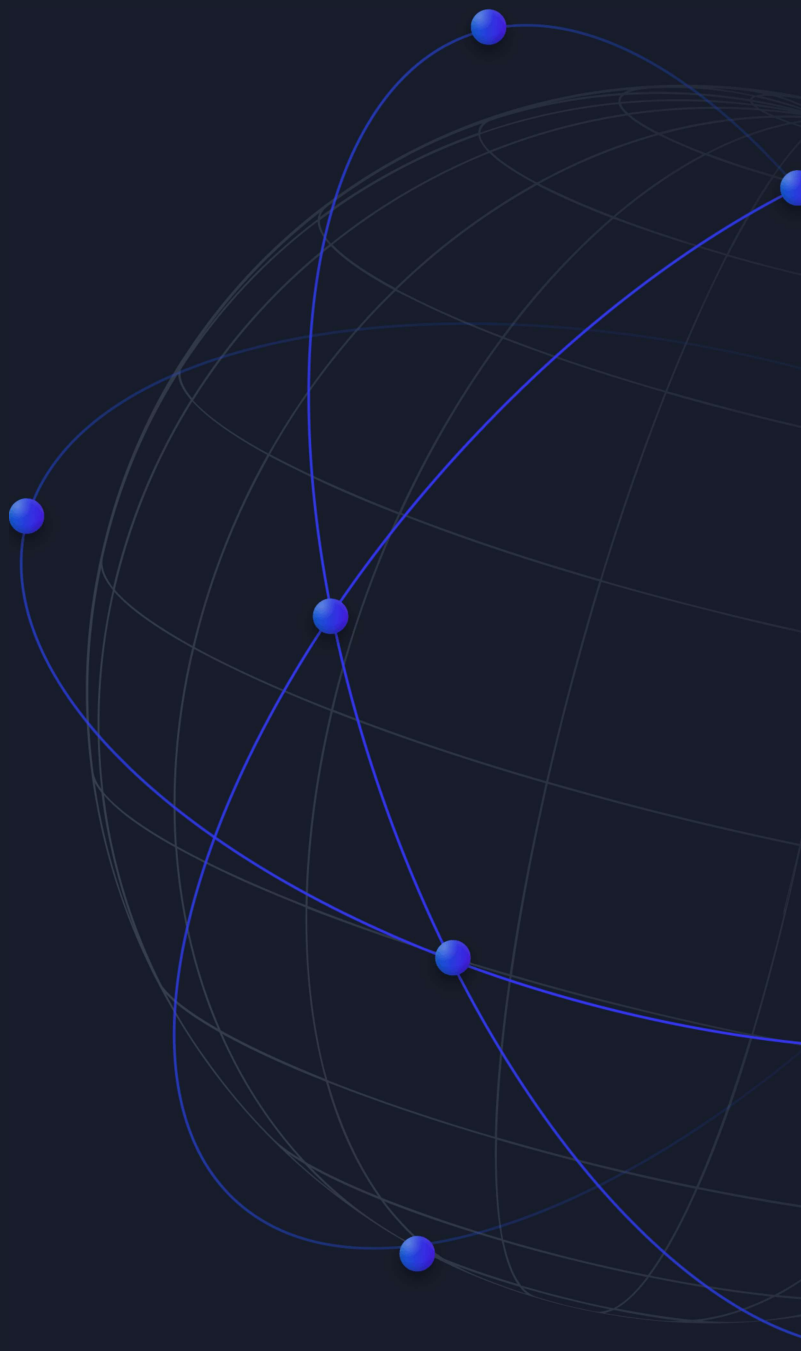
Black paper

Degens

Security audit



First review
06/03/2024

Summary

I. Introduction

1. About Black Paper

Black Paper has been created to help developer teams. Our goal is to help you to make your smart contract safer.
Cybersecurity requires specific expertise which is very different from smart contract development logic. To ensure everything is well fixed, we stay available to help you.

2. Methodology

a. Preparation

As we embark on the audit of the smart contract developed with the Solang compiler and utilizing the SLP Library, it is imperative to approach this task with heightened diligence and awareness due to several critical factors outlined below. The combination of these factors necessitates a thorough and cautious audit process to ensure the security and reliability of the contract.

