// Security Assessment

02.05.2026 - 02.09.2026

Xyber Sale Solana Program

Xyber

HALBORN

Xyber Sale Solana Program - Xyber

Prepared by: HALBORN

Last Updated 02/17/2026

Date of Engagement: February 5th, 2026 - February 9th, 2026

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW
9	0	0	1	1

INFORMATIONAL
7

TABLE OF CONTENTS

1. Introduction

`Xyber` engaged Halborn to conduct a security assessment on their `xyber-sale` program beginning on February 5th, 2026 and ending on February 9th, 2026. The security assessment was scoped to the smart contracts provided in the GitHub repository xyber-ico-solana, commit hashes, and further details can be found in the Scope section of this report.

The `Xyber team` is releasing a Solana program for conducting an ICO (Initial Coin Offering) of the XYBER token. The program manages the full lifecycle of the token sale, including initialization, round configuration, bucket-based token allocation, SOL and quote asset deposits, deterministic and priceless vesting schedules, token claiming with burn mechanics, and fund withdrawals by the admin.

2. Assessment Summary

Halborn was provided 3 business 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 Program.
- 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 `Xyber team`. The main ones were the following:

- `Validate bucket vesting type before accepting deposits or creating deterministic vesting configs to prevent funds lock`
- `Replace set_inner with individual field assignments in setup_bucket to preserve accounting state on re-initialization`
- `Validate base_period_index bounds in vesting plan setup to prevent panics or silent zero allocations`
- `Implement two-step ownership transfer for the multisig role to prevent irreversible loss of program control`
- `Validate time parameters in round setup to prevent misconfigured rounds`
- `Handle equal claim and burn ratios explicitly in remainder distribution to prevent silent burn bias`
- `Validate base_mint against a hardcoded address to prevent configuration of an incorrect ICO token`
- `Validate price and exponent parameters in quote mint configuration to prevent temporal denial of service`
- `Centralize all PDA seed strings in the constants module to prevent typos and ensure consistency`

3. Test Approach And Methodology

Halborn performed a combination of manual review and security testing based on scripts to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Differences analysis using GitLens to have a proper view of the differences between the mentioned commits
- Graphing out functionality and programs logic/connectivity/functions along with state changes

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
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
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: xyber-ico-solana

(b) Assessed Commit ID: e69420c

(c) Items in scope:

- programs/xyber-sale/src/lib.rs
- programs/xyber-sale/src/constants.rs
- programs/xyber-sale/src/data.rs
- programs/xyber-sale/src/errors.rs
- programs/xyber-sale/src/vesting_calculator.rs
- programs/xyber-sale/src/instructions/initialize.rs
- programs/xyber-sale/src/instructions/set_quote_mint.rs
- programs/xyber-sale/src/instructions/setup_round.rs
- programs/xyber-sale/src/instructions/setup_bucket.rs
- programs/xyber-sale/src/instructions/deposit_sol.rs
- programs/xyber-sale/src/instructions/deposit_asset.rs
- programs/xyber-sale/src/instructions/setup_vesting_plan.rs
- programs/xyber-sale/src/instructions/setup_deterministic_vesting.rs
- programs/xyber-sale/src/instructions/claim.rs
- programs/xyber-sale/src/instructions/withdraw.rs

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

REMEDIATION COMMIT ID:

- 6545c33
- 5f2a952
- aa5f987
- 1e7f189
- 97309ac
- dbb2f89
- f58eb69
- ba699b4
- b9bb3dd

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	0	1	1

INFORMATIONAL
7

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
MISSING BUCKET VESTING TYPE VALIDATION ON DEPOSIT AND DETERMINISTIC SETUP CAUSES FUNDS PERMANENTLY LOCKED	MEDIUM	SOLVED - 02/09/2026
UNVALIDATED BUCKET DATA OVERWRITES ACCOUNTING STATE ON	LOW	SOLVED - 02/09/2026

