SMART CONTRACT AUDIT REPORT

for

SZNS

Prepared By: Yiqun Chen

PeckShield

September 16, 2021

Document Properties

Client	SZNS
Title	Smart Contract Audit Report
Target	SZNS
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 16, 2021	Xiaotao Wu	Final Release
1.0-rc	August 27, 2021	Xiaotao Wu	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the SZNS smart contracts, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About SZNS

SZNS brings community curation tools to the NFT ecosystem. At launch the protocol will offer DAO management of fractionalized NFT baskets called "Albums". While many platforms offer fractionalization of NFTs, `Albums` takes this concept a step further by combining collective ownership with out-of-the-box decentralized governance. Owners of the `Albums` tokens are not only holding fractional ownership of NFTs, they are able to engage in governance to direct the future of the `Albums`, whether it means adding more NFTs together or exploring other ways to cooperate.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of SZNS

Item	Description
Name	SZNS
Website	https://szns.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 16, 2021

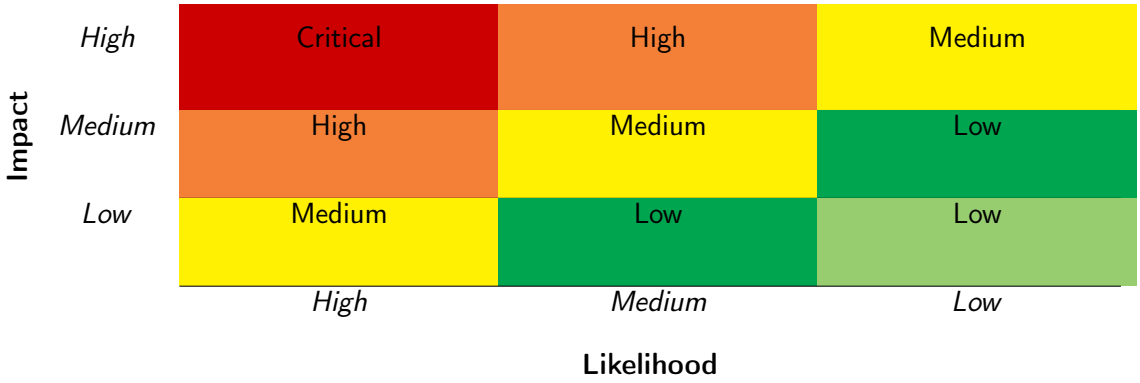
In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/NFTree/szns-sc (93dbbf6)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact, and can be accordingly classified into four categories, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the SZNS smart contracts. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	2	■■
Informational	2	■■
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Reentrancy Risk In SZNSChef	Time and State	Fixed
PVE-002	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Informational	Recommended Explicit Pool Validity Checks	Coding Practices	Confirmed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-006	Informational	Improved Sanity Checks Of System/-Function Parameters	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Potential Reentrancy Risk In SZNSChef

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: SZNSChef
- Category: Time and State [8]
- CWE subcategory: CWE-682 [3]

Description

A common coding best practice in Solidity is the adherence of checks-effects-interactions principle. This principle is effective in mitigating a serious attack vector known as re-entrancy. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [13] exploit, and the recent Uniswap/Lendf.Me hack [12].

In the SZNSChef contract, we notice that the deposit(), withdraw() and emergencyWithdraw() functions have potential reentrancy risk. In the following, we use the deposit() routine as an example. To elaborate, we show below the code snippet of the deposit() routine in SZNSChef. In the deposit() function, we notice pool.token.safeTransferFrom() (line 183) will be called to transfer tokens from msg.sender to the SZNSChef contract for reward allocation. If the pool.token faithfully implements the ERC777-like standard, then the deposit() routine is vulnerable to reentrancy and this risk needs to be properly mitigated.

Specifically, the ERC777 standard normalizes the ways to interact with a token contract while remaining backward compatible with ERC20. Among various features, it supports send/receive hooks to offer token holders more control over their tokens. Specifically, when transfer() or transferFrom() actions happen, the owner can be notified to make a judgment call so that she can control (or even reject) which token they send or receive by correspondingly registering tokensToSend() and

tokensReceived() hooks. Consequently, any transfer() or transferFrom() of ERC777-based tokens might introduce the chance for reentrancy or hook execution for unintended purposes (e.g., mining GasTokens).