Short Deadline

The timeline for this audit is notably constrained, with only a few days allocated to complete the review. This limited timeframe places additional pressure on the audit process, emphasizing the need for efficient and focused examination of the contract's code. It is crucial to prioritize the identification of major vulnerabilities that could pose significant risks, while also noting areas where further review would be warranted under a less restrictive timeline.

Solang Compiler Usage

Solang, while a promising tool for Solana smart contract development, is not as widely adopted as other compilers within the blockchain development community. This relative novelty means that best practices, common patterns, and known vulnerabilities are less established compared to more widely used platforms. The Solang compiler itself has undergone only one formal audit to date. This limited external scrutiny suggests that there may be undiscovered vulnerabilities or edge cases in the compiler that could affect the security of contracts it compiles. Given this context, it is essential that the audit pays special attention to the nuances of Solang's compilation process and its potential impact on contract behavior.

Recommendations for Asset Management

In light of the limited audit history of the Solang compiler and the critical role compilers play in the security of smart contracts, it is advisable to exercise caution when deploying significant assets on contracts compiled with Solang. Until the compiler undergoes further audits and its use becomes more widespread—thereby allowing for a broader evaluation of its security profile—it would be prudent to limit the exposure of valuable assets on these contracts. This conservative approach can help mitigate the risks associated with potential, yet undiscovered, compiler-level vulnerabilities.

SLP Library Considerations

The SLP Library, an integral component of the contract's functionality, has not been formally audited. This absence of an audit means that there could be inherent vulnerabilities within the library that have yet to be identified and addressed. The use of unaudited libraries in smart contracts introduces additional risk, as any vulnerabilities present in the library could be exploited to compromise the security of the contract. It is crucial for the audit to carefully review the usage of the SLP Library within the contract, understanding that any findings related to the library itself may require engagement with the library's developers for resolution.

b. Review

Before manually auditing, we pass contracts into automatic tools. This allows us to find some easy-to-find vulnerabilities.
Afterward, we manually go deeper. Every variable and function in the scope are analyzed.

You can find many articles on the lesson website. Here is a snippet list of what we test :

- 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
- Use Of Untrusted Libraries
- Use Of Predictable Variables
- Transaction Ordering Dependence
- Deprecated Uses

This is not an exhaustive list since we also focus on logic execution exploits, and help optimizing gas price.

c. Reporting

Every point in the code is subject to internal discussions with the team. At this stage, a majority of the probable issues have already been identified and documented.

Post the completion of the code review, analysis, and testing, we prepare a report. For each vulnerability, it contains the following informations.

- Description

- Severity
- Recommendation

Here are severity score definitions.

Critical

A critical vulnerability is a severe issue that can cause significant damage to the contract and its users. These vulnerabilities are easy to exploit and can result in the loss of funds, theft of sensitive data, or other serious consequences. Immediate attention is required to address these vulnerabilities.

Major

A major vulnerability is an issue that can cause significant problems for the contract and its users, but not to the same extent as a critical vulnerability. These vulnerabilities are also easy to exploit and may result in the loss of funds or other negative consequences, but they can be mitigated with timely action.

Medium

A medium vulnerability is an issue that could potentially cause problems for the contract and its users, but the difficulty to exploit is higher than major or critical vulnerabilities. These vulnerabilities may pose a risk to the contract's functionality or security, but they can be addressed without causing significant disruption.

Low

A low vulnerability is a minor issue that does not pose a significant risk to the contract or its users. These vulnerabilities are difficult to exploit and may be cosmetic or technical in nature, but they do not compromise the contract's security or functionality.

Informational

An informational finding is not a vulnerability but rather a suggestion or recommendation for improvement. These findings may include best practices for contract design, suggestions for improving code readability, or other non-critical issues. While not urgent, addressing these findings can help to optimize the contract's performance and reduce the risk of future vulnerabilities.

3. Disclaimer

In this audit, we sent all vulnerabilties found by our team. **We can't guarantee all vulnerabilities have been found.**

4. Scope

It is important to note that the audit's scope is limited to the staking.sol file. As such, the following areas are explicitly excluded from this audit:

- The broader ecosystem or platform within which the staking.sol contract operates, except where directly relevant to the contract's functionality.
- The Solang compiler and the SLP Library's inherent vulnerabilities, beyond their interaction with the staking.sol contract. While recommendations may be made regarding these tools, a detailed examination of these components falls outside the audit's scope.
- Frontend interfaces or off-chain components that interact with the staking.sol contract.
- Economic or game theory analysis of the staking mechanism's incentives and potential exploits beyond direct security vulnerabilities.