RE-INITIALIZATION		
UNVALIDATED BASE PERIOD INDEX IN VESTING PLAN CAUSES PANIC OR SILENT ZERO ALLOCATION	INFORMATIONAL	SOLVED - 02/11/2026
MISSING TWO-STEP OWNERSHIP TRANSFER FOR MULTISIG ROLE ALLOWS IRREVERSIBLE LOSS OF PROGRAM CONTROL	INFORMATIONAL	SOLVED - 02/11/2026
MISSING TIME PARAMETER VALIDATION IN ROUND SETUP ALLOWS MISCONFIGURED ROUNDS	INFORMATIONAL	SOLVED - 02/11/2026
REMAINDER TOKEN DISTRIBUTION SILENTLY FAVORS BURN WHEN RATIOS ARE EQUAL	INFORMATIONAL	SOLVED - 02/11/2026
MISSING BASE MINT ADDRESS VALIDATION ALLOWS CONFIGURATION OF INCORRECT ICO TOKEN	INFORMATIONAL	SOLVED - 02/11/2026
INCORRECT PARAMETERS IN QUOTE MINT CONFIGURATION MIGHT LEAD TO TEMPORAL DOS	INFORMATIONAL	SOLVED - 02/11/2026
INCONSISTENT USE OF INLINE PDA SEED STRINGS INSTEAD OF CENTRALIZED CONSTANTS	INFORMATIONAL	SOLVED - 02/11/2026

7. FINDINGS & TECH DETAILS

7.1 MISSING BUCKET VESTING TYPE VALIDATION ON DEPOSIT AND DETERMINISTIC SETUP CAUSES FUNDS PERMANENTLY LOCKED

// MEDIUM

Description

The `deposit_sol`, `deposit_asset`, and `setup_deterministic_vesting` instructions do not validate that the bucket's `vesting_type` matches the type of operation being performed.

The `deposit_sol` and `deposit_asset` instructions call `update_allocation`, which always creates a `Priceless` vesting config, but neither instruction checks that `bucket_data.vesting_type` is `Some(BucketVestingType::Priceless)` before accepting the deposit, as shown in the code snippet below.

Similarly, `setup_deterministic_vesting` creates a `Deterministic` vesting config without checking that `bucket_data.vesting_type` is `Some(BucketVestingType::Deterministic)`.

The type mismatch is only detected at claim time in `claim.rs`, where a `match` on `(bucket_data.vesting_type, vesting_config.vesting_type)` panics if the types do not match.

programs/xyber-sale/src/instructions/claim.rs

Copy Code

```
86          (Some(BucketVestingType::Deterministic), Some(VestingType::Determ
87          (Some(BucketVestingType::Priceless), Some(VestingType::Priceless
89      }
```

A user who deposits SOL or quote assets into a `Deterministic` bucket has their funds transferred to the pool but receives a `Priceless` vesting config that will

never pass the claim-time type check.

The deposited funds are permanently locked since there is no refund mechanism, and the user can never claim tokens from this vesting config.

Similarly, an admin who creates a `Deterministic` vesting config in a `Priceless` bucket increases `registered_supply` for tokens that can never be claimed, wasting bucket supply.

