QuillAudits

# AUDIT REPORT

September 2025

For

ENCIFHER

# Table of Content

# Executive Summary

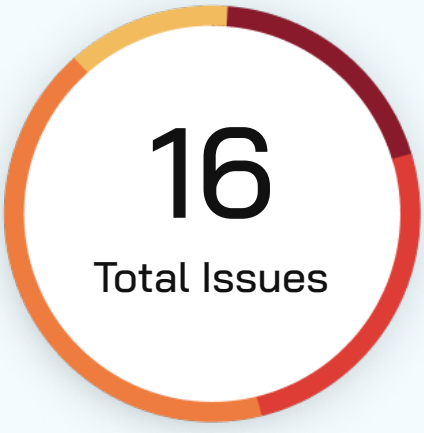
| | |
|---|---|
| **Project Name** | Encifher |
| **Protocol Type** | Payment gateway |
| **Project URL** | https://encifher.io/ |
| **Overview** | Encifher is a comprehensive Solana-based Fully Homomorphic Encryption (FHE) ecosystem with three main components:<br>-pet_executor (the core FHE engine),<br>-etoken (encrypted SPL tokens), and<br>-swap_om (encrypted order matching).<br><br>The pet_executor program provides fundamental encrypted operations (add, subtract, compare, select) using Euint64handles that represent encrypted values, with a relayer system for decryption requests. The etoken program implements privacy-preserving tokens where balances and transfer amounts remain encrypted throughout all operations, while ewrapper enables bridging between regular SPL tokens and encrypted tokens.<br><br>The swap_om program demonstrates a practical application by implementing an encrypted order book where users can place orders with hidden amounts and receiver addresses, maintaining complete transaction privacy. All cryptographic operations use deterministic hashing to simulate FHE computations, with type safety enforced through handle encoding (type information stored in the lower bits of 128-bit handles).<br><br>The system enables private DeFi operations where transaction amounts, balances, and participant identities can remain confidential while still allowing for automated matching and settlement. |
| **Audit Scope** | The scope of this Audit was to analyze the Encifher Smart Contracts for quality, security, and correctness. |
| **Source Code link** | https://github.com/RizeLabs/encifher-svm-contracts/tree/feat/mainnet-programs |

| | |
|---|---|
| **Contracts in Scope** | 1. programs/etoken<br>2. programs/ewrapper<br>3. programs/pet_executor<br>4. programs/swap_om/src<br>5. Program/acl |
| **Branch** | feat/mainnet-programs |
| **Commit Hash** | de3993f2ab84a1591d07d50afdb74b211acf0645 |
| **Language** | Rust |
| **Blockchain** | Solana |
| **Method** | Manual Analysis, Functional Testing, Automated Testing |
| **Review 1** | 23rd July 2025 - 4th August 2025 |
| **Updated Code Received** | 27th August 2025 |
| **Review 2** | 27th August 2025 - 2nd September 2025 |
| **Fixed In** | df934ad525ccb119a35488b7054ea71335bdc470 |

## Verify the Authenticity of Report on QuillAudits Leaderboard:

https://www.quillaudits.com/leaderboard

# Number of Issues per Severity




| | |
|---|---|
| Critical | 3 (18.75%) |
| High | 4 (25%) |
| Medium | 7 (43.75%) |
| Low | 2 (12.5%) |
| Informational | 0 (0%) |

Severity

| Issues | Critical | High | Medium | Low | Informational |
|---|---|---|---|---|---|
| Open | 0 | 0 | 0 | 0 | 0 |
| Acknowledged | 1 | 1 | 6 | 1 | 0 |
| Partially Resolved | 0 | 0 | 0 | 0 | 0 |
| Resolved | 2 | 3 | 1 | 1 | 0 |

# Summary of Issues

| Issue No. | Issue Title | Severity | Status |
|---|---|---|---|
| 1 | Unrestricted Burn Allows Any User to Burn Tokens from Arbitrary Accounts | Critical | Resolved |
| 2 | Duplicate Mutable Accounts in `etransfer` Allow Arbitrary Balance Inflation | Critical | Resolved |
| 3 | Missing Program Token Account Enforcement in `etransfer` Allows Repeated Unauthorized Withdrawals | Critical | Acknowledged |
| 4 | Burn Instruction Allows Burning from Frozen Accounts, Bypassing Freeze Protection | High | Resolved |
| 5 | Unrestricted Mint Initialization Allows Malicious Decimals and Freeze Authority Assignment | High | Resolved |
| 6 | Token Supply Not Decreased on Burn, Enabling DoS and Supply Inaccuracy | High | Acknowledged |
| 7 | Minting Allowed to Frozen Token Accounts, Leading to Permanent Token Lock | High | Resolved |
| 8 | Mismatched Parameter Names Between Instruction Functions and Account Structs Break Anchor Seed Constraints | Medium | Acknowledged |

| Issue No. | Issue Title | Severity | Status |
|---|---|---|---|
| 9 | Double Accounting of rewardsEarly Return in `grant_batch_access` Causes Only First Handle to Be Processed | Medium | Acknowledged |
| 10 | Faulty Conditional on Storage Index Prevents Proper Storage Initialization and Causes DoS | Medium | Acknowledged |
| 11 | Broken freeze/thaw functionality | Medium | Resolved |
| 12 | Incorrect Program ID Used for Mint Authority Derivation Causes Permanent DoS in Minting | Medium | Acknowledged |
| 13 | Incorrect Handle Hash Inserted in `delegate_access` When Current Storage Is Full | Medium | Acknowledged |
| 14 | Incorrect Storage Index Increment Prevents Proper Storage Account Initialization and Breaks Access Revocation | Medium | Acknowledged |
| 15 | Unnecessary data allocation for liquidity vaultManual Initialization of Storage Accounts Increases Operational Overhead | Low | Acknowledged |
| 16 | Lack of Pausing Functionality Exposes Protocol to Irrecoverable Exploits | Low | Resolved |

# Checked Vulnerabilities

We have scanned the solana program for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Signer authorization
- Account data matching
- Sysvar address checking
- Owner checks
- Type cosplay
- Initialization
- Arbitrary cpi
- Duplicate mutable accounts
- Bump seed canonicalization
- PDA Sharing
- Incorrect closing accounts
- Missing rent exemption checks
- Arithmetic overflows/underflows
- Numerical precision errors
- Solana account confusions
- Casting truncation
- Insufficient SPL token account verification
- Signed invocation of unverified programs

# Techniques and Methods

**Throughout the audit of Solana Programs, care was taken to ensure:**

- The overall quality of code
- Use of best practices
- Code documentation and comments, match logic and expected behavior
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper
- Efficient use of gas
- Code is safe from re-entrancy and other vulnerabilities

**The following techniques, methods, and tools were used to review all the smart contracts:**

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

### Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

### Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

# Types of Severity

Every issue in this report has been assigned to a severity level. There are five levels of severity, and each of them has been explained below.

### Critical: Immediate and Catastrophic Impact

Critical issues are the ones that an attacker could exploit with relative ease, potentially leading to an immediate and complete loss of user funds, a total takeover of the protocol's functionality, or other catastrophic failures. Critical vulnerabilities are non-negotiable; they absolutely must be fixed.

### High (H): Significant Risk of Major Loss or Compromise

High-severity issues represent serious weaknesses that could result in significant financial losses for users, major malfunctions within the protocol, or substantial compromise of its intended operations. While exploiting these vulnerabilities might require specific conditions to be met or a moderate level of technical skill, the potential damage is considerable. These findings are critical and should be addressed and resolved thoroughly before the contract is put into the Mainnet.

