// Security Assessment

03.25.2026 - 03.27.2026

USX Solana Program

Solstice Labs

HALBORN

USX Solana Program - Solstice Labs

Prepared by: HALBORN

Last Updated 03/31/2026

Date of Engagement: March 25th, 2026 - March 27th, 2026

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	0	0	0	2

TABLE OF CONTENTS

1. Introduction

`Solstice Labs` engaged Halborn to conduct a security assessment on their USX Solana programs beginning on March 25th, 2026, and ending on March 27th, 2026. The security assessment was scoped to the Updated freeze token functionality Solana Programs provided in usx-program GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The `USX` Program enables authorized users to mint and redeem the USX stablecoin (referred to as `Redeemable`) in exchange for a `Collateral` token. It supports multiple types of collateral and follows a two-step process for both minting and redeeming : request and confirmation.

- To mint USX, a user deposits collateral into an escrow account during the request phase. In the confirmation step, the collateral is transferred to a program-owned token account, and the corresponding amount of USX is minted to the user's wallet.
- The redeem process works similarly in reverse. The user sends USX to an escrow account in the request phase. During confirmation, the USX is burned, and the equivalent collateral is returned to the user's wallet.

The program also includes instructions for managing authorizations, configurations, collateral types, and funds. Moreover, the `USX` program now also supports Token-2022 mints.

2. Assessment Summary

`Halborn` was provided 2 days for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Solana Programs.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified two minor improvements to reduce the likelihood and impact of risks, which have been acknowledged by the `Solstice Labs` team. The findings were the following:

- `freeze_token relies on raw SPL Token failures for repeated freeze and thaw attempts.`
- `Token-2022 extension relaxation expands accepted collateral semantics and shifts safety to governance.`

3. Verified Program Build Hashes

The following program artifacts for the commit (`a940fa`) were independently reproduced and verified using a verifiable build process (`anchor build --verifiable`).

- `programID`: USXyiSTsPEWz55pSK7sZoUL79ntoVGQbaTDT57tH6bx
- `sha`: f7c99b7ca1cd3e545cfbd2e623cece54156d41b9254dee8672e23a0cf3c1f3ee
- `checksum`: b6f5881bc078f4f870692a0276ed270fe2b19b14989d8db41eaef24d4481c513

4. Test Approach And Methodology

`Halborn` performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities `cargo audit`).
- Local runtime testing `anchor test`)

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

REPOSITORY

(a) Repository: usx-program

(b) Assessed Commit ID: a940fa4

(c) Items in scope:

- programs/usx/src/events.rs
- programs/usx/src/instructions/edit_program_admin.rs
- programs/usx/src/instructions/freeze_token.rs
- programs/usx/src/instructions/initialize_program_admin.rs
- programs/usx/src/instructions/mod.rs
- programs/usx/src/lib.rs
- programs/usx/src/state/program_admin.rs
- programs/usx/src/utils/validate_token_extension.rs
- tests/src/api/program_usx/process_freeze_token.rs

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
TOKEN-2022 EXTENSION RELAXATION EXPANDS ACCEPTED COLLATERAL SEMANTICS AND SHIFTS SAFETY TO GOVERNANCE	INFORMATIONAL	ACKNOWLEDGED - 03/27/2026

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
FREEZE_TOKEN RELIES ON RAW SPL TOKEN FAILURES FOR REPEATED FREEZE AND THAW ATTEMPTS	INFORMATIONAL	ACKNOWLEDGED - 03/27/2026

8. FINDINGS & TECH DETAILS

8.1 TOKEN-2022 EXTENSION RELAXATION EXPANDS ACCEPTED COLLATERAL SEMANTICS AND SHIFTS SAFETY TO GOVERNANCE

// INFORMATIONAL

Description

The updated `validate_token_extension` logic now rejects only `InterestBearingConfig` and no longer blocks `TransferFeeConfig`, `TransferHook`, `PermanentDelegate`, `ConfidentialTransferMint`, `ConfidentialTransferFeeConfig`, `MintCloseAuthority`, or `NonTransferable` on the code level.