II. Vulnerabilities

CRIT-1 Unauthorized Ownership Change Vulnerability

Impact: Critical

Description:

The current staking implementation allows any user to stake tokens and become the new "owner" of the staked tokens, before or even after an initial owner has already staked their tokens. This flaw undermines the integrity of the staking process and the security of staked assets, allowing for potential unauthorized ownership changes.

Recommendation:

Step 1: **Introduction of `mintAccount` Variable:** To ensure that only tokens of a specific type (identified by their mint address) can be staked, introduce a `mintAccount` variable in the contract. This variable will store the address of the token mint whose tokens are eligible for staking.

```
// Mint Account
address mintAccount;
```

Step 2: **Initialization in the Constructor:** Initialize the `mintAccount` variable within the contract's constructor. This setup involves receiving the mint address as a parameter to the constructor.

```
constructor(@seed address _mintToken, @bump bytes1 _bump){
    mintAccount = _mintToken;
    emit Init(tx.accounts.dataAccount.key);
}
```

Note: The parameter name `_tokenAddress` has been replaced with `_mintToken` for enhanced clarity, and its type has been specified as `address` to accurately represent its nature.

Step 3: **Validation Against `mintAccount`:** Incorporate logic to verify that the `from` Token Account `mintAccount` storage match the PDA's `mintAccount`. Since SplToken library does not currently implement the `transferChecked` instruction from the SPL Token program, which inherently includes mint address verification, you will need to manually recreate this logic using the `transfer` function from the SPL Token library as a base model.

CRIT-2 Correct Stake Function with Associated Token Account Creation

Impact: Critical

Description:

The original implementation of the stake function does not enforce that tokens are staked to a specific account designated for staking purposes, allowing users to potentially transfer tokens to any account, including themselves. This undermines the intended functionality of staking by not ensuring that tokens are deposited into a controlled, secure account associated with the staking contract or system.

Tokens should be sent to a newly created token account owned by the PDA, specifically an Associated Token Account (ATA), for the staking process.

Recommendation:

To rectify this and implement a secure, standard-compliant staking mechanism, the `stake` function should be modified to include a step for creating an ATA for the _mintToken (if not already existing) before transferring tokens to this ATA. This ensures that tokens are staked to a specific, user-associated token account managed by the staking contract. Below is the revised version of the function incorporating ATA creation:

```
@mutableAccount(from)
@mutableAccount(to)
@signer(owner)
function stake() external {
    // Create an Associated Token Account (ATA) if it doesn't exist
    address ATA_created = createOrGetATA();

    // Transfer tokens to the newly created or existing ATA
    SplToken.transfer(
        tx.accounts.from.key,
        ATA_created,
        tx.accounts.owner.key,
        1
    );

    // Update contract state to reflect the staking action
    owner = tx.accounts.from.key;
    origin = tx.accounts.owner.key;

    // Emit an event to log the staking action
    emit Stake(tx.accounts.from.key, ATA_created); // Corrected to accurately reflect the
transfer destination
}
```

Implementation Notes:

- **ATA Management**: The process for creating or retrieving an ATA should be done with the SPL Associated Token Account Program.
- This change can be done by a few way. It depends on the architecture the technical team want to choose. For instance, a dedicated vault manager can be created to split staking and vault logic in two different programs.

MAJ-1 Improper PDA bump seed validation

Impact: Major

Description:

The current implementation lacks a critical validation step for Program Derived Addresses (PDAs) concerning their bump seeds. PDAs are a unique feature in Solana, allowing the creation of addresses that do not have an associated private key, thus ensuring they can only be manipulated under the program's logic. The bump seed is a crucial component in generating a PDA, ensuring the generated address does not lie on the elliptic curve, thereby not having a corresponding private key. However, the issue lies in the fact that seeds can produce multiple valid PDAs with different bumps. Without validating the bump seed used to be generat the PDA with the first valid value, the program is vulnerable to exploitation. An attacker could potentially generate a PDA with the correct program ID but a different bump seed, misleading the program into interacting with an unauthorized or unintended account.