### Medium (M): Potential for Moderate Harm Under Specific Circumstances

Medium-severity bugs are loopholes in the protocol that could lead to moderate financial losses or partial disruptions of the protocol's intended behavior. However, exploiting these vulnerabilities typically requires more specific and less common conditions to occur, and the overall impact is generally lower compared to high or critical issues. While not as immediately threatening, it's still highly recommended to address these findings to enhance the contract's robustness and prevent potential problems down the line.

### Low (L): Minor Imperfections with Limited Repercussions

Low-severity issues are essentially minor imperfections in the smart contract that have a limited impact on user funds or the core functionality of the protocol. Exploiting these would usually require very specific and unlikely scenarios and would yield minimal gain for an attacker. While these findings don't pose an immediate threat, addressing them when feasible can contribute to a more polished and well-maintained codebase.

### Informational (I): Opportunities for Improvement, Not Immediate Risks

Informational findings aren't security vulnerabilities in the traditional sense. Instead, they highlight areas related to the clarity and efficiency of the code, gas optimization, the quality of documentation, or adherence to best development practices. These findings don't represent any immediate risk to the security or functionality of the contract but offer valuable insights for improving its overall quality and maintainability. Addressing these is optional but often beneficial for long-term health and clarity.

# Types of Issues

**Open**

Security vulnerabilities identified that must be resolved and are currently unresolved.

**Resolved**

These are the issues identified in the initial audit and have been successfully fixed.

**Acknowledged**

Vulnerabilities which have been acknowledged but are yet to be resolved.

**Partially Resolved**

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

# Severity Matrix

Impact

| Likelihood | High | Medium | Low |
|---|---|---|---|
| High | Critical | High | Medium |
| Medium | High | Medium | Low |
| Low | Medium | Low | Low |

**Impact**

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low** - can lead to any kind of unexpected behavior with some of the protocol's functionalities that's not so critical.

**Likelihood**

- High - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost.
- Medium - only a conditionally incentivized attack vector, but still relatively likely.
- Low - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive.

# Critical Severity Issues

## Unrestricted Burn Allows Any User to Burn Tokens from Arbitrary Accounts

Resolved

### Path

etoken

### Function Name

`burn`

### Description

The `burn` instruction in the `etoken` program is vulnerable because it allows any user to burn tokens from any other user's token account. The instruction takes a `to: Pubkey` parameter and uses it to derive the token account to burn from:

```rust
#[derive(Accounts)]
#[instruction(to: Pubkey)]
pub struct MintTo<'info> {
    #[account(mut)]
    pub authority: Signer<'info>,
    pub mint: InterfaceAccount<'info, Mint>,
    #[account(
        mut,
        seeds = [b"emint", mint.key().as_ref()],
        bump,
    )]
    pub emint: Account<'info, EMint>,
    #[account(
        mut,
        seeds = [b"eaccount", emint.key().as_ref(), to.as_ref()],
        bump
    )]
    pub token_account: Account<'info, ETokenAccount>,
    // ...
}
```

In the handler:

```rust
pub fn process_burn(ctx: Context<MintTo>, to: Pubkey, amount: Einput) -> Result<Euint64> {
    let token_account = &mut ctx.accounts.token_account;
    let from_amount = token_account.amount.clone();
    // ...burn logic...
}
```

There is no check that the `authority` (signer) is the owner of the `token_account`. This means any user can specify any `to` address and burn tokens from any account, not just their own.

### Impact

- Loss of funds: Any malicious user can burn tokens from any other user's account, resulting in permanent loss of tokens for victims.
- Denial of service: Attackers can grief users by burning their tokens, making the token unusable or untrustworthy.

### Recommendation

Enforce ownership check: Before allowing a burn, ensure that the `authority` (signer) is the owner of the `token_account`. Add a check in the handler:

```
require!(
        token_account.owner == ctx.accounts.authority.key(),
        CustomError::InvalidOwner
    );
```

This ensures that only the owner of a token account can burn tokens from it, protecting users from unauthorized

### Specific Fixed In Commit

https://github.com/RizeLabs/encifher-svm-contracts/commit/5b0715f3e4dda8373e84cd316f572ea87d838f55

### Encifher team's Response

Added the recommended authority check

## Duplicate Mutable Accounts in `etransfer` Allow Arbitrary Balance Inflation

Resolved

### Path

etoken

### Function Name

**`etransfer`**

### Description

The `etransfer` instruction in the `etoken` program allows users to specify both the `from` and `to` accounts. There is no restriction preventing a user from passing the same account for both parameters. This leads to the classic "duplicate mutable accounts" bug in Solana, where the same account is passed as two mutable references, allowing the attacker to bypass intended logic and arbitrarily increase their token balance.

```
pub fn etransfer(
    ctx: Context<ETransfer>,
    receiver: Pubkey,
    amount: Einput,
) -> Result<()> {
    process_etransfer(ctx, receiver, amount)
}
```

In the `ETransfer` context, both `from` and `to` are mutable accounts, and there is no check that they are different when called directly.

### Concrete Exploit Example

Suppose Alice has a token account with a balance of 100 tokens. She calls `etransfer` and passes her own account as both the sender and receiver:

```
// Alice's token account: 100 tokens

// Alice calls:
etoken::etransfer(
    ctx, // where ctx.accounts.from == ctx.accounts.to == Alice's token account
    alice_pubkey, // receiver
    amount // some Einput representing 100 tokens
)
```

Inside the handler, the logic:

```
let from_account = &mut ctx.accounts.from;
let to_account = &mut ctx.accounts.to;

// Subtract from sender
from_account.amount = from_account.amount - 100; // 100 - 100 = 0

// Add to receiver
to_account.amount = to_account.amount + 100; // 100 + 100 = 200
```

### Example with values

**1. Initial state:**

Alice's token account: `amount = 100`

**2. After subtraction:**

`from_account.amount = 100 - 100 = 0`

**3. After addition:**

Alice's token account: `amount = 200` (doubled)

### Impact

- Arbitrary inflation of token balances: by malicious users without actually transferring tokens to protocol
- Potential for draining liquidity or breaking integrations: that rely on correct balances.

### Recommendation

Add a check in the `etransfer` instruction to ensure that `from` and `to` accounts are not the same:

```
require!(
        ctx.accounts.from.key() != ctx.accounts.to.key(),
        CustomError::InvalidTransferAccounts
    );
```

### Reference

Solana Sealevel Attacks: Duplicate Mutable

Accounts)(https://github.com/coral-xyz/sealevel-attacks/blob/master/programs/6-duplicate-mutable-accounts/insecure/src/lib.rs)

Solana StackExchange: Why does duplicate account deserialization behavior contradict expected logic in Solana?)(https://solana.stackexchange.com/questions/22485/why-does-duplicate-account-deserialization-behavior-contradict-expected-logic-in)

**Specific Fixed In Commit**

https://github.com/RizeLabs/encifher-svm-contracts/commit/5b0715f3e4dda8373e84cd316f572ea87d838f55

**Encifher team's Response**

Added the recommended constraint for passing duplicate accounts

## Missing Program Token Account Enforcement in `etransfer` Allows Repeated Unauthorized Withdrawals

Acknowledged

### Path
etoken

### Function Name
`etransfer`

### Description
In the `etransfer` function, the program allows users to specify arbitrary `from` and `to` accounts for token transfers. However, there is no check to ensure that the `to` account is the designated program token account . This enables an attacker to transfer tokens to any account they control, rather than the intended program vault.