In our case, the above hook can be planted in pool.token.safeTransferFrom() (line 183) before the actual transfer of the underlying assets occurs. By doing so, we can effectively keep user.rewardDebt intact (used for the calculation of pending rewards at line 180). With a lower user.rewardDebt, the re-entered deposit() is able to obtain more rewards. It can be repeated to exploit this vulnerability for gains, just like earlier Uniswap/imBTC hack [12].

```
172     // Deposit tokens to SZNSChef for reward allocation.
173     function deposit(uint256 _pid, uint256 _amount) public {
174         PoolInfo storage pool = poolInfo[_pid];
175         UserInfo storage user = userInfo[_pid][msg.sender];
176         updatePool(_pid);
177         if (user.amount > 0) {
178             uint256 pending =
179                 ((user.amount * pool.accRewardPerShare) / 1e12) -
180                     user.rewardDebt;
181             safeRewardTransfer(msg.sender, pending);
182         }
183         pool.token.safeTransferFrom(
184             address(msg.sender),
185             address(this),
186             _amount
187         );
188         user.amount = user.amount + _amount;
189         user.rewardDebt = ((user.amount * pool.accRewardPerShare) / 1e12);
190         emit Deposit(msg.sender, _pid, _amount);
191     }
```

Listing 3.1: SZNSChef::deposit()

Note the withdraw() and emergencyWithdraw() routines in the same contract shares the same issue.

Recommendation Add necessary reentrancy guards to prevent unwanted reentrancy risks.

Status The issue has been fixed by this commit: b057551.

3.2 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: SZNSChef
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In SZNS, the SZNSChef contract is the master of rewards and can mint reward tokens. SZNS users can get these reward tokens by depositing the supported pool tokens to the contract. In particular, one entry routine, i.e., deposit(), accepts user deposits of supported assets (e.g., pool.token). Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the deposit() routine that is used to deposit pool.token to the SZNSChef contract.

```
172     // Deposit tokens to SZNSChef for reward allocation.
173     function deposit(uint256 _pid, uint256 _amount) public {
174         PoolInfo storage pool = poolInfo[_pid];
175         UserInfo storage user = userInfo[_pid][msg.sender];
176         updatePool(_pid);
177         if (user.amount > 0) {
178             uint256 pending =
179                 ((user.amount * pool.accRewardPerShare) / 1e12) -
180                     user.rewardDebt;
181             safeRewardTransfer(msg.sender, pending);
182         }
183         pool.token.safeTransferFrom(
184             address(msg.sender),
185             address(this),
186             _amount
187         );
188         user.amount = user.amount + _amount;
189         user.rewardDebt = ((user.amount * pool.accRewardPerShare) / 1e12);
190         emit Deposit(msg.sender, _pid, _amount);
191     }
```

Listing 3.2: SZNSChef::deposit()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations,

such as deposit(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the transfer() or transferFrom() is expected and aligned well with our operation.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into SZNS for borrowing/lending. In fact, SZNS is indeed in the position to effectively regulate the set of assets that can be listed. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status This issue has been confirmed.

3.3 Recommended Explicit Pool Validity Checks

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SZNSChef
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The SZNSChef contract provides the functionalities of the pool management, the staking/unstaking support, and the reward distribution to various pools and stakers. In the following, we show the key pool data structure. Note all added pools are maintained in an array poolInfo.

```
35      // Info of each pool.
36      struct PoolInfo {
37          IERC20 token; // Address of pool token contract.
38          uint256 allocPoint; // How many allocation points assigned to this pool. Rewards
                to distribute per block.
39          uint256 lastRewardBlock; // Last block number that reward distribution occurs.
40          uint256 accRewardPerShare; // Accumulated rewards per share, times 1e12. See
                below.
41      }
```

```
42      ...
43      // Info of each pool.
44      PoolInfo[] public poolInfo;
```

Listing 3.3: The `PoolInfo` Data Structure in `SZNSChef`

When there is a need to update rewards for an existing pool, query pending rewards, stake (by depositing the supported assets), unstake (by withdrawing previously deposited assets), there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range `[0, poolInfo.length-1]`. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validatePool`. This new modifier essentially ensures the given `_pid` indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not!

```
172     // Deposit tokens to SZNSChef for reward allocation.
173     function deposit(uint256 _pid, uint256 _amount) public {
174         PoolInfo storage pool = poolInfo[_pid];
175         UserInfo storage user = userInfo[_pid][msg.sender];
176         updatePool(_pid);
177         if (user.amount > 0) {
178             uint256 pending =
179                 ((user.amount * pool.accRewardPerShare) / 1e12) -
180                     user.rewardDebt;
181             safeRewardTransfer(msg.sender, pending);
182         }
183         pool.token.safeTransferFrom(
184             address(msg.sender),
185             address(this),
186             _amount
187         );
188         user.amount = user.amount + _amount;
189         user.rewardDebt = ((user.amount * pool.accRewardPerShare) / 1e12);
190         emit Deposit(msg.sender, _pid, _amount);
191     }
```

