fuzzland

# Cytonic Audit Report

Signature: 048ab315eca35803548cb08829bc83e0cdd08dfd8bc28672c9fc670301771847

# Cytonic Audit Report

## Executive Summary

From Oct 4, 2024, to Oct 10, 2024, the Cytonic team engaged Fuzzland to conduct a thorough security audit of their bridge project. The primary objective was to identify and mitigate potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Fuzzland conducted this assessment over 10 person-days, involving 2 engineers who reviewed the code over a span of 5 days. Employing a multifaceted approach that included static analysis, fuzz testing, formal verification, and manual code review, the Fuzzland team identified 10 issues across different severity levels and categories.

## Scope

| | |
|---|---|
| **Project Name** | Cytonic Bridge EVM |
| **Repository** | cytonic-bridge-evm |
| **Commit** | d07736c87afa20bea192b58a65ce47c6757b5730 |
| **Fix Commit** | 15593a0ad49d5793ed778ab4dc574ff926747215 |
| **Language** | Solidity |
| **Scope** | **/*.sol |

| | |
|---|---|
| **Project Name** | Cytonic Bridge Solana |
| **Repository** | cytonic-bridge-solana |
| **Commit** | 61cc8622154cc0925795942ed8f9773551ab19b4 |
| **Fix Commit** | 106dcf395c387e6be06542be09dd906887502a59 |
| **Language** | Rust - Anchor (Solana) |
| **Scope** | programs/**/*.rs |

## Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

## Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.
- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry and ItyFuzz, to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

## Vulnerability Severity

We divide severity into four distinct levels: high, medium, low, and info. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.
- **Informational Severity Issues** represent informational findings that do not directly impact the system's security, functionality, or performance. These findings are typically observations or recommendations for potential improvements or optimizations. Addressing info severity issues can enhance the system's robustness and efficiency but is not necessary for the system's immediate operation or security. These issues can be considered for future development or enhancement plans.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

| | Number | Resolved |
|---|---|---|
| **High Severity Issues** | 0 | 0 |
| **Medium Severity Issues** | 0 | 0 |
| **Low Severity Issues** | 4 | 4 |
| **Informational Severity Issues** | 6 | 6 |

## Findings

### [Low] `migrate` uses wrong address

In the `migrate` function, the contract owner uses `msg.sender` as the recipient address. This may result in assets being transferred to the wrong address, leading to asset loss or incorrect transfers.

The contract owner incorrectly sets the to parameter to `msg.sender` when calling migrate, resulting in assets being transferred to the wrong address.

```
function migrate(address asset, address to) external onlyOwner whenPaused {
    if (asset == address(0)) {
        payable(to).transfer(address(this).balance);
    } else {
        IERC20(asset).safeTransfer(msg.sender,
IERC20(asset).balanceOf(address(this)));
    }
}
```

**Recommendation:**

It is recommended to replace `msg.sender` with `to` to ensure that assets are correctly transferred to the specified recipient address.

```
function migrate(address asset, address to) external onlyOwner whenPaused {
    if (asset == address(0)) {
        payable(to).transfer(address(this).balance);
    } else {
        IERC20(asset).safeTransfer(to,
IERC20(asset).balanceOf(address(this))); // Fix: Use to instead of msg.sender
    }
}
```

**Status**: Fixed

## [Low] `Depositor::migrate` lacks zero address check

Since this function is transferring funds, a 0 address check is required to prevent the funds from being transferred to the 0 address by mistake when transferring ETH funds.

```
function migrate(address asset, address to) external onlyOwner whenPaused {
    if (asset == address(0)) {
        payable(to).transfer(address(this).balance);
    } else {
        IERC20(asset).safeTransfer(msg.sender,
IERC20(asset).balanceOf(address(this)));
    }
}
```

**Recommendation:**

Add a zero address check.