Recommendation:

To mitigate this vulnerability, it is essential to include a step in the contract's initialization process to validate the bump seed used in PDA generation. This can be achieved by independently deriving the PDA using the provided seeds and the program's address, then comparing the derived bump seed with the one supplied to the constructor. This validation ensures only a unique bump can be used to generate a PDA, . The proposed code snippet to be added to the constructor is as follows:

```
// Independently derive the PDA address from the seeds, bump, and programId
(address pda, bytes1 derived_bump) = try_find_program_address(["staking-degens", _mintToken],
address(this));

// Verify that the bump passed to the constructor matches the bump derived from the seeds and
programId
// This ensures that only the canonical PDA address can be used to create the account (first bump
that generates a valid PDA address)
require(bump == derived_bump, 'INVALID_BUMP');
```

MAJ-2 Inappropriate Mutable Signer Declaration

Impact: Major

Description:

In the `stake` function, the `sender` is incorrectly marked as a `@mutableSigner`, suggesting that the function may modify the state of the `sender` account. However, the `sender` account's role within this context is solely to authorize the transaction through its signature, without requiring any modifications to its state. This misdeclaration could lead to confusion regarding the intended use and security implications of the account within the function.

Recommendation:

Replace the `@mutableSigner(sender)` decorator with `@signer(sender)` to accurately reflect the account's usage within the `stake` function. This change clarifies that the `sender` account is only needed to sign the transaction, aligning with the principle of least privilege and enhancing the contract's security posture. By doing so, you ensure that the contract's intentions are clear, preventing any unnecessary permissions that could potentially be exploited.

Adjusting the account decorator to `@signer` not only corrects the semantic meaning but also ensures that the Solang compiler and runtime enforce the correct permissions for the `sender` account during the execution of the `stake` function. This adjustment helps maintain the integrity of account roles and minimizes the risk of unauthorized modifications to account states.

```
// Before
@mutableSigner(sender)

// After
@signer(sender)
```

MED-1 Owner signature in `unstake` Program

Impact: Medium

Description:

The `unstake` function includes an unnecessary specification for the owner's account as a signer in the token transfer operation. This is not necessary and unsafe because ownership verification is already enforced at the function's start, ensuring only the owner can call `unstake`, and the transfer does not need receiver signature.

Recommendation:

Simplify the `AccountMeta` array by removing the owner as a required signer for the token transfer. Ownership should be verified at the function entry without affecting the transfer operation.

```
AccountMeta[3] metasTransfer = [
    AccountMeta({pubkey : tx.accounts.from.key, is_writable : true, is_signer : false}),
    AccountMeta({pubkey : tx.accounts.to.key, is_writable : true, is_signer : false}),
    AccountMeta({pubkey : tx.accounts.dataAccount.key, is_writable: false, is_signer: true})
];
```

This change maintains the security intent of the program while removing unnecessary signature. Moreover, this changes can be unecessary if using the SLP token library as explain in an other finding.

MED-2 Useless Mutable Account Declaration for PDA in Stake Function

Impact: Medium

Description:

The `stake` function includes an `@mutableAccount(pda)` decorator for a Program Derived Account (PDA), which is designated to indicate that the account's state may be altered within the function. However, in the context of Solang, explicitly marking a storage account (especially a PDA owned by the program) as mutable in this manner is redundant when the account's state change is inherent to the contract's logic. This unnecessary declaration could mislead the understanding of the contract's operation, suggesting a need for mutability where the Solang compiler already detect it.

Recommendation:

Remove the `@mutableAccount(pda)` decorator from the `stake` function to streamline the contract's annotations and align with Solang's conventions regarding account state management.

LOW-1 Unnecessary Declaration of Token Program Address

Impact: Low

Description:

Within the smart contract, the declaration of the `tokenProgram` address as a constant is redundant, given that this address is already defined and accessible within the `spl_token.sol` library. The SPL Token Program address (`TokenkegQfeZyiNwAJbNbGKPFXCWuBvf9Ss623VQ5DA`) is a well-known address on the Solana network that represents the official SPL Token Program, responsible for handling token-related operations. Since this address is a fundamental part of the SPL Token library and is unlikely to change, directly referencing it within individual contracts without a specific need can introduce unnecessary duplication and potential for inconsistency.