Even though this relaxation is intentional and protected in off-chain setting, it still worth documenting as an informational configuration-risk change, because future onboarding of complex Token-2022 collateral may have side effects that the original stricter filter would have prevented by default.

Code Location

In the `programs/usx/src/utils/validate_token_extension.rs`, the disallow list has been commented out as per the commit in scope:

Copy Code

```
const DISALLOWED: [ExtensionType; 1] = [
    // ExtensionType::TransferFeeConfig,
    // ExtensionType::TransferHook,
    ExtensionType::InterestBearingConfig,
    // ExtensionType::PermanentDelegate,
    // ExtensionType::ConfidentialTransferMint,
    // ExtensionType::ConfidentialTransferFeeConfig,
    // ExtensionType::MintCloseAuthority,
    // ExtensionType::NonTransferable,
];
```

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:C/Y:N (1.3)

Recommendation

It is recommended to keep this relaxation explicitly documented as a governance-level assumption, and require collateral onboarding review to evaluate Token-2022 extension behavior before approval.

Remediation Comment

ACKNOWLEDGED: The **Solstice Labs team** has acknowledged the issue and decided not to make any changes in their business logic based on the statement:

> Solstice wants to work on a case by case basis to accept new collaterals. Therefore authorised extensions may vary from token to another, hence governance based assessment.

8.2 FREEZE_TOKEN RELIES ON RAW SPL TOKEN FAILURES FOR REPEATED FREEZE AND THAW ATTEMPTS

// INFORMATIONAL

Description

In the `programs/usx/src/instructions/freeze_token.rs` unlike `freeze_program`, which validates the requested state transition and returns a protocol-defined `InvalidFreezeState` error, `freeze_token` does not check whether the target account is already frozen or already thawed before issuing the CPI. Repeated freeze or thaw attempts therefore fail with the downstream SPL Token program's generic "Invalid account state for operation" error. This does not endanger funds, but it creates inconsistent behavior across the protocol's emergency controls and makes operational tooling, monitoring, and incident handling more brittle because callers cannot rely on a stable protocol-level error surface.

Code Location

The `freeze_program` explicitly rejects a no-op state transition first:

Copy Code

```
impl FreezeProgram<'_> {
    pub fn validate(&self, is_frozen: bool) -> Result<()> {
        if is_frozen as u8 == self.controller.load()?.is_frozen {
            return Err(UsxError::InvalidFreezeState.into());
        }
```

The `freeze_token` directly issues the CPI without checking whether the token account is already in the requested state:

Copy Code

```
pub(crate) fn handler(ctx: Context<FreezeToken>, is_frozen: bool) -> Result<()> {
    let controller = ctx.accounts.controller.load()?;
    let version = controller.version;
    let controller_signer_seeds: &[&[&[u8]]] = &[&[CONTROLLER_NAMESPACE, &[controller.bump]]];
    drop(controller);

    if is_frozen {
        let cpi_accounts = token::FreezeAccount {
            mint: ctx.accounts.redeemable_mint.to_account_info(),
            account: ctx.accounts.token_account.to_account_info(),
            authority: ctx.accounts.controller.to_account_info(),
        };
        token::freeze_account(CpiContext::new_with_signer(
            cpi_program,
            cpi_accounts,
            controller_signer_seeds,
        ))?;
    } else {
        let cpi_accounts = token::ThawAccount {
            mint: ctx.accounts.redeemable_mint.to_account_info(),
            account: ctx.accounts.token_account.to_account_info(),
            authority: ctx.accounts.controller.to_account_info(),
        };
        token::thaw_account(CpiContext::new_with_signer(
            cpi_program,
            cpi_accounts,
            controller_signer_seeds,
        ))?;
    }
```

BVSS

AO:A/AC:L/AX:M/R:F/S:U/C:N/A:N/I:L/D:N/Y:N (0.4)

Recommendation

It is recommended to read the target token account state before performing the CPI and reject no-op transitions with a protocol-defined error, mirroring the behavior of `freeze_program`.