### Expected Flow

1. AttackerA: transfers 100 tokens from their own account to another account they control (e.g., AttackerB) using `etransfer`.
2. AttackerB then withdraws 100 real tokens from the program using the withdrawal flow.
3. AttackerB: transfers 100 tokens to yet another account they control (e.g., AttackerC) using `etransfer`.
4. AttackerC: withdraws another 100 real tokens.
5. This process can be repeated, allowing the attacker to withdraw multiple times the amount of tokens they originally deposited, draining the vault.

Vulnerable code Example

```
#[account(
    mut,
    seeds = [b"eaccount", emint.key().as_ref(), order_manager.owner.as_ref()],
    bump,
    seeds::program = etoken::ID,
)]
pub program_etoken_account: Account<'info, ETokenAccount>,
```

But in the transfer logic, there is no check that the `to` account is actually this `program_etoken_account`:

```
pub fn etransfer(
    ctx: Context<ETransfer>,
    receiver: Pubkey,
    amount: Einput,
) -> Result<()> {
    process_etransfer(ctx, receiver, amount)
}
```

### Impact

- Vault draining: Attackers can repeatedly transfer tokens to new accounts they control and withdraw real tokens multiple times, extracting more value than they are entitled to.
- Double withdrawal: Users can withdraw from both the original and the new account, effectively doubling (or more) their withdrawal.
- Loss of funds: The program's vault can be drained, resulting in a total loss of user and protocol funds.

### Recommendation

Enforce that the `to` account in `etransfer` is always the program's designated token account (e.g., `program_etoken_account` for the relevant mint and owner).

- Add a check in the instruction handler:

```
require!(
        ctx.accounts.to.key() == //program token account for that mint,
        CustomError::InvalidDestinationAccount
    );
```

Never allow arbitrary destination accounts for internal program transfers that are meant to credit the program's vault.

This ensures that tokens can only be transferred to the correct program-controlled account, preventing attackers from exploiting the system to withdraw more than their fair share.

### Specific Fixed In Commit

**NA**

### Encifher team's Response

NA since anchor by default expects account matching the e seeds

# High Severity Issues

## Burn Instruction Allows Burning from Frozen Accounts, Bypassing Freeze Protection

Resolved

### Path

etoken

### Function

`burn`

### Description

The `burn` instruction in the `etoken` program does not check if the token account is frozen before allowing tokens to be burned. This means that even if an account has been frozen (using the `freeze_account` instruction), tokens can still be burned from it. This oversight renders the freeze functionality ineffective for preventing burns, as freezing is intended to lock an account from all token operations.

```
pub fn process_burn(ctx: Context<MintTo>, to: Pubkey, amount: Einput) -> Result<Euint64> {
    let token_account = &mut ctx.accounts.token_account;
    // ...burn logic...
    // No check for token_account.is_frozen!
    Ok(amount)
}
```

### Impact

Bypasses freeze protection - Attackers or malicious actors can burn tokens from accounts that have been intentionally frozen, defeating the purpose of the freeze mechanism.

### Remediation

Add a check in the burn instruction to ensure the token account is not frozen before burning:

```
require!(
        !token_account.is_frozen,
        CustomError::AccountFrozen
    );
```

Place this check at the start of the `process_burn` function, before any burn logic is executed.

This ensures that tokens cannot be burned from frozen accounts, preserving the integrity and intent of

**Specific Fixed In Commit**

https://github.com/RizeLabs/encifher-svm-contracts/commit/5b0715f3e4dda8373e84cd316f572ea87d838f55

**Encifher team's Response**

Added recommended account status check

## Unrestricted Mint Initialization Allows Malicious Decimals and Freeze Authority Assignment

Resolved

### Path

etoken

### Function

`initialize_mint`

### Description

The `initialize_mint` instruction in the `etoken` program is currently not admin-gated, allowing any user to call it and initialize a new `EMint` account for any SPL mint. This exposes two critical risks:

**1. Decimals Manipulation:**

The caller can set an arbitrary value for `decimals`, which is later used in supply and transfer calculations. If set to an extremely high or low value, this can cause denial-of-service (DoS) or break the token's economic logic. For example, the minting logic checks:

```rust
require!(
    mint.supply.checked_add(amount).unwrap() <=
    1_000_000_000 * 10u64.pow(mint.decimals as u32),
    CustomError::ExceedsMaxSupply
);
```

If `decimals` is set too high, the max supply becomes unreasonably large; if too low, users may be unable to mint or transfer reasonable amounts.

**2. Freeze Authority Hijack:**

The caller can set themselves (or any arbitrary address) as the `freeze_authority` for the mint:

```rust
emint.freeze_authority = Some(authority);
```

Since there is no mechanism to thaw frozen accounts, a malicious actor could freeze any token account, permanently locking user funds.

### Vulnerable code snippet:

```rust
pub fn process_init_mint(
    ctx: Context<InitializeMint>,
    decimals: u8,
    authority: Pubkey
) -> Result<()> {
    let emint = &mut ctx.accounts.emint;
    if emint.is_initialized {
        return Err(CustomError::AlreadyInitialized.into());
    }
    emint.is_initialized = true;
    emint.mint = ctx.accounts.mint.key();
    emint.supply = 0;
    emint.decimals = decimals; // <-- user-controlled
    emint.authority = authority; // <-- user-controlled
    emint.freeze_authority = Some(authority); // <-- user-controlled
    Ok(())
}
```

### Impact

- Denial of Service (DoS): Malicious users can set extreme decimals, making the token unusable for minting or transferring.
- Permanent Account Freezing: Attackers can set themselves as freeze authority and freeze any token account, with no way to thaw, resulting in permanent loss of user funds.

### Recommendation

**1. Admin-Gate the Instruction:**

Restrict `initialize_mint` so that only a trusted admin or governance authority can call it. For example, add an `admin` signer and check:

```rust
#[account(address = <ADMIN_PUBKEY>)]
pub admin: Signer<'info>,
```

And require the signer in the handler.

**2. Fetch Decimals from Actual Mint:**

Instead of accepting `decimals` as a parameter, read the decimals from the actual SPL mint account:

```rust
emint.decimals = ctx.accounts.mint.decimals;
```

This ensures consistency and prevents manipulation.

**3. Freeze Authority Assignment:**

Only allow a trusted authority to be set as the freeze authority, or remove the freeze authority if not needed.

```
Example Remediation:**
```rust
pub fn process_init_mint(
    ctx: Context<InitializeMint>,
    authority: Pubkey
) -> Result<()> {
    let emint = &mut ctx.accounts.emint;
    if emint.is_initialized {
        return Err(CustomError::AlreadyInitialized.into());
    }
    // Only admin can call this
    require!(ctx.accounts.admin.key() == <ADMIN_PUBKEY>, CustomError::Unauthorized);

    emint.is_initialized = true;
    emint.mint = ctx.accounts.mint.key();
    emint.supply = 0;
    emint.decimals = ctx.accounts.mint.decimals; // <-- fetch from actual mint
    emint.authority = authority;
    emint.freeze_authority = Some(authority);
    Ok(())
}
```

Without admin gating and proper validation, anyone can initialize a mint with malicious parameters, leading to DoS and permanent loss of funds. Always restrict mint initialization and fetch critical.

**Example Recommendation:**