Recommendation:

Remove the explicit declaration of the `tokenProgram` constant from the smart contract. Instead, directly utilize the SPL Token library's mechanisms or references to interact with the SPL Token Program. This approach simplifies the contract by eliminating redundant information and ensures that your contract automatically benefits from any updates or enhancements made to the SPL Token library, including the handling of the token program address.

LOW-2 Revision of stake function variable names

Impact: Low

Description:

The original implementation of the `stake` function uses variable names (`sender`, `source`, `destination`) that do not clearly convey the roles of the involved accounts, potentially leading to confusion. Specifically, the terms used do not intuitively match the typical financial transaction terminology (e.g., `from`, `to`, `owner`), which can hinder understanding and maintenance of the code.

Recommendation:

To enhance the readability and clarity of the contract, variable names within the `stake` function should be revised to more accurately reflect the roles of the participating accounts. The recommended changes involve renaming `sender` to `owner` to denote the account initiating the stake, `source` to `from` to indicate the account from which tokens are being staked, and `destination` to `to` to signify the account to which tokens are staked. Additionally, the incorrect use of `@mutableSigner(sender)` should be corrected to `@signer(owner)` to properly indicate that the `owner` is a signer without implying mutability. Here is the revised function:

```
@mutableAccount(from)
@mutableAccount(to)
@signer(owner)
function stake() external {
    SplToken.transfer(
        tx.accounts.from.key,
        tx.accounts.to.key,
        tx.accounts.owner.key,
        1
    );
    owner = tx.accounts.owner.key;
    origin = tx.accounts.from.key;

    emit Stake(tx.accounts.from.key, tx.accounts.to.key);
}
```

LOW-3 Redundant Logic and Underutilization of SplToken Library

Impact: Low

Description:

The current implementation of the `stake` function in the smart contract redundantly recreates token transfer logic, despite the presence of an imported `SplToken` library. This approach not only leads to unnecessary code complexity and potential for errors but also diminishes the utility and benefits of leveraging the standardized and tested functions provided by the `SplToken` library. The manual construction of byte arrays for token transfer instructions, followed by manual account meta creation, makes the contract harder to read, understand, and maintain.

The improvment can be done on the unstake function too.

Recommendation:

To streamline the contract and enhance readability, it is advised to replace the custom token transfer logic with a direct call to the `SplToken.transfer` function provided by the `SplToken` library. The recommended modification to the `stake` function is as follows:

```
@mutableSigner(sender)
@mutableAccount(source)
@mutableAccount(destination)
@mutableAccount(pda)
function stake() external {
    SplToken.transfer(
        tx.accounts.source.key,
        tx.accounts.destination.key,
        tx.accounts.sender.key,
        1
    );
    owner = tx.accounts.sender.key;
    origin = tx.accounts.source.key;

    emit Stake(tx.accounts.source.key, tx.accounts.destination.key);
}
```

This change utilizes the `SplToken` library's `transfer` method to execute the token transfer, specifying the source, destination, and sender accounts along with the transfer amount. It

eliminates the manual assembly of instruction bytes and account meta arrays, thereby improving the contract's clarity and maintainability.

Furthermore, ensure that the `SplToken` library is correctly imported and accessible within the contract's context, and review other parts of the contract to replace any similar redundant logic with appropriate library functions. Adopting library functions not only reduces code redundancy but also aligns the contract with best practices for smart contract development, leveraging community-tested code for standard operations.

LOW-4 Redundant definition of TokenInstruction enum

Impact: Low

Description:

The `TokenInstruction` enum is redundantly defined in the smart contract, despite already being available within the `spl_token.sol` library. This duplication can lead to confusion, maintenance difficulties, and potential integration issues, as the contract's version of the enum might diverge from the standard SPL token instructions over time or could lead to conflicts if both versions are used interchangeably within the project.

Recommendation:

Remove the custom definition of the `TokenInstruction` enum from the smart contract to rely solely on the version provided by the `spl_token.sol` library. This approach ensures consistency with the SPL Token standard and reduces the risk of discrepancies or conflicts arising from having multiple definitions. To access the `TokenInstruction` enum in your contract, import it directly from `spl_token.sol` where needed. Additionally, review the contract's logic to ensure it correctly interfaces with SPL Token instructions using the standard definitions, adjusting any usage of the custom enum to reference the SPL library's version instead.

This change not only simplifies the codebase by eliminating unnecessary code but also enhances maintainability.