Remediation Comment

ACKNOWLEDGED: The **Solstice Labs team** has acknowledged the issue and decided not to make any changes in their business logic based on the statement:

> This poses no risk the protocol, error is handled by the token program itself.

// Security Assessment

12.01.2025 - 12.03.2025

USX Staking

Solstice Labs

HALBORN

USX Staking - Solstice Labs

Prepared by: HALBORN

Last Updated 03/31/2026

Date of Engagement: December 1st, 2025 - December 3rd, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	0	0	0	2

TABLE OF CONTENTS

1. Introduction

`Solstice Labs` engaged Halborn to conduct a security assessment on their USX YieldVault Solana programs beginning on December 1st, 2025, and ending on December 3rd, 2025. The security assessment was scoped to the Solana Programs provided in usx-staking GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The `USX Yield Vault` Program enables users to lock their USX stablecoin in exchange for eUSX tokens, which represent their share in the vault. As the vault earns rewards in USX over time, the value of eUSX increases. Users can redeem their asset at any point by burning eUSX to receive their original USX deposit along with a proportional share of the accumulated rewards. This mechanism provides a simple and secure way to earn passive yield on USX holdings.

2. Assessment Summary

`Halborn` was provided 3 days for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Solana Programs.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were addressed by the `Solstice Labs` team. The main ones were the following:

- `Reject zero amount yield transfers to prevent unintended vesting resets.`
- `Require the proposed new authority to sign the confirmation step.`

3. Verified Program Build Hashes

The following program artifacts for the commit (`d9d2e7`) were independently reproduced and verified using a verifiable build process (`anchor build --verifiable`). Both programs correspond to the same underlying program, with the relevant difference between them being the `program ID`. For each program, the SHA-256 digest of the compiled `.so` artifact and the executable hash were confirmed.

- `programID`: **strcY2mYnQPyKLJvKyqRyaTkon4Lab4bNjUh5DZ6N7u**
- `sha`: **d1ec255db6bcd9a089415cb886bdf4f19079ec7eb03c80dae3f7bdf1c50e5b00**
- `checksum`: **5f7d18be4e4f92307ac0f63b14b3f3537c2b207022651da7335a26c533a326f3**

- `programID`: **eUSXyKoZ6aGejYVbnp3wtWQ1E8zuokLAJPecPxxtgG3**
- `sha`: **77466e8b0805ae78a520de1bc4e0c614d97294cd1c2fc9a247edc965888fc19e**
- `checksum`: **667c53b3dfd0bd8c8df00995a4c4ce88cb5094a2181db224cc4a4d8f41bd32cc**

4. Test Approach And Methodology

`Halborn` performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities `cargo audit`).
- Local runtime testing `anchor test`)

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

REPOSITORY

(a) Repository: usx-staking

(b) Assessed Commit ID: 13d7fa2

(c) Items in scope:

- programs/yield-vault/Cargo.toml
- programs/yield-vault/Xargo.toml
- programs/yield-vault/src/instructions/edit_program_freeze_status.rs
- programs/yield-vault/src/instructions/initialize_authority_transfer.rs
- programs/yield-vault/src/instructions/initialize_vesting_schedule.rs
- programs/yield-vault/src/instructions/transfer_in_yield.rs
- programs/yield-vault/src/instructions/initialize_permissioned_account.rs
- programs/yield-vault/src/instructions/confirm_authority_transfer.rs
- programs/yield-vault/src/instructions/edit_permissioned_account.rs
- programs/yield-vault/src/instructions/withdraw.rs
- programs/yield-vault/src/instructions/edit_controller.rs
- programs/yield-vault/src/instructions/initialize_metadata.rs
- programs/yield-vault/src/instructions/unlock.rs
- programs/yield-vault/src/instructions/edit_metadata.rs
- programs/yield-vault/src/instructions/mod.rs
- programs/yield-vault/src/instructions/initialize_yield_pool.rs
- programs/yield-vault/src/instructions/lock.rs
- programs/yield-vault/src/instructions/initialize_controller.rs
- programs/yield-vault/src/events.rs
- programs/yield-vault/src/constants.rs
- programs/yield-vault/src/error.rs
- programs/yield-vault/src/lib.rs
- programs/yield-vault/src/utils/validate_permission.rs
- programs/yield-vault/src/utils/validate_controller.rs
- programs/yield-vault/src/utils/validate_cooldown.rs
- programs/yield-vault/src/utils/mod.rs
- programs/yield-vault/src/utils/validate_yield.rs
- programs/yield-vault/src/utils/maths.rs
- programs/yield-vault/src/utils/validate_amount.rs
- programs/yield-vault/src/utils/helper.rs
- programs/yield-vault/src/state/yield_pool.rs
- programs/yield-vault/src/state/vesting_schedule.rs
- programs/yield-vault/src/state/controller.rs
- programs/yield-vault/src/state/mod.rs
- programs/yield-vault/src/state/permissioned_account.rs
- programs/yield-vault/src/state/cooldown_escrow.rs