```
```rust
pub fn process_init_mint(
    ctx: Context<InitializeMint>,
    authority: Pubkey
) -> Result<()> {
    let emint = &mut ctx.accounts.emint;
    if emint.is_initialized {
        return Err(CustomError::AlreadyInitialized.into());
    }
    // Only admin can call this
    require!(ctx.accounts.admin.key() == <ADMIN_PUBKEY>, CustomError::Unauthorized);

    emint.is_initialized = true;
    emint.mint = ctx.accounts.mint.key();
    emint.supply = 0;
    emint.decimals = ctx.accounts.mint.decimals; // <-- fetch from actual mint
    emint.authority = authority;
    emint.freeze_authority = Some(authority);
    Ok(())
}
```
```

**Summary**

Without admin gating and proper validation, anyone can initialize a mint with malicious parameters, leading to DoS and permanent loss of funds. Always restrict mint initialization and fetch critical

**Specific Fixed In Commit**

https://github.com/RizeLabs/encifher-svm-contracts/commit/5b0715f3e4dda8373e84cd316f572ea87d838f55

**Encifher team's Response**

removed decimals from fn params and assigned original mint's decimals.
```

## Token Supply Not Decreased on Burn, Enabling DoS and Supply Inaccuracy

Acknowledged

### Path

etoken

### Function

`process_burn`

### Description

In the `process_burn` function of the `etoken` program, when tokens are burned (i.e., removed from a user's account), the total supply in the `EMint` account is not decreased by the burned amount. This is a critical flaw: the circulating supply is reduced, but the recorded total supply remains unchanged and inflated.

### Vulnerable Code Snippet:

```rust
pub fn process_burn(ctx: Context<MintTo>, to: Pubkey, amount: Einput) -> Result<Euint64> {
    let token_account = &mut ctx.accounts.token_account;
    // ...burn logic...
    // No code to decrease emint.supply by the burned amount!
    Ok(amount)
}
```

Meanwhile, minting enforces a strict supply cap:

```rust
require!(
    mint.supply.checked_add(amount).unwrap() <=
    1_000_000_000 * 10u64.pow(mint.decimals as u32),
    CustomError::ExceedsMaxSupply
);
```

But since burning does not decrease `mint.supply`, the cap is never relaxed after tokens are burned.

### Impact:

#### Inaccurate Supply Data:

The reported total supply is higher than the actual circulating tokens, misleading users and integrators.

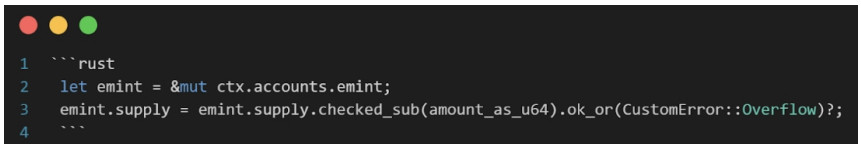
#### Denial of Service (DoS):

- A malicious user (whale) can mint up to the maximum supply, then burn (withdraw) their tokens.
- Since burning does not decrease supply, no one else can mint new tokens, as the supply cap is still reached.
- This locks out all other users from minting or interacting with the token, effectively freezing the token ecosystem.

### Recommendation

Decrease the total supply on burn:

In the `process_burn` function, after burning tokens from the user's account, subtract the burned amount from `emint.supply`:



```rust
let emint = &mut ctx.accounts.emint;
emint.supply = emint.supply.checked_sub(amount_as_u64).ok_or(CustomError::Overflow)?;
```

Where `amount_as_u64` is the actual amount burned, converted to `u64`.

### Specific Fixed In Commit

NA

### Encifher team's Response

NA since we are no longer maintaining supply logic as it un-necessary

## Minting Allowed to Frozen Token Accounts, Leading to Permanent Token Lock

Resolved

### Path

etoken

### Function

`process_mint_to`

### Description

In the `process_mint_to` function, the program checks if the mint is frozen before allowing minting:

```rust
require!(!mint.is_frozen, CustomError::MintFrozen);
```

Which is incorrect, and it should have checked if the recipient token account is frozen. This means tokens can be minted to a frozen token account. Once tokens are minted to a frozen account, the user cannot transfer them out, because the transfer logic correctly checks:

```rust
require!(from_account.owner == authority.key(), CustomError::InvalidOwner);
require!(!from_account.is_frozen, CustomError::AccountFrozen);
require!(!to_account.is_frozen, CustomError::AccountFrozen);
require!(from_account.mint == to_account.mint, CustomError::MintMismatch);
```

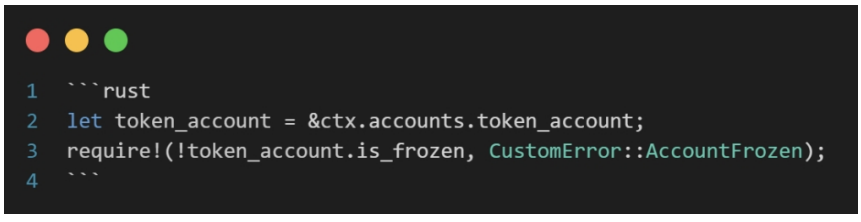
As a result, tokens minted to a frozen account become permanently locked and unusable, given that currently there is no mechanism for thawing a token account.

### Impact:

- Permanent loss of tokens: Users can receive tokens in a frozen account but will never be able to transfer or use them.
- Poor user experience: Users may not realize their account is frozen until after minting, resulting in loss of funds.

### Recommendation

• Add a check in `process_mint_to` to ensure the recipient token account is not frozen:




Place this check before minting tokens to the account. This will prevent minting to frozen accounts and avoid permanent token lock.

### Specific Fixed In Commit

https://github.com/RizeLabs/encifher-svm-contracts/commit/5b0715f3e4dda8373e84cd316f572ea87d838f55

### Encifher team's Response

added recommended account check

# Medium Severity Issues

## Mismatched Parameter Names Between Instruction Functions and Account Structs Break Anchor Seed Constraints

Acknowledged

### Path

ACL

### Function

`Most functions`

### Description

Several instructions in the ACL program (such as `CheckAccess`, `RevokeAccess`, and `InitializeAclStorage`) have a mismatch between the parameter names in the instruction function and the names used in the corresponding `#[instruction(...)]` attribute of the `#[derive(Accounts)]` struct. While the data types match, Anchor relies on both name and order to inject values from the instruction into the account struct. If the names do not match, Anchor will not populate the value, leading to runtime errors when constraints (such as `seeds`) are evaluated.

### Example of the Issue

```rust
pub fn initialize_acl_storage(
    ctx: Context<InitializeAclStorage>,
    acl_storage_index: u64, // <-- parameter name is `acl_storage_index`
) -> Result<()> { ... }

#[derive(Accounts)]
#[instruction(storage: u64)] // <-- expects `storage`, not `acl_storage_index`
pub struct InitializeAclStorage<'info> {
    #[account(
        init,
        payer = payer,
        space = AclStorageAccount::LEN,
        seeds = [b"acl_storage", storage.to_le_bytes().as_ref()], // uses `storage`
        bump
    )]
    pub storage_account: Account<'info, AclStorageAccount>,
    // ...
}
```

Here, the instruction expects `storage`, but the function provides `acl_storage_index`. As a result, Anchor cannot injec t the value, and the seeds constraint fails at runtime.