Listing 3.4: `SZNSChef::deposit()`

Note a number of functions that can be benefited from the new pool-validating modifier, including `pendingRewards()`, `updatePool()`, `deposit()`, `withdraw()`, and `emergencyWithdraw()`.

Recommendation Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
    modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
```

```
    }
```

Listing 3.5: The New `validatePool()` Modifier

Status This issue has been confirmed.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `SZNSChef`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
64      function transfer(address _to, uint _value) returns (bool) {
65          //Default assumes totalSupply can't be over max (2^256 - 1).
66          if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
67              balances[msg.sender] -= _value;
68              balances[_to] += _value;
69              Transfer(msg.sender, _to, _value);
70              return true;
71          } else { return false; }
72      }

74      function transferFrom(address _from, address _to, uint _value) returns (bool) {
75          if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
                balances[_to] + _value >= balances[_to]) {
76              balances[_to] += _value;
77              balances[_from] -= _value;
78              allowed[_from][msg.sender] -= _value;
```

```
79              Transfer(_from, _to, _value);
80              return true;
81          } else { return false; }
82      }
```

Listing 3.6: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `safeRewardTransfer()` routine in the `SZNSChef` contract. If the `USDT` token is supported as `reward` token, the unsafe version of `rewardToken.transfer(_to, bal)` (line 222) and `rewardToken.transfer(_to, _amount)` (line 224) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
218     // Safe reward token transfer function, just in case if rounding error causes pool
            to not have enough.
219     function safeRewardTransfer(address _to, uint256 _amount) internal {
220         uint256 bal = rewardToken.balanceOf(address(this));
221         if (_amount > bal) {
222             rewardToken.transfer(_to, bal);
223         } else {
224             rewardToken.transfer(_to, _amount);
225         }
226     }
```

Listing 3.7: `SZNSChef::safeRewardTransfer()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `transfer()`.

Status The issue has been fixed by this commit: `a4a260e`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `SZNS` protocol, there are certain privileged accounts, i.e., `_owner`, `admin`, `minter` and `pauser`. When examining the related contracts, i.e., `Album` and `SZNSToken`, we notice inherent trust on these privileged accounts. To elaborate, we show below the related functions.

Firstly, the `mint()` function allows for the `minter` role to mint more tokens into circulation without being capped.

```
42      /**
43       * @dev Creates ‘amount‘ new tokens for ‘to‘.
44       *
45       * See {ERC20-_mint}.
46       *
47       * Requirements:
48       *
49       * - the caller must have the ‘MINTER_ROLE‘.
50       */
51      function mint(address to, uint256 amount) public virtual {
52          require(hasRole(MINTER_ROLE, _msgSender()), "ERC20PresetMinterPauser: must have
                minter role to mint");
53          _mint(to, amount);
54      }
```

Listing 3.8: `ERC20PresetMinterPauser::mint()`

Note if the `SZNS` users call the `claim()` function of the `Album` contract to sell `SZNS` tokens in exchange for `ETH`, the calculated `owed ETH` amount for the users can be manipulated by the `minter` role if the `minter` role mints more `SZNS` tokens.

```
67      function claim(uint256 _amount) public {
68          require(bought, "No buyout yet.");
69          uint256 owed = checkOwedAmount(_amount, buyoutCost);
70          payable(msg.sender).transfer(owed);
71          emit BuyoutPortionClaimed(msg.sender, _amount, owed);
72      }
```

Listing 3.9: `AlbumBuyoutManager::claim()`

```
42      function checkOwedAmount(uint256 _amount, uint256 buyoutCost)
43          internal
```

```
44          override
45          returns (uint256 owed)
46      {
47          token.transferFrom(msg.sender, address(this), _amount);
48          owed = (_amount * buyoutCost) / token.totalSupply();
49      }
```

Listing 3.10: `Album::checkOwedAmount()`

Secondly, the `pause()` and `unpause()` functions allow for the `pauser` role to pause or unpause the `SZNS` token transfers.