Out-of-Scope: External dependencies and economic attack vectors.

REMEDIATION COMMIT ID:

- 0576de2

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
ZERO ASSET TRANSFERS CAUSE FULL VESTING DELAY	INFORMATIONAL	SOLVED - 01/28/2026
TWO-STEP AUTHORITY TRANSFER MECHANISM CAN BE IMPROVED	INFORMATIONAL	SOLVED - 01/28/2026

8. FINDINGS & TECH DETAILS

8.1 ZERO ASSET TRANSFERS CAUSE FULL VESTING DELAY

// INFORMATIONAL

Description

The `transfer_in_yield` instruction deposits yield from a permissioned account into the protocol's asset vault, updates the yield pool's total assets, and starts a new vesting schedule based on the configured `vesting_period_in_seconds`.

However, if it is called with a zero `asset_amount`, the program still creates a new vesting period with zero rewards, unnecessarily blocking all subsequent yield deposits for the entire vesting duration and could delay the expected yield emissions.

programs/yield-vault/src/instructions/transfer_in_yield.rs

Copy Code

```
79  pub(crate) fn handler(ctx: Context<TransferInYield>, asset_amount: u64) -> Result<()> {
80      // - 1 [TRANSFER HARVESTER'S ASSETS TO ASSET VAULT] -----------------
81      {
82          let cpi_program = ctx.accounts.token_program.to_account_info();
83          let cpi_accounts = token::TransferChecked {
84              from: ctx.accounts.harvester_assets.to_account_info(),
85              to: ctx.accounts.asset_vault.to_account_info(),
86              authority: ctx.accounts.harvester.to_account_info(),
87              mint: ctx.accounts.asset_mint.to_account_info(),
88          };
89          token::transfer_checked(
90              CpiContext::new(cpi_program, cpi_accounts),
91              asset_amount,
92              ctx.accounts.asset_mint.decimals,
93          )?;
94      }
95
96      // - 2 [UPDATE ACCOUNTING FOR YIELD POOL] -----------------
97      let yield_pool = &mut ctx.accounts.yield_pool.load_mut()?;
98      let total_assets_before = yield_pool.total_assets;
99
100     yield_pool.total_assets = total_assets_before
101         .checked_add(asset_amount as u128)
102         .ok_or_else(|| error!(YieldVaultError::MathError))?;
103
104     let total_assets_after = yield_pool.total_assets;
105
106     // - 3 [UPDATE ACCOUNTING FOR VESTING SCHEDULE] --------------------------------
107
108     let vesting_schedule: &mut std::cell::RefMut<'_, VestingSchedule> = &mut ctx.accounts.vest
109     let vesting_period_in_seconds = ctx.accounts.controller.load()?.vesting_period_in_seconds;
110     vesting_schedule.vesting_amount = asset_amount;
111     vesting_schedule.start_time = u64::try_from(Clock::get()?.unix_timestamp)
112         .map_err(|_| error!(YieldVaultError::InvalidCurrentBlockTime))?;
113     vesting_schedule.end_time = vesting_schedule
114         .start_time
115         .checked_add(vesting_period_in_seconds)
116         .ok_or_else(|| error!(YieldVaultError::MathError))?;
117
118     emit!(TransferInYieldEvent {
119         controller: ctx.accounts.controller.key(),
120         yield_pool: ctx.accounts.yield_pool.key(),
121         harvester: ctx.accounts.harvester.key(),
122         asset_amount,
123         total_assets_before,
124
```

```