### Impact

Denial of Service (DoS): Instructions will fail at runtime with seed constraint errors, preventing initialization or accessing accounts as intended

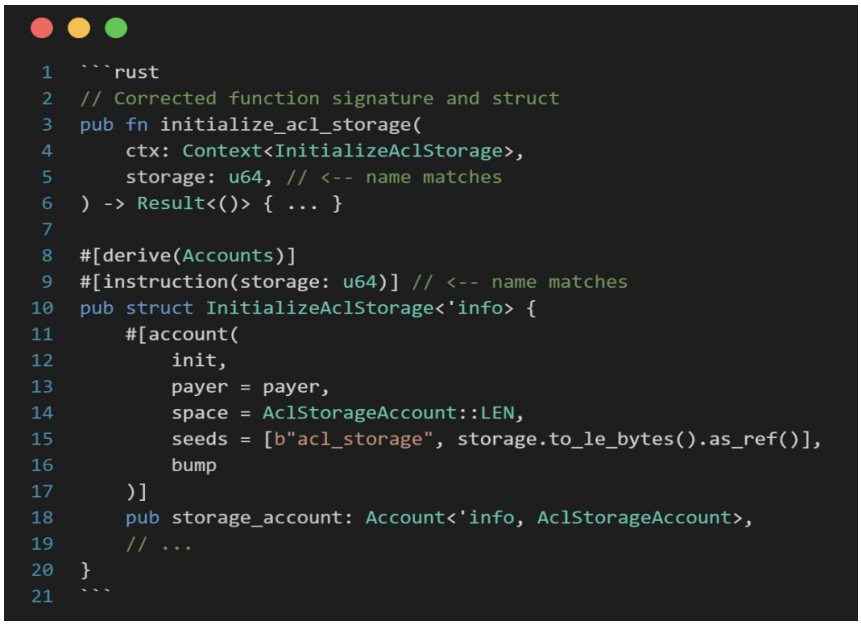
### Recommendation

Always ensure that the parameter names in the instruction function exactly match the names in the `#[instruction(...)]` attribute of the account struct, and that the order is preserved.

- Rename function parameters to match the names used in the account struct's `#[instruction(...)]` attribute.

- Review all instructions and account structs for consistency.

### Example Recommendation :



```rust
// Corrected function signature and struct
pub fn initialize_acl_storage(
    ctx: Context<InitializeAclStorage>,
    storage: u64, // <-- name matches
) -> Result<()> { ... }

#[derive(Accounts)]
#[instruction(storage: u64)] // <-- name matches
pub struct InitializeAclStorage<'info> {
    #[account(
        init,
        payer = payer,
        space = AclStorageAccount::LEN,
        seeds = [b"acl_storage", storage.to_le_bytes().as_ref()],
        bump
    )]
    pub storage_account: Account<'info, AclStorageAccount>,
    // ...
}
```

### Rule of Thumb

Anchor matches by name and order, not just type. If the names mismatch, Anchor will not inject the value, and constraints will

### Specific Fixed In Commit

NA

### Encifher team's Response

We are no longer maintaining ACL onchain

## Double Accounting of rewardsEarly Return in `grant_batch_access` Causes Only First Handle to Be Processed

Acknowledged

### Path
acl

### Function
`grant_batch_access`

### Description
In the `grant_batch_access` function, the intention is to process up to 10 handles in a single batch. However, due to an early `return Ok(())` statement inside the loop, only the first handle in the `handles` vector is processed. After the first successful insertion, the function exits, and the remaining handles are ignored.

### Vulnerable Code Snippet:

```rust
for handle in handles {
    let handle_hash = get_handle_hash(ctx.accounts.payer.key(), handle);
    // ...
    if current_storage.handle_hashes.len() < 100 {
        match current_storage.handle_hashes.binary_search(&handle_hash) {
            Ok(_) => {
                return Err(AclError::HandleAlreadyExists.into());
            }
            Err(pos) => {
                current_storage.handle_hashes.insert(pos, handle_hash);
            }
        }
        master.total_handle_hashes += 1;
        if current_storage.handle_hashes.len() >= 100 {
            current_storage.is_full = true;
            master.current_acl_storage_index += 1;
        }
        // Only the first handle is processed due to this return
        return Ok(());
    }

```

### Impact
Only the first handle in the batch is processed; the rest are ignored.
- Users expecting all handles in the batch to be granted access will not have their requests fulfilled.
- This breaks the batch access functionality and may lead to confusion or failed operations for users and integrators.

### Recommendation

Replace the `return Ok(())` statement inside the loop with a `continue;` statement. This allows the loop to process all handles in the batch as intended.

```rust
for handle in handles {
    let handle_hash = get_handle_hash(ctx.accounts.payer.key(), handle);
    // ...
    if current_storage.handle_hashes.len() < 100 {
        match current_storage.handle_hashes.binary_search(&handle_hash) {
            Ok(_) => {
                return Err(AclError::HandleAlreadyExists.into());
            }
            Err(pos) => {
                current_storage.handle_hashes.insert(pos, handle_hash);
            }
        }
        master.total_handle_hashes += 1;
        if current_storage.handle_hashes.len() >= 100 {
            current_storage.is_full = true;
        }
        // Use continue to process the next handle in the batch
        continue;
    }
    // ...
}
```

This change ensures that all handles in the batch are processed

### Specific Fixed In Commit

NA

### Encifher team's Response

We are no longer maintaining ACL onchain

## Faulty Conditional on Storage Index Prevents Proper Storage Initialization and Causes DoS

Acknowledged

### Path

ACL

### Function

**`Most of the functions`**

### Description

The ACL contract uses the following condition to determine when to initialize a new storage account and set its index:

```rust
if new_storage.handle_hashes.is_empty() && new_storage.index == 0 {
    new_storage.index = master.current_acl_storage_index + 1;
    new_storage.is_full = false;
    master.current_acl_storage_index += 1;
}
```

However, when an admin creates a new storage account using the `initialize_acl_storage` instruction, the `index` is set by the admin and is typically non-zero. As a result, the above conditional (`new_storage.index == 0`) is never true for admin-created storage accounts. This prevents the program from ever entering the block that sets the new storage's index and increments `master.current_acl_storage_index`.

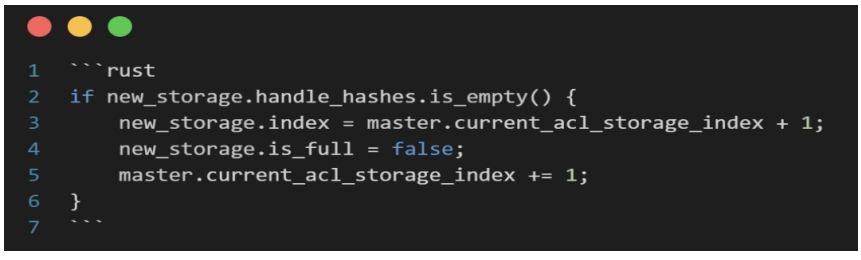
This logic flaw means the program cannot properly propagate or update storage indices, and the new storage account is never initialized by the program as intended.

### Impact

- Denial of Service (DoS): The program never enters the block to initialize new storage accounts, breaking the flow for adding new access entries once the first storage is full.
- Index Propagation Failure: The `master.current_acl_storage_index` is not updated, and new storage accounts' indices are not set by the program, leading to inconsistencies.
- Revocation and Access Checks Break: Functions like `revoke_access` that rely on correct storage indexing may fail, as the index is never set or incremented properly.