Proof of Concept

PoC Code

Copy Code

```
/// TS23 POC: Depositing SOL into a Deterministic bucket creates a Priceless
/// vesting config. At claim time, the vesting type mismatch (bucket=Determinist
/// user=Priceless) causes a panic, permanently locking the user's deposited SOL
#[test]
fn ts23_deposit_sol_into_deterministic_bucket_locks_funds() {
    // ============================================================
    // SETUP
    // ============================================================
    let mut test = XyberSaleTest::new();
    test.initialize();

    let now = test.current_time();
    test.setup_round(now - 60, now + 3600);

    // Setup a DETERMINISTIC bucket (intended for team allocations)
    let bucket_name = "PUBLIC";
    test.setup_bucket(
        bucket_name,
        BucketData {
            vesting_type: Some(BucketVestingType::Deterministic),
            total_deposit: 0,
            bucket_supply: 500_000_000_000,
            registered_supply: 0,
            claimed_supply: 0,
            burnt_supply: 0,
            vesting_plan: vec!["PUBLIC".to_string()],
        },
    );

    // Setup vesting plan (100% unlock at TGE)
    test.setup_vesting_plan(
        "PUBLIC",
        VestingPlan {
            periods: vec![VestingPeriod {
                start_timestamp: now as u64,
                claim_ratio: 1.0,
                burn_ratio: 0.0,
```

```
            base_period_index: None,
        }],
    },
);

// Mint tokens to bucket so claim would theoretically work
test.mint_tokens_to_bucket(bucket_name, 500_000_000_000);

let buyer_pubkey = test.get_user("buyer").pubkey();
let deposit_amount: u64 = 5_000_000_000; // 5 SOL

// ============================================================
// PRE-CALL ASSERTIONS
// ============================================================
let buyer_balance_before = test.get_sol_balance(&buyer_pubkey);
let bucket_pool_balance_before = test.get_sol_balance(&test.get_bucket_pool_

// ============================================================
// EXECUTE: Deposit SOL into a Deterministic bucket
// This SHOULD fail but doesn't — no validation on bucket vesting type
// ============================================================
let deposit_result = test.deposit_sol_tx("buyer", bucket_name, deposit_amoun
assert!(
    deposit_result.is_ok(),
    "Deposit into Deterministic bucket should NOT fail (this is the bug)"
);

// ============================================================
// POST-DEPOSIT ASSERTIONS: SOL was transferred, Priceless vesting created
// ============================================================
let buyer_balance_after = test.get_sol_balance(&buyer_pubkey);
let bucket_pool_balance_after = test.get_sol_balance(&test.get_bucket_pool_p

// Buyer lost SOL
assert!(
    buyer_balance_before - buyer_balance_after >= deposit_amount,
    "Buyer should have lost at least {} lamports",
    deposit_amount
);

// Bucket pool received SOL
assert!(
    bucket_pool_balance_after > bucket_pool_balance_before,
    "Bucket pool should have received SOL"
);

// ============================================================
// EXECUTE: Try to claim — panics due to vesting type mismatch
// Bucket is Deterministic but user's vesting is Priceless
// ============================================================
test.svm.expire_blockhash();

let claim_result = test.claim_tx("buyer", bucket_name, "PUBLIC");
assert!(
    claim_result.is_err(),
    "Claim should fail due to vesting type mismatch (Deterministic vs Pricel
);

// ============================================================
// FINAL ASSERTIONS: User's SOL is permanently locked
// ============================================================
let buyer_final_balance = test.get_sol_balance(&buyer_pubkey);
```

```
    // The buyer's SOL is still in the bucket pool, unrecoverable
    // There is no refund mechanism in the program
    assert!(
        buyer_balance_before - buyer_final_balance >= deposit_amount,
        "Buyer's SOL is permanently locked — deposited {} lamports with no way t
        deposit_amount
    );

    println!("=== TS23 POC: Deposit SOL into Deterministic Bucket ===");
    println!(
        "Buyer deposited {} lamports into a Deterministic bucket",
        deposit_amount
    );
    println!("Deposit succeeded (no bucket vesting type validation)");
    println!("Claim failed (vesting type mismatch: bucket=Deterministic, user=Pr
    println!(
        "Buyer lost {} lamports with no refund mechanism",
        buyer_balance_before - buyer_final_balance
    );
    println!("=== END TS23 POC ===");
}
```

Evidence

```
* Executing task: cargo test --package xyber-sale --test mod -- instructions::deposit_sol::ts23_deposit_sol_into_d
eterministic_bucket_locks_funds --exact --nocapture

    Finished `test` profile [unoptimized + debuginfo] target(s) in 0.31s
     Running tests/mod.rs (target/debug/deps/mod-e2e1329a07199d95)

running 1 test
=== TS23 POC: Deposit SOL into Deterministic Bucket ===
Buyer deposited 5000000000 lamports into a Deterministic bucket
Deposit succeeded (no bucket vesting type validation)
Claim failed (vesting type mismatch: bucket=Deterministic, user=Priceless)
Buyer lost 5001450720 lamports with no refund mechanism
=== END TS23 POC ===
test instructions::deposit_sol::ts23_deposit_sol_into_deterministic_bucket_locks_funds ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 15 filtered out; finished in 0.11s

* Terminal will be reused by tasks, press any key to close it.
```

BVSS

AO:A/AC:L/AX:M/R:P/S:U/C:N/A:H/I:C/D:M/Y:C (5.2)

Recommendation

It is recommended to validate `bucket_data.vesting_type` at the beginning of each instruction before any state changes or transfers occur.

For `deposit_sol` and `deposit_asset`, add a check that the bucket is `Priceless`:

```
require!(
    bucket_data.vesting_type == Some(BucketVestingType::Priceless),
    CustomError::InvalidBucketVestingType
);
```

For `setup_deterministic_vesting`, add a check that the bucket is `Deterministic`:

```
require!(
    bucket_data.vesting_type == Some(BucketVestingType::Deterministic),
    CustomError::InvalidBucketVestingType
);
```

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/6545c33a4e5741fada482556ae74dd373319f87c

7.2 UNVALIDATED BUCKET DATA OVERWRITES ACCOUNTING STATE ON RE-INITIALIZATION

// LOW

Description

In the `setup_bucket` instruction, the `bucket_data` argument is stored directly

via `set_inner` without any validation, as shown in the code snippet below.