**Status**: Fixed

## [Low] The lack of restrictions on `migrate` may cause funds to be stuck due to `unlockPeriod` after the user `withdraw`

The `migrate` function of the contract allows the contract owner to migrate assets when the contract is paused. If the contract is paused and the assets are migrated during the `unlockperiod` of the user's withdrawal request, the user will not be able to withdraw the assets when calling claim, but the data in the `usersClaims` mapping still exists, resulting in the user being unable to claim.

```
function migrate(address asset, address to) external onlyOwner whenPaused {
    if (asset == address(0)) {
        payable(to).transfer(address(this).balance);
    } else {
        IERC20(asset).safeTransfer(msg.sender,
IERC20(asset).balanceOf(address(this)));
    }
}
```

```
function withdraw(address asset, uint128 id, uint256 amount) external
whenNotPaused {
    if (!allowedAssets[asset]) revert AssetNotAllowed();
    _withdraw(asset, amount, msg.sender);

    if (usersClaims[id].user != address(0)) revert ExistingClaimId();
    uint256 claimableAfter = block.timestamp + unlockPeriod;
    usersClaims[id] = ClaimData(msg.sender, asset, claimableAfter, amount);
    emit Withdraw(asset, msg.sender, id, amount, claimableAfter,
block.timestamp);
}
```

**Recommendation:**

Before executing `migrate`, if the current user's `usersClaims` is not 0, cancel the user's withdrawal request, or advance the user's claim.

**Status:** Acknowledged

## [Low] `Migrate::actuate` does not check if the `VaultData` account is frozen

In other functions related to `VaultData` accounts, the `is_frozen` field is always checked. Yet in `Migrate::actuate` this field is not checked.

```
impl Migrate<'_> {
    pub fn actuate(ctx: &mut Context<Self>, _params: &MigrateParams) ->
Result<()> {
        spl_transfer(
            CpiContext::new_with_signer(
                ctx.accounts.token_program.to_account_info(),
                SplTransfer {
                    authority: ctx.accounts.vault_data.to_account_info(),
                    from: ctx.accounts.vault_token_account.to_account_info(),
                    to: ctx.accounts.sender_token_account.to_account_info(),
                },
                &[&[
                    b"vault-data".as_ref(),
                    &ctx.accounts.vault_data.admin.to_bytes(),
                    &ctx.accounts.vault_data.mint.to_bytes(),
                    &[ctx.accounts.vault_data.bump],
                ]],
            ),
            ctx.accounts.vault_token_account.amount,
        )?;
        Ok(())
    }
}
```

**Recommendation**:

```
if !ctx.accounts.vault_data.is_frozen {
    return Err(error!(Errors::IsFrozen));
}
```

**Status**: Fixed

## [Info] Not emit events

The `toggleAsset` and `togglePurchaseAsset` functions are used to enable or disable the deposit and purchase functionality for assets, respectively. However, these functions do not emit any events. This makes it impossible for external systems or users to track state changes, reducing the transparency and auditability of the contract.

Other modifications to key parameters also lack corresponding events

```
    function toggleAsset(address asset) external onlyOwner {
        allowedAssets[asset] = !allowedAssets[asset];
    }

    function updateUnlockPeriod(uint256 _unlockPeriod) external onlyOwner {
        unlockPeriod = _unlockPeriod;
    }

    function togglePurchaseAsset(address asset) external onlyOwner {
        allowedPurchaseAssets[asset] = !allowedPurchaseAssets[asset];
    }

    function updateTreasury(address _treasury) external onlyOwner {
        treasury = _treasury;
    }
```

**Recommendation**:

Consider defining and emitting events whenever sensitive changes occur.

**Status**: Fixed

## [Info] No need to check allowedAssets when withdrawing

Since the `if (!allowedAssets[asset])` check has already been done during deposit, there is no need to check again during withdrawal.

```
function withdraw(address asset, uint128 id, uint256 amount) external
whenNotPaused {
    if (!allowedAssets[asset]) revert AssetNotAllowed();
    _withdraw(asset, amount, msg.sender);

    if (usersClaims[id].user != address(0)) revert ExistingClaimId();
    uint256 claimableAfter = block.timestamp + unlockPeriod;
    usersClaims[id] = ClaimData(msg.sender, asset, claimableAfter, amount);
    emit Withdraw(asset, msg.sender, id, amount, claimableAfter,
block.timestamp);
}
```

**Recommendation:**

Delete `if (!allowedAssets[asset]) revert AssetNotAllowed();`

**Status:** Fixed

## [Info] Upgradeable contract is missing a `__gap` storage variable to allow for new storage variables in later versions

See this link for a description of this storage variable. While some contracts may not currently be sub-classed, adding the variable now protects against forgetting to add it in the future.