### Recommendation

Remove the `new_storage.index == 0` check from the conditional. The program should not rely on the admin to set the storage index. Instead, always allow the program to initialize the storage account's index when it is first used.



```rust
if new_storage.handle_hashes.is_empty() {
    new_storage.index = master.current_acl_storage_index + 1;
    new_storage.is_full = false;
    master.current_acl_storage_index += 1;
}
```

This ensures that the program can always initialize and propagate storage indices correctly, regardless of how the storage account was created. Apply this fix to all relevant instructions that handle the storage account

### Specific Fixed In Commit

NA

### Encifher team's Response

We are no longer maintaining ACL onchain

## Broken freeze/thaw functionality

Resolved

### Path

ACL

### Function

`process_freeze_account`

### Description

The `process_freeze_account` instruction in the `etoken` program allows authorized users to freeze token accounts, setting `is_frozen = true`. However, there is no corresponding instruction or mechanism to thaw (unfreeze) a token account. Once an account is frozen, it is permanently locked, and users cannot transfer, mint, or burn tokens from it

```rust
pub fn process_freeze_account(ctx: Context<FreezeAccount>) -> Result<()> {
    let token_account = &mut ctx.accounts.token_account;
    let mint = &ctx.accounts.emint;

    require!(
        mint.freeze_authority == Some(ctx.accounts.authority.key()),
        CustomError::InvalidFreezeAuthority
    );

    token_account.is_frozen = true;
    Ok(())
}
```

### Impact

- Permanent loss of access: Once frozen, a token account cannot be unfrozen, leading to permanent loss of access to tokens.
- No recovery from mistakes: Accidental or malicious freezing cannot be undone, even by the freeze authority.

### Recommendation

Add thaw functionality: Modify the freeze instruction to accept a boolean flag (e.g., `freeze: bool`) that determines whether to freeze or thaw the account. If `freeze` is `true`, set `is_frozen = true`; if `false`, set `is_frozen = false`.

```rust
pub fn process_freeze_account(ctx: Context<FreezeAccount>, freeze: bool) -> Result<()> {
    let token_account = &mut ctx.accounts.token_account;
    let mint = &ctx.accounts.emint;

    require!(
        mint.freeze_authority == Some(ctx.accounts.authority.key()),
        CustomError::InvalidFreezeAuthority
    );

    token_account.is_frozen = freeze;
    Ok(())
}
```

This ensures that frozen accounts can be recovered and tokens are not permanently lost

### Specific Fixed In Commit

https://github.com/RizeLabs/encifher-svm-contracts/commit/5b0715f3e4dda8373e84cd316f572ea87d838f55

### Encifher team's Response

Added recommended fix

## Incorrect Program ID Used for Mint Authority Derivation Causes Permanent DoS in Minting

Acknowledged

### Path
ACL

### Function
`process_mint_to`

### Description
A mismatch in the program ID used for PDA derivation causes the mint authority check to always fail, resulting in a permanent DoS for minting. In the `process_mint_to` function of the `etoken` program, the mint authority is derived using a hardcoded program ID:

```rust
let (expected_authority, _bump) = Pubkey::find_program_address(
    &[b"ewrapper"],
    // @audit wrong program Id, should have been id of wrapper program
    &pubkey!("HutgaHDMVeh1jk9tMq4syaXafJp6yyhMDSVdekLMHrPE"),
);
```

However, the correct program ID should be the one declared in the wrapper program:

```rust
declare_id!("FCVq1PPmUw9Rqpxs1aSq7n9AMP8ExQixDrFVM5k8X8J3");
```

The deposit flow in `ewrapper` calls the mint instruction and passes `ctx.accounts.program_account.to_account_info()` as the authority, which is derived using the correct wrapper program's ID and the `ewrapper` seed.

But since `process_mint_to` expects the authority to be derived using the wrong program ID, the check will always fail:

```rust
require!(
    ctx.accounts.authority.key() == expected_authority,
    CustomError::InvalidMintAuthority
);
```

This causes a permanent denial of service (DoS) for minting, as the authority will never match.

### Impact

Permanent DoS: No user will ever be able to mint tokens via the deposit flow, as the authority check will always fail.
Breaks all integrations: Any protocol or user relying on minting will be unable to use the system.

### Recommendation

Replace the hardcoded program ID with the actual wrapper program's ID when deriving the expected authority:

```rust
    let (expected_authority, _bump) = Pubkey::find_program_address(
        &[b"ewrapper"],
        // use the correct wrapper program's ID
    );
```

Ensure consistency: Always use the same seeds and program ID for PDA derivation in both the caller (`ewrapper`) and callee (`etoken`) programs.

### Specific Fixed In Commit

**NA**

### Encifher team's Response

The hardcoded address is our mainnet deployed program address, so it does not need to match the devnet one.

## Incorrect Handle Hash Inserted in `delegate_access` When Current Storage Is Full

Acknowledged

### Path

ACL

### Function

`delegate_access`

### Description

In the `delegate_access` function, when the current storage account is full (`handle_hashes.len() >= 100`), the program is supposed to insert the grantee's handle hash into the new storage account. However it mistakenlly, it inserts the payer's handle hash instead:

```rust
let handle_hash = get_handle_hash(ctx.accounts.payer.key(), handle);
// ...
let grantee_handle_hash = get_handle_hash(grantee, handle);
// ...
if current_storage.handle_hashes.len() < 100 {
    // ...correctly uses grantee_handle_hash...
} else {
    // Incorrect: inserts payer's handle hash instead of grantee's
    new_storage.handle_hashes.push(handle_hash);
}
```

This means that, after the first storage account is full, delegated access is not properly granted to the intended grantee.

### Impact

- Delegated access is not granted to the intended grantee when the current storage account is full.
- Access control is broken for delegated access, as the wrong user's handle hash is stored.
- Potential denial of service for users expecting delegated access, as their access will not be recognized by the system.

### Recommendation

Updatethe code in the `delegate_access` function to always insert the grantee's handle hash, regardless of whether the current or new storage account is being used.

```rust
let handle_hash = get_handle_hash(ctx.accounts.payer.key(), handle);
let grantee_handle_hash = get_handle_hash(grantee, handle);
// ...
if current_storage.handle_hashes.len() < 100 {
    // ...existing logic...
    current_storage.handle_hashes.insert(pos, grantee_handle_hash);
    // ...
} else {
    // Correct: insert grantee's handle hash
    new_storage.handle_hashes.push(grantee_handle_hash);
}
```

This ensures that delegated access is always granted to the correct grantee.

### Specific Fixed In Commit

NA

### Encifher team's Response

The hardcoded address is our mainnet deployed program address, so it does not need to match the devnet one.

## Incorrect Storage Index Increment Prevents Proper Storage Account Initialization and Breaks Access Revocation

Acknowledged

### Path

ACL

### Function

**`Most of the mentioned functions`**

### Description

In the ACL contract, the logic for incrementing `current_acl_storage_index` in the `grant_access_account`, `grant_access_general`, `delegate_access`, and `grant_batch_access` instructions is flawed. The index is incremented immediately after the current storage account reaches its maximum capacity (100 entries), rather than after a new storage account is actually initialized and used. This causes all subsequent access grants to operate on a newly derived, empty storage account, but the new storage's `index` field is never set. As a result, the system never properly initializes or uses new storage accounts, and the `index` field remains unset.

```rust
if current_storage.handle_hashes.len() < 100 {
    // ...insert logic...
    master.total_handle_hashes += 1;

    if current_storage.handle_hashes.len() >= 100 {
        current_storage.is_full = true;
        //@audit don't do this here, otherwise execution never reach passt this if block, new storage's index can't be set
        master.current_acl_storage_index += 1;
    }
    return Ok(());
}

// ...new storage logic...
if new_storage.handle_hashes.is_empty() && new_storage.index == 0 {
    new_storage.index = master.current_acl_storage_index + 1;
    new_storage.is_full = false;//Increase here only
    master.current_acl_storage_index += 1;
}
```

granted to the intended grantee.