The bucket account uses `init_if_needed`, which allows the admin to call the instruction on an already-initialized bucket.

When called on an existing bucket that has received deposits or claims, `set_inner` overwrites all `BucketData` fields, including the accounting fields `total_deposit`, `claimed_supply`, `burnt_supply`, and `registered_supply`.

These fields are modified by `deposit_sol`, `deposit_asset`, `setup_deterministic_vesting`, and `claim` during the bucket's lifecycle, and their values represent the current state of user deposits and token distributions.

Additionally, the `bucket_name` argument is not validated for length, which could cause issues with PDA derivation or increased transaction costs if an excessively long string is provided.

programs/xyber-sale/src/instructions/setup_bucket.rs

Copy Code

```
    #[account(
        payer = admin,
        space = 8 + BucketData::INIT_SPACE,
        seeds = [SEED_ROOT, b"BUCKET", bucket_name.as_bytes()],
        bump
    )]
    pub bucket: Box<Account<'info, BucketData>>,
    ...

pub fn setup_bucket(
    ctx: Context<SetupBucket>,
    _bucket_name: String,
    bucket_data: BucketData,
) -> Result<()> {
    Ok(())
}
```

If the admin calls `setup_bucket` on a bucket that already has deposits, the `total_deposit` field is reset, which directly affects the `get_participant_allocation` formula `deposit * bucket_supply / total_deposit`).

A reset `total_deposit` of zero causes a division-by-zero panic in `get_participant_allocation`, permanently blocking all claims for that bucket.

A reset to a smaller value inflates all participant allocations, potentially allowing claims that exceed the actual token balance in the pool.

The `claimed_supply`, `burnt_supply`, and `registered_supply` fields are also reset, breaking the bucket-level accounting.

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:H/I:C/D:H/Y:H (3.1)

Recommendation

It is recommended to replace `set_inner` with individual field assignments that only update the configurable fields `vesting_type`, `bucket_supply`, `vesting_plan`) while preserving the accounting fields.

programs/xyber-sale/src/instructions/setup_bucket.rs

Copy Code

```
48 pub fn setup_bucket(
49     ctx: Context<SetupBucket>,
50     _bucket_name: String,
51     bucket_data: BucketData,
52 ) -> Result<()> {
58 }
```

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes. A compile-time feature flag `reset-allowed` was introduced to guard the `set_inner` call. When disabled (production), only configurable fields `vesting_type`, `bucket_supply`, `vesting_plan`) are writable, preserving

accounting state.

Remediation Hash

https://github.com/Xyber-Labs/xyber-genesis-sale/commit/5f2a952fe837262b59b97cf45c8c7bcdd17d7dea

7.3 UNVALIDATED BASE PERIOD INDEX IN VESTING PLAN CAUSES PANIC OR SILENT ZERO ALLOCATION

// INFORMATIONAL

Description

In the `recalculate_plan` function of `VestingCalculator`, the `base_period_index` field from each `VestingPeriod` is used to index into the `calc_base` vector without any bounds or ordering validation, as shown in the code snippet below.

The `base_period_index` value is set by the admin via `setup_vesting_plan`, which stores the `VestingPlan` argument directly through `set_inner` without validating the individual period fields.
When `base_period_index` is `Some(i)`, the value `calc_base[i as usize]` is used as the base for the ratio calculation instead of `total_allocation`.

- If `i >= len` (number of periods), the access panics with an index-out-of-bounds error.
- If `i >= current_iteration_index`, `calc_base[i]` has not been calculated yet and is still `0` from initialization, silently producing a base of `0` for the ratio calculation.

programs/xyber-sale/src/vesting_calculator.rs

Copy Code

```
71      pub fn recalculate_plan(&mut self, vesting_plan: Vec<VestingPeriod>)
72          let len = vesting_plan.len();
```

```
73          let mut calc_base = vec![0; len];
74          ...
75          for (i, period) in vesting_plan.iter().enumerate() {
76              let b =
77                  period.base_period_index.map_or(self.total_allocation, |i
78
79              to_allocate_sum += Self::mul_corrected(b, period.claim_ratio)
81              ...
82          }
83      }
```

If an admin configures a vesting plan with a `base_period_index` that equals or exceeds the number of periods, the `claim` instruction panics for every participant in that bucket, permanently blocking all token claims until the admin reconfigures the plan.
If the index points to a period that has not been calculated yet (forward reference), the base becomes `0`, causing `mul_corrected(0, ratio)` to return `0` for that period's allocation and burn amounts.