**Recommendation**:

It is considered a best practice in upgradeable contracts to include astate variable named `__gap`. This `__gap` state variable will be used as a reserved space for future upgrades. It allows adding new state variables freely in the future without compromising the storage compatibility with existing deployments. The size of the `__gap` array is usually calculated so that the amount of storage used by a contract always adds up to the same number (usually 50 storage slots).

**Status**: Fixed

## [Info] Uses `call` Instead of `transfer`

The contracts sends ETH using the `transfer` method (at most 2300 gas) instead of the safer `call` method. If receive address is a contract, then this transfer may fall.

```
src/Depositor.sol:
155          if (claimOrder.asset != address(0))
IERC20(claimOrder.asset).safeTransfer(to, claimOrder.amount);
156:         else payable(to).transfer(claimOrder.amount);
157          emit Claim(id, false, block.timestamp);

178          if (asset != address(0)) IERC20(asset).safeTransfer(treasury,
amount);
179:         else payable(treasury).transfer(amount);
180          emit Purchase(asset, msg.sender, amount, true, block.timestamp);

194          if (!allowedPurchaseAssets[address(0)]) revert
AssetNotAllowedForPurchase();
195:         payable(treasury).transfer(msg.value);
196          emit Purchase(address(0), msg.sender, msg.value, false,
block.timestamp);

220          if (asset == address(0)) {
221:             payable(to).transfer(address(this).balance);
222          } else {
```

**Recommendation:**
Replace the `transfer` method with the `call` method and ensure proper handling of the return value. For example:

```
(bool success, ) = payable(_treasury).call{value: msg.value}("");
require(success, "ETH transfer failed");
```

**Status**: Fixed

## [Info] `freeze` naming convention

In the Solana program, the `freeze` function actually toggles the frozen flag for a `VaultData` account.

```
/// This method pauses all non authorized contract actions
pub fn freeze(mut ctx: Context<Freeze>, params: FreezeParams) -> Result<()> {
    Freeze::actuate(&mut ctx, &params)
}

impl Freeze<'_> {
    pub fn actuate(ctx: &mut Context<Self>, _params: &FreezeParams) ->
Result<()> {
        let vault_data = &mut ctx.accounts.vault_data;
        vault_data.is_frozen = !vault_data.is_frozen;
        Ok(())
    }
}
```

The code has same pattern with `toggle_purchase`, so it's recommended to rename this function to follow `toggle_*` naming convention.

**Status**: Fixed

## [Info] Centralization risk

The owner permission in the contract can transfer the funds in the contract to the specified address through the migrate function in any state, and the funds also include the user's staked funds.

```rust
impl Migrate<'_> {
    pub fn actuate(ctx: &mut Context<Self>, _params: &MigrateParams) ->
Result<()> {
        spl_transfer(
            CpiContext::new_with_signer(
                ctx.accounts.token_program.to_account_info(),
                SplTransfer {
                    authority: ctx.accounts.vault_data.to_account_info(),
                    from: ctx.accounts.vault_token_account.to_account_info(),
                    to: ctx.accounts.sender_token_account.to_account_info(),
                },
                &[&[
                    b"vault-data".as_ref(),
                    &ctx.accounts.vault_data.admin.to_bytes(),
                    &ctx.accounts.vault_data.mint.to_bytes(),
                    &[ctx.accounts.vault_data.bump],
                ]],
            ),
            ctx.accounts.vault_token_account.amount,
        )?;
        Ok(())
    }
}
```

There is no range limit for the withdrawal period. When `withdraw_duration` is 0 or a very large value, there may be certain risks.

```rust
impl ChangeWithdrawDuration<'_> {
    pub fn actuate(ctx: &mut Context<Self>, params:
&ChangeWithdrawDurationParams) -> Result<()> {
        ctx.accounts.vault_data.withdraw_duration =
params.new_withdraw_duration;
        Ok(())
    }
}
```

**Recommendation:**

It is recommended to use a multi-signature wallet or other methods to control the risk of single account failure.

**Status**: Acknowledged