### Impact

- New storage accounts are never properly initialized: The `index` field of new storage accounts is never
- set, except for the first one.
- Subsequent access grants always operate on empty storage: The logic keeps using a new, empty storage account, never filling or indexing it correctly.
- Revocation and access checks break: Functions like `revoke_access` that rely on correct storage indexing will fail, as the index is never set or incremented properly.

### Recommendation

Increment `current_acl_storage_index` only after initializing and using the new storage account.

Move the increment logic from the current storage branch to the new storage initialization branch. This ensures that the index is only incremented when a new storage account is actually created and used.

```
```rust
if current_storage.handle_hashes.len() < 100 {
    match current_storage.handle_hashes.binary_search(&handle_hash) {
        Ok(_) => {
            return Err(AclError::HandleAlreadyExists.into());
        }
        Err(pos) => {
            current_storage.handle_hashes.insert(pos, handle_hash);
        }
    }
    master.total_handle_hashes += 1;

    if current_storage.handle_hashes.len() >= 100 {
        current_storage.is_full = true;
        // DO NOT increment index here!
    }
    return Ok(());
}

// ...new storage logic...
if new_storage.handle_hashes.is_empty() && new_storage.index == 0 {
    new_storage.index = master.current_acl_storage_index + 1;
    new_storage.is_full = false;
    master.current_acl_storage_index += 1; // Increment index only here
}

new_storage.handle_hashes.push(handle_hash);
master.total_handle_hashes += 1;
```

**Apply this fix to all relevant instructions:**
- `grant_
```

Apply this fix to all relevant instructions:
- `grant_access_account`
- `grant_access_general`
- `delegate_access`
- `grant_batch_access`

This ensures that storage accounts are properly initialized, indexed, and used, and that access revocation and other features relying on correct indexing will work as intended.

### Specific Fixed In Commit

**NA**

### Encifher team's Response

We are no longer maintaining ACL onchain

# Low Severity Issues

## Unnecessary data allocation for liquidity vaultManual Initialization of Storage Accounts Increases Operational Overhead

Acknowledged

### Path

acl

### Function

`initialize_acl_storage`

### Description

Currently, the program requires the admin to manually invoke the `initialize_acl_storage` instruction to create new storage accounts as the ACL grows. This approach adds unnecessary operational complexity and increases the risk of errors if the admin forgets to initialize a required storage account before granting access.

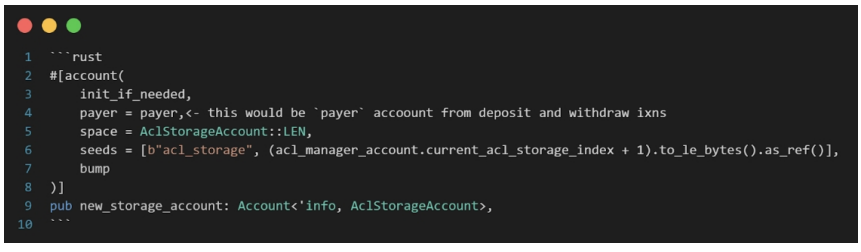
```rust
// Admin must call this before storage is used
pub fn initialize_acl_storage(
    ctx: Context<InitializeAclStorage>,
    acl_storage_index: u64,
) -> Result<()> {
    let storage = &mut ctx.accounts.storage_account;
    storage.index = acl_storage_index;
    storage.handle_hashes = Vec::new();
    storage.is_full = false;
    Ok(())
}
```

### Impact

- Increased operational overhead for administrators.
- Potential for failed transactions if a required storage account is not initialized in advance.
- Less user-friendly contract interface.

### Remediation

Use Anchor's `init_if_needed` constraint in the `GrantAccess`-related instructions to automatically initialize new storage accounts when required. This allows the contract to create storage accounts on-the-fly, reducing manual intervention and improving usability.



```rust
#[account(
    init_if_needed,
    payer = payer,<- this would be `payer` accoount from deposit and withdraw ixns
    space = AclStorageAccount::LEN,
    seeds = [b"acl_storage", (acl_manager_account.current_acl_storage_index + 1).to_le_bytes().as_ref()],
    bump
)]
pub new_storage_account: Account<'info, AclStorageAccount>,
```

Apply this change to all relevant instructions:
- `grant_access_account`
- `grant_access_general`
- `delegate_access`
- `grant_batch_access`

One more thing to keep in mind is that when calling the `MintTo` instruction from `deposit.rs` and `withdraw.rs`, you should pass the payer account from those instructions into `MintTo`. Then, in `MintTo`, use this payer as the payer for any CPI to the ACL contract (for example, when using `init_if_needed` to create a new storage account). This we mitigate the risk of garbage collection of `program account(pub program_account: Account<'info, WrapperAccount>,)`(if payer not used to init new account and this account is used instead)

### Specific Fixed In Commit

**NA**

### Encifher team's Response

We are no longer maintaining ACL onchain

## Lack of Pausing Functionality Exposes Protocol to Irrecoverable Exploits

Resolved

### Path

program

### Function

`Global level`

### Description

The protocol currently lacks a pausing mechanism that would allow administrators to temporarily halt critical operations (such as minting, burning, transferring, depositing, or withdrawing tokens) in the event of a detected vulnerability, exploit, or emergency. Without this feature, if a bug or attack is discovered, there is no way to quickly stop protocol activity to prevent further damage or loss of funds.

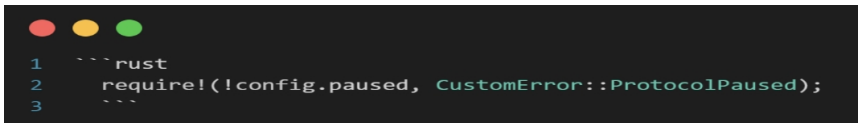
A pausing mechanism is a standard best practice in DeFi and token protocols, providing a crucial layer of operational security and incident response.

### Impact

-No emergency response: If a critical bug or exploit is discovered, the protocol cannot be paused, allowing attackers to continue exploiting the vulnerability until a full upgrade is deployed.
-Potential for catastrophic loss: Funds can be drained or irreversibly lost before a fix can be implemented.