            total_assets_after,
            vesting_start_time: vesting_schedule.start_time,
            vesting_end_time: vesting_schedule.end_time,
        });

        Ok(())
    }

    impl TransferInYield<'_> {
        pub(crate) fn validate(&self) -> Result<()> {
            validate_program_is_not_frozen(self.controller.load()?)?;
            validate_permission(self.permissioned_account.load()?.can_send_yield)?;
            validate_is_last_vesting_ended(self.vesting_schedule.load()?.end_time)?;

            Ok(())
        }
    }
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.0)

Recommendation

To address this finding, it is recommended to reject zero amount yield transfers to prevent unintended vesting resets.

Remediation Comment

SOLVED: The **Solstice Labs team** solved the finding by adding a validation to prevent zero amount yield transfers.

Remediation Hash

0576de2415927b455df8037863a22b4522e6a8f8

8.2 TWO-STEP AUTHORITY TRANSFER MECHANISM CAN BE IMPROVED

// INFORMATIONAL

Description

The current `initialize_authority_transfer` and `confirm_authority_transfer` flow allows the current authority to set a new authority address and finalize the transfer without confirmation from the proposed party.

While this is functionally correct and poses no direct security risk (since only the current authority can initiate the transfer), it relies on the assumption that the provided new authority address is correct and controlled.

A secure way would be to require the proposed new authority to explicitly confirm the transfer by signing the `confirm_authority_transfer` instruction. This ensures that:

1. The new authority address is valid and correct (i.e., the private key is known and usable).
2. Accidental misconfigurations do not result in a loss of control.
3. The authority transfer becomes an intentional requiring action from both parties.

programs/yield-vault/src/instructions/confirm_authority_transfer.rs

Copy Code

```
#[derive(Accounts)]
pub struct ConfirmAuthorityTransfer<'info> {
    /// #1 Authored call accessible only to the signer matching Controller.authority
    pub authority: Signer<'info>,

    /// #2 Payer of the transaction
    #[account(mut)]
    pub payer: Signer<'info>,

    /// #3 Top level Controller account managing the program
    #[account(
        mut,
        seeds = [CONTROLLER_NAMESPACE],
        bump = controller.load()?.bump,
        has_one = authority @YieldVaultError::InvalidAuthority,
        has_one = proposed_new_authority @YieldVaultError::InvalidAuthorityTransfer,
    )]
    pub controller: AccountLoader<'info, Controller>,

    /// #4 Address of the proposed new authority to be confirmed
    /// CHECK: checked in the controller account
    pub proposed_new_authority: UncheckedAccount<'info>,
}

pub(crate) fn handler(ctx: Context<ConfirmAuthorityTransfer>) -> Result<()> {
    let controller = &mut ctx.accounts.controller.load_mut()?;
    controller.authority = controller.proposed_new_authority;
    controller.has_proposed_authority = 0;

    emit!(ConfirmAuthorityTransferEvent {
        controller: ctx.accounts.controller.key(),
        new_authority: ctx.accounts.proposed_new_authority.key(),
    });
    Ok(())
}

