete old inactive positions, perhaps ones that saw no activity for months or years.

Funding Exchange Ratio Cannot Be a Decimal

ID	SAY-03
Status	Irrelevant for now - Acknowledged for future versions
Risk	Low
Business Impact	The funding exchange ratio can only be a whole number, making certain configurations impossible.
Location	- `Portal.sol:647; activatePortal()`
Description	The fundingBalance in *activatePortal()* is directly multiplied by *fundingExchangeRatio* to calculate the required liquidity. Because of this logic, this ratio can only be an integer, values like 2.5 (which may be feasible in certain deployments) cannot be represented in the contract. ● `Portal.sol:647; activatePortal()` `uint256 requiredPortalEnergyLiquidity = fundingBalance * fundingExchangeRatio;`
Mitigation	Consider storing the ratio as a decimal number with 18 decimal places and then dividing by 1e18 after the calculation.

Rounding of Earned Portal Energy Leads to Lower Rewards

ID	SAY-04
Status	Acknowledged for future versions
Risk	Low
Business Impact	The user may receive smaller rewards than expected.
Location	- `Portal.sol:141-142; _updateAccount(address, uint256)`
Description	*_updateAccount(address, uint256)* divides the value earned since last time by 31536000 (the number of seconds each year) every time it is called, i.e. on every modification of the staked position. `uint256 portalEnergyEarned = (accounts[_user].stakedBalance * (block.timestamp - accounts[_user].lastUpdateTime)) / secondsPerYear;` Because division rounds down in Solidity, this can lead to situations where less rewards than expected are accrued. Consider an extreme example where the user's balance is 3153599. If, for example, *_updateAccount(address, uint256)* is called every 10 seconds, no rewards at all are accrued, because the division always rounds down to zero. If it were only called once after a year, the accrued portal energy would be 3153599, i.e. the user loses 3153599 in rewards.
Mitigation	Consider not dividing by secondsPerYear when accruing and only doing the division when the value is used. Like this, the impact of rounding is reduced significantly.

Contracts May not Be Deployable on Arbitrum

ID	SAY-05
Status	Fixed
Risk	Informational
Business Impact	The contracts may not be deployable on Arbitrum because of unsupported opcodes.
Location	—
Description	All contracts are configured to be compiled with Solidity 0.8.20: `pragma solidity ^0.8.20;` Solidity 0.8.20 introduced the PUSH0 opcode, which currently is not supported on Arbitrum. For a deployment on Arbitrum, an older version of Solidity needs to be used or the EVM version needs to be set explicitly in the used framework (if this is not done already).
Mitigation	Either use Solidity 0.8.19 or set the EVM version explicitly in the used framework.

Unnecessary Checks

ID	SAY-06
Status	Acknowledged for future versions
Risk	Informational
Business Impact	The redundant require statement may cause increased gas usage.
Location	- `Portal.sol:282; forceUnstakeAll()` - `Portal.sol:426; buyPortalEnergy(uint256, uint256)` - `Portal.sol:703; _burnPortalEnergyToken(address, uint256)`
Description	The contract contains a few unnecessary checks that are performed multiple times: ● *forceUnstakeAll()* checks the balance before burning. This is not necessary because the burn function reverts if the user does not have enough tokens. `require(IERC20(portalEnergyToken).balanceOf(address(msg.sender)) >= remainingDebt, "Not enough Portal Energy Tokens");` ● *buyPortalEnergy(uint256, uint256)* checks the ERC20 token balance of the user before performing the transfer. This is unnecessary because the transfer will fail if the user does not have sufficient tokens. `require(IERC20(tokenToAcquire).balanceOf(msg.sender) >= _amountInput, "Insufficient balance");` ● *_burnPortalEnergyToken(address, uint256)* checks that the user has a position. However, the function is only called from forceUnstakeAll which already performs this check. `require(accounts[_user].isExist == true);`
Mitigation	These statements can be safely removed.

Value Caching Can Reduce Gas Usage

ID	SAY-07
Status	Fixed
Risk	Informational
Business Impact	Redundant SLOADs unnecessarily increase gas usage.
Location	- `Portal.sol:206-211; stake(uint256)` - `Portal.sol:252-257; unstake(uint256)` - `Portal.sol:279, 304-309; forceUnstakeAll()`
Description	In multiple places, values are written to storage first and later read again in the same function. This increases the gas usage, because every read operation is an additional SLOAD. All functions related to staking have this problem, because they emit the updated accounts field in the event by reading them from storage. However, these fields were directly written / modified by the functions. ● For example, in *unstake(uint256)*, *stakedBalance* and *maxStakeDebt* are modified then read again.

```
/// @dev Update the user's stake info
accounts[msg.sender].stakedBalance -= _amount;
accounts[msg.sender].maxStakeDebt -= (_amount * maxLockDuration) /
secondsPerYear;
accounts[msg.sender].availableToWithdraw -= _amount;

                              [ … ]

/// @dev Emit an event with the updated stake information
emit StakePositionUpdated(msg.sender,
accounts[msg.sender].lastUpdateTime,
accounts[msg.sender].stakedBalance,
accounts[msg.sender].maxStakeDebt,
accounts[msg.sender].portalEnergy,
accounts[msg.sender].availableToWithdraw);
```

Moreover, *forceUnstakeAll()* reads the fields *portalEnergy* and *maxStakeDebt* twice when the first if statement is entered:

```
if(accounts[msg.sender].portalEnergy <
accounts[msg.sender].maxStakeDebt) {

uint256 remainingDebt = accounts[msg.sender].maxStakeDebt -
accounts[msg.sender].portalEnergy;
```

Mitigation	Consider caching these reused values in variables to save gas.