This silently reduces the participant's claimable tokens without any error, and the remainder is redistributed to burn or claim in the last period depending on the ratio comparison.

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:C/I:H/D:N/Y:C (1.4)

Recommendation

It is recommended to validate `base_period_index` in the `setup_vesting_plan` instruction before storing the plan.

Each period's `base_period_index` should be strictly less than its own position in the periods array, ensuring it only references previously calculated periods.

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes. A `VestingPlan::is_valid()` method now validates that `base_period_index` is strictly less than the current period index, preventing out-of-bounds panics and

silent forward references.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/aa5f9877ef96ee09e0dd76d3a932f0177edaebdd

7.4 MISSING TWO-STEP OWNERSHIP TRANSFER FOR MULTISIG ROLE ALLOWS IRREVERSIBLE LOSS OF PROGRAM CONTROL

// INFORMATIONAL

Description

In the `initialize` instruction, the `multisig` parameter is directly assigned to `config.multisig` in a single step, as shown in the code snippet below.
After the first initialization, the admin constraint requires that `config.multisig == admin.key()`, meaning only the current multisig signer can call `initialize` to update any configuration value.

If the multisig is set to an incorrect or inaccessible address, no entity can call `initialize` again to correct it, because the constraint will always reject any other signer.

Unlike the `admin` role, which can be recovered by the multisig calling `initialize` with a corrected address, the multisig role itself has no recovery mechanism.

programs/xyber-sale/src/instructions/initialize.rs

Copy Code

```
#[derive(Accounts)]
pub struct Initialize<'info> {

    pub admin: Signer<'info>,
    ...
}

pub fn initialize(ctx: Context<Initialize>, new_admin: Pubkey, multisig:
    ...
```

```
    ...
}
```

If the multisig is accidentally set to an incorrect address, all administrative operations that require multisig authorization become permanently inaccessible.

This includes the ability to update the admin, change the multisig itself, withdraw funds, and any other operation gated by the multisig constraint.

BVSS

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

Recommendation

It is recommended to implement a two-step ownership transfer pattern for the multisig role.
This ensures the new multisig address is accessible before the transfer is finalized.
1. In the first step, the current multisig proposes a new multisig address, which is stored in a `pending_multisig` field on the config account.
programs/xyber-sale/src/data.rs

Copy Code

```
pub struct SaleConfig {
    pub admin: Pubkey,
    pub base_mint: Pubkey,
    pub multisig: Pubkey,
}
```

2. In the second step, the proposed address must call a separate `accept_multisig` instruction to confirm the transfer, which then writes the new value to `config.multisig`.

Copy Code

```
#[derive(Accounts)]
pub struct ProposeMultisig<'info> {
    #[account(
        signer,
    )]

```

```
    #[account(
        mut,
        seeds = [SEED_ROOT, b"CONFIG"],
        bump
    )]
    pub config: Box<Account<'info, SaleConfig>>,
}

pub fn propose_multisig(ctx: Context<ProposeMultisig>, new_multisig: Pubk
    require!(new_multisig != Pubkey::default(), CustomError::BadParams);

    config.pending_multisig = new_multisig;
    Ok(())
}

#[derive(Accounts)]
pub struct AcceptMultisig<'info> {
    #[account(

        constraint = config.pending_multisig == new_multisig.key() @ Cust

    pub new_multisig: Signer<'info>,

    #[account(
        mut,
        seeds = [SEED_ROOT, b"CONFIG"],
        bump
    )]
    pub config: Box<Account<'info, SaleConfig>>,
}

pub fn accept_multisig(ctx: Context<AcceptMultisig>) -> Result<()> {


    config.pending_multisig = Pubkey::default();
    Ok(())
}
```

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/1e7f18993c9ac0063cb24a662b0e139eef8903e6