impl ConfirmAuthorityTransfer<'_> {
```

```
    pub(crate) fn validate(&self) -> Result<()> {
        validate_program_is_not_frozen(self.controller.load()?)?;
        require!(
            self.controller.load()?.has_proposed_authority == 1,
            YieldVaultError::InvalidAuthorityTransfer
        );
        Ok(())
    }
}
```

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To address this finding, it is recommended to require the proposed new authority to sign the confirmation step.

Remediation Comment

SOLVED: The **Solstice Labs team** solved the finding by requiring the signature of proposed_new_authority during the `initialize_authority_transfer` instruction.

Remediation Hash

0576de2415927b455df8037863a22b4522e6a8f8

9. AUTOMATED TESTING

Static Analysis Report

Description

`Halborn` used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was `cargo audit`, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Cargo Audit Results

ID	CRATE	DESCRIPTION
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
RUSTSEC-2025-0009	ring	Some AES functions may panic when overflow checking is enabled.
RUSTSEC-2025-0024	crossbeam-channel	crossbeam-channel: double free on Drop

---

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.

// Security Assessment

12.08.2025 - 12.15.2025

USX Rewarder

Solstice Labs

HALBORN

USX Rewarder - Solstice Labs

Prepared by: HALBORN

Last Updated 01/12/2026

Date of Engagement: December 8th, 2025 - December 15th, 2025

Summary

NO REPORTED FINDINGS TO ADDRESS

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	0	0	0

TABLE OF CONTENTS

1. Introduction

`Solstice Labs` engaged Halborn to conduct a security assessment on their USX Rewarder Solana programs beginning on December 8th, 2025, and ending on December 15th, 2025. The security assessment was scoped to the Solana Programs provided in usx-rewarder GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

The `Rewarder` program acts as a centralized operational gateway for injecting and distributing USX yield into the Yield Vault ecosystem. It allows authorized admin operators to deposit USX rewards into an isolated asset vault and subsequently trigger the distribution of those funds to the main Yield Vault protocol. By decoupling yield injection from user deposits, it provides a secure, permissioned buffer for managing reward flows while maintaining strict access controls via a dedicated Admin PDA system.

2. Assessment Summary

`Halborn` was provided 6 days for the engagement and assigned one full-time security engineer to review the security of the Solana Programs in scope. The engineer is a blockchain and smart contract security expert with advanced smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the Solana Programs.
- Ensure that smart contract functionality operates as intended.

In summary, `Halborn` did not identify any security or operational risks during the review. The assessed components were found to be well designed and implemented, with no issues requiring remediation at the time of the assessment.

3. Test Approach And Methodology

`Halborn` performed a combination of a manual review of the source code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the program assessment. While manual testing is recommended to uncover flaws in business logic, processes, and implementation; automated testing techniques help enhance coverage of programs and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Manual program source code review to identify business logic issues.
- Mapping out possible attack vectors
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning dependencies for known vulnerabilities (`cargo audit`).
- Local runtime testing (`anchor test`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILITY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

REPOSITORY

(a) Repository: usx-rewarder

(b) Assessed Commit ID: bdbcf2d

(c) Items in scope:

- programs/harvester/src/constants.rs
- programs/harvester/src/error.rs
- programs/harvester/src/events.rs
- programs/harvester/src/instructions/add_yield.rs
- programs/harvester/src/instructions/distribute_yield.rs
- programs/harvester/src/instructions/edit_admin.rs
- programs/harvester/src/instructions/edit_program_freeze_status.rs
- programs/harvester/src/instructions/initialize_admin.rs
- programs/harvester/src/instructions/initialize_controller.rs
- programs/harvester/src/instructions/mod.rs
- programs/harvester/src/lib.rs
- programs/harvester/src/state/admin.rs
- programs/harvester/src/state/controller.rs
- programs/harvester/src/state/mod.rs
- programs/harvester/src/utils.rs

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	0	0

INFORMATIONAL
0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE

7. FINDINGS & TECH DETAILS

8. AUTOMATED TESTING

Static Analysis Report

Description

`Halborn` used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was `cargo audit`, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in https://crates.io are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. All vulnerabilities shown here were already disclosed in the above report. However, to better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

Cargo Audit Results

ID	Crate	Description
RUSTSEC-2025-0024	crossbeam-channel	The internal `Channel` type's `Drop` method has a race which could, in some circumstances, lead to a double-free.
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on `ed25519-dalek`
RUSTSEC-2025-0009	ring	Some AES functions may panic when overflow checking is enabled.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.