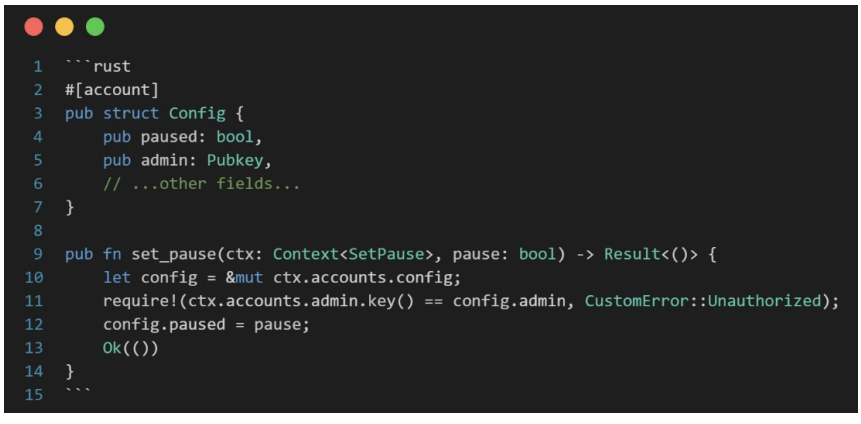
### Recommendation

- Implement a global pause mechanism:
  Add a `paused` boolean flag to a central config or admin account (e.g., `WrapperAccount` or a new `Config` account).
- Gate all critical instructions:
  Add a check at the start of each sensitive instruction (mint, burn, transfer, deposit, withdraw, etc.):



- Admin control:

Only allow a trusted admin or governance authority to toggle the pause state.



```rust
#[account]
pub struct Config {
    pub paused: bool,
    pub admin: Pubkey,
    // ...other fields...
}

pub fn set_pause(ctx: Context<SetPause>, pause: bool) -> Result<()> {
    let config = &mut ctx.accounts.config;
    require!(ctx.accounts.admin.key() == config.admin, CustomError::Unauthorized);
    config.paused = pause;
    Ok(())
}
```

Update all entrypoints: At the start of each instruction, check the `paused` flag and revert if the protocol is paused.

### Specific Fixed In Commit

**https://github.com/RizeLabs/encifher-svm-contracts/commit/5b0715f3e4dda8373e84cd316f572ea87d838f55**

### Encifher team's Response

Added global halting logic as recommended

# Functional Tests

## etoken

### Mint Initialization & Account Creation

- ✓ Test successful mint initialization with valid authority and decimals
- ✓ Test only admin can initialize mint (admin gating)
- ✓ Test mint initialization fails if already initialized
- ✓ Test token account initialization for valid owner
- ✓ Test token account cannot be initialized twice
- ✓ Test correct PDA derivation for mint and token accounts
- ✓ Test rent exemption for mint and token accounts

### Minting & Burning

- ✓ Test successful minting to unfrozen token accounts
- ✓ Test minting fails if supply exceeds max supply
- ✓ Test supply increases correctly on mint
- ✓ Test successful burning from own token account

### Transfers & Freezing

- ✓ Test successful transfer between unfrozen accounts
- ✓ Test transfer fails if sender or receiver is frozen
- ✓ Test freeze authority can freeze token accounts
- ✓ Test freeze authority cannot freeze if not authorized
- ✓ Test thaw functionality (if implemented) restores account usability

### Access Control

- ✓ Test ACL storage initialization and automatic creation
- ✓ Test granting, delegating, and revoking access
- ✓ Test batch access grants
- ✓ Test correct handle hash propagation and indexing

### ewrapper

#### Deposit & Mint

- ✓ Test successful deposit and mint flow
- ✓ Test deposit fails if token account mint mismatch
- ✓ Test correct transfer of tokens to program vault
- ✓ Test correct minting of etokens to user account
- ✓ Test rent exemption for vault and wrapper accounts

#### Withdraw & Burn

- ✓ Test successful burn and withdraw flow
- ✓ Test withdraw fails if user has insufficient etokens
- ✓ Test correct transfer of tokens from vault to user

#### Admin & Config

- ✓ Test wrapper initialization with valid relayer
- ✓ Test only relayer can perform admin actions

### swap_om

#### Order Management

- ✓ Test successful order manager initialization
- ✓ Test order manager fails to initialize twice
- ✓ Test placing orders with valid parameters
- ✓ Test placing orders fails if not enough liquidity
- ✓ Test correct transfer of etokens during order placement
- ✓ Test correct PDA derivation for order manager and program accounts

#### ACL Integration

- ✓ Test ACL access checks for order placement
- ✓ Test correct storage and indexing of order handles
- ✓ Test batch order placement and access propagation

# Threat Model

| Contract | Function | Threats |
|---|---|---|
| etoken | initialize_mint | - Unrestricted access allows anyone to initialize mints with arbitrary decimals and freeze authority (see H-01)<br>- Decimals manipulation can cause DoS or break token economics<br>- Freeze authority hijack can permanently freeze user accounts<br>- No admin gating for critical mint parameters |
| | mint_to | - Incorrect program ID for mint authority derivation causes permanent DoS (see M-06)<br>- Minting allowed to frozen token accounts, leading to permanent token lock (see H-02)<br>- No check for pausing; protocol cannot be halted in emergencies<br>- Supply cap enforced, but decimals can be manipulated if not fetched from SPL mint<br>Function: burn<br>- No ownership check allows anyone to burn tokens from any account (see C-03)<br>- Burning from frozen accounts allowed, bypassing freeze protection (see H-04)<br>- Supply not decreased on burn, causing inaccurate supply and DoS (see H-03) |

| Contract | Function | Threats |
|---|---|---|
| | etransfer | - Duplicate mutable accounts bug allows arbitrary balance inflation (see C-01)<br>- No enforcement that to is program token account, enabling repeated unauthorized withdrawals (see C-02)<br>- No check for pausing; protocol cannot be halted in emergencies |
| | freeze_account | - No thaw functionality; frozen accounts are permanently locked (see M-07)<br>- Freeze authority can freeze any account, but cannot reverse |
| ewrapper | deposit_and_mint / deposit_and_mint_2022 | - No pausing mechanism; protocol cannot be halted in emergencies (see L-02)<br>- Incorrect payer usage can cause garbage collection of PDAs if not handled properly (see L-01)<br>- Program token account must be enforced for vaults<br>Function: burn_and_withdraw<br>- No pausing mechanism; protocol cannot be halted in emergencies<br>- Withdrawals can be abused if etransfer does not enforce program token account<br>Function: admin/init<br>- No admin gating for critical config changes<br>- Relayer can be set arbitrarily if not checked |

| Contract | Function | Threats |
| --- | --- | --- |
| swap_om | initialize | - No admin gating for order manager initialization<br>- Owner can be set arbitrarily<br>Function: place_order<br>- No pausing mechanism; protocol cannot be halted in emergencies<br>- Order placement relies on ACL and etoken correctness<br>- No check for order limits or liquidity exhaustion |
| acl | grant_access_account / grant_access_general / delegate_access / grant_batch_access | - Incorrect index increment and faulty conditional can break access propagation and revocation (see M-01, M-02, M-04)<br>- Early return in batch access causes only first handle to be processed (see M-04)<br>- Mismatched parameter names between instruction and account struct can cause DoS (see M-05)<br>- Manual storage initialization increases operational overhead and risk (see L-01)<br>Function: revoke_access<br>- Relies on correct index propagation; broken if index is never set |

# Automated Tests

No major issues were found. Some false-positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

# Closing Summary

In this report, we have considered the security of Encifher. We performed our audit according to the procedure described above.

Issues of Critical, High, Medium and Low severities were found. Encifher team resolved few issues and acknowledged others.

# Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

# About QuillAudits

QuillAudits is a leading name in Web3 security, offering top-notch solutions to safeguard projects across DeFi, GameFi, NFT gaming, and all blockchain layers. With seven years of expertise, we've secured over 1400 projects globally, averting over $3 billion in losses. Our specialists rigorously audit smart contracts and ensure DApp safety on major platforms like Ethereum, BSC, Arbitrum, Algorand, Tron, Polygon, Polkadot, Fantom, NEAR, Solana, and others, guaranteeing your project's security with cutting-edge practices.


| | |
|---|---|
| **7+** Years of Expertise | **1M+** Lines of Code Audited |
| **50+** Chains Supported | **1400+** Projects Secured |

**Follow Our Journey**

# AUDIT REPORT

September 2025

For

ENCIFHER

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com