```
56      /**
57       * @dev Pauses all token transfers.
58       *
59       * See {ERC20Pausable} and {Pausable-_pause}.
60       *
61       * Requirements:
62       *
63       * - the caller must have the `PAUSER_ROLE`.
64       */
65      function pause() public virtual {
66          require(hasRole(PAUSER_ROLE, _msgSender()), "ERC20PresetMinterPauser: must have
                pauser role to pause");
67          _pause();
68      }
69
70      /**
71       * @dev Unpauses all token transfers.
72       *
73       * See {ERC20Pausable} and {Pausable-_unpause}.
74       *
75       * Requirements:
76       *
77       * - the caller must have the `PAUSER_ROLE`.
78       */
79      function unpause() public virtual {
80          require(hasRole(PAUSER_ROLE, _msgSender()), "ERC20PresetMinterPauser: must have
                pauser role to unpause");
81          _unpause();
82      }
```

Listing 3.11: `ERC20PresetMinterPauser::pause()/unpause()`

Lastly, the `addNfts()/sendNfts()/setTimeout()/setBuyout()` functions allow for the `_owner` to add NFTs, remove NFTs, set the `timeout`, and set the `buyer` and `buyoutCost` for the `Album` contract.

```
37      function addNfts(address[] memory _nfts, uint256[] memory _ids)
38          public
39          onlyOwner
40      {
41          _addNfts(_nfts, _ids);
42      }
```

```
43
44      function sendNfts(address to, uint256[] memory idxs) public onlyOwner {
45          _sendNfts(to, idxs);
46      }
47
48      function setTimeout(uint256 _timeout) public onlyOwner {
49          _setTimeout(_timeout);
50      }
```

Listing 3.12: `Album::addNfts()/sendNfts()/setTimeout()`

```
67      function setBuyout(address _buyer, uint256 _cost) public onlyOwner {
68          _setBuyout(_buyer, _cost);
69      }
```

Listing 3.13: `Album::setBuyout()`

We understand the need of the privileged function for contract operation, but at the same time the extra power to the `_owner/minter/pauser/admin` may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Make the list of extra privileges granted to `_owner/minter/pauser/admin` explicit to SZNS users.

Status This issue has been confirmed. Confirmed with the `SZNS` team, much of the admin privileges described are part of the functionality of a DAO and they will clearly doucument the privileges granted to admin accounts in their gitbooks.

3.6 Improved Sanity Checks Of System/Function Parameters

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AlbumBuyoutManager`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In the `AlbumBuyoutManager` contract, the `buyout()` function allows for the verified buyer to buy out all the NFTs held by the `Album` contract at the set price. While reviewing the implementation of this routine, we notice that it can benefit from additional sanity checks.

To elaborate, we show below the full implementation of the `buyout()` function. Specifically, the execution of `sendAllToSender()` will revert if these NFTs held by the `Album` contract have been sold (line 62).

```
58    function buyout() public payable {
59        require(msg.sender == buyer, "Caller is not the buyer.");
60        require(msg.value == buyoutCost, "Not enough ETH.");
61        require(block.timestamp < buyoutEnd, "Buyout timeout already passed.");
62        sendAllToSender();
63        bought = true;
64        emit Buyout(buyer, buyoutCost);
65    }
```

Listing 3.14: `AlbumBuyoutManager::buyout()`

```
52    function sendAllToSender() internal override {
53        address[] memory nfts = getNfts();
54        uint256[] memory ids = getIds();
55        bool[] memory sent = getSent();
56        for (uint256 i = 0; i < nfts.length; i++) {
57            if (!sent[i]) {
58                IERC721(nfts[i]).safeTransferFrom(
59                    address(this),
60                    msg.sender,
61                    ids[i]
62                );
63            }
64        }
65    }
```

Listing 3.15: `Album::sendAllToSender()`

Recommendation Validate the value of state variable `bought` to ensure the NFTs held by the `Album` contract are not sold.

Status The issue has been fixed by this commit: `5607062`.

4 Conclusion

In this audit, we have analyzed the SZNS design and implementation. SZNS brings community curation tools to the NFT ecosystem. At launch the protocol will offer DAO management of fractionalized NFT baskets called "Albums". The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.

[12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[13] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.