7.5 MISSING TIME PARAMETER VALIDATION IN ROUND SETUP ALLOWS MISCONFIGURED

ROUNDS

// INFORMATIONAL

Description

In the `setup_round` instruction, the `start_time` and `end_time` parameters are directly assigned to the `round_config` account without any validation, as shown in the code snippet below.
There is no check that `end_time > start_time`, that `start_time` is not in the past, or that either value is a reasonable timestamp.

programs/xyber-sale/src/instructions/setup_round.rs

Copy Code

```
30 pub fn setup_round(
31     ctx: Context<SetupRound>,
32     _round: Round,
33     start_time: i64,
34     end_time: i64,
35 ) -> Result<()> {
36     let round_config = &mut ctx.accounts.round_config;
39
40     Ok(())
41 }
```

The `round_config` account can store inconsistent temporal values, such as `start_time` greater than `end_time`, zero or negative timestamps, or dates in the past.

While the deposit instructions independently validate `now >= start_time` and `end_time >= now` before accepting deposits, the lack of validation at configuration time means the admin receives no immediate feedback about invalid parameters.

BVSS

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

Recommendation

It is recommended to validate that `end_time` is strictly greater than `start_time` before storing the values.

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/97309acceef95ecfabf0a48990ff04e12074312c

7.6 REMAINDER TOKEN DISTRIBUTION SILENTLY FAVORS BURN WHEN RATIOS ARE EQUAL

// INFORMATIONAL

Description

In the `recalculate_plan` function, the last period distributes any remainder tokens (caused by rounding in `mul_corrected`) to either the claim or burn sum based on a strict less-than comparison, as shown in the code snippet below.

- When `burn_ratio < claim_ratio`, the remainder goes to claim.
- When `burn_ratio >= claim_ratio` (including the equal case), the remainder goes entirely to burn.

This means that when both ratios are identical, all remainder tokens are silently assigned to burn instead of being split or assigned to claim.

programs/xyber-sale/src/vesting_calculator.rs

Copy Code

```
85              if i == len - 1 {
86                  if period.burn_ratio < period.claim_ratio {
88                          .total_allocation
89                          .saturating_sub(to_allocate_sum)
90                          .saturating_sub(to_burn_sum);
```

```
                    } else {
                        to_burn_sum += self
                            .total_allocation
                            .saturating_sub(to_allocate_sum)
                            .saturating_sub(to_burn_sum);
                    };
                    ...
                }
```

For example, with a single period where `claim_ratio = 0.1` and `burn_ratio = 0.1` on `total_allocation = 1000`:

- `to_allocate_sum = 100`
- `to_burn_sum = 100`
- `remainder = 800`
- Result: `900` tokens burned, only `100` claimable

Participants in vesting plans where the last period has equal claim and burn ratios receive fewer claimable tokens than expected.

The remainder tokens are permanently burned instead of being distributed for claiming, reducing the effective allocation without any explicit indication to the participant or the admin.

BVSS

AO:S/AC:L/AX:L/R:P/S:U/C:N/A:N/I:H/D:M/Y:H (1.1)

Recommendation

It is recommended to handle the equal-ratio case explicitly by assigning the remainder to claim instead of burn, since burning tokens is an irreversible operation.

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes. The condition was changed from `<` to `<=`, so the equal-ratio case now assigns the remainder to claim instead of burn.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/dbb2f8967c6fcbad0c387

6d233fc51367a075a41

7.7 MISSING BASE MINT ADDRESS VALIDATION ALLOWS CONFIGURATION OF INCORRECT ICO TOKEN

// INFORMATIONAL

Description

In the `initialize` instruction, the `base_mint` account is declared as `InterfaceAccount<'info, Mint>` without any address constraint, as shown in the code snippet below.

This means any valid SPL token mint can be passed as the `XYBER` base mint during initialization, even though the token address is known before deployment.

The program already hardcodes known addresses via the `DEPLOYER` constant in `constants.rs` using `pubkey!("...")`, but this same pattern is not applied to the `XYBER` token mint.

programs/xyber-sale/src/instructions/initialize.rs

Copy Code

```
10  #[derive(Accounts)]
11  pub struct Initialize<'info> {
12      #[account(signer, mut, constraint = config.multisig != Pubkey::defaul
13      pub admin: Signer<'info>,
14
15      #[account(
16          init_if_needed,
17          payer = admin,
18          space = 8 + SaleConfig::INIT_SPACE,
19          seeds = [SEED_ROOT, b"CONFIG"],
20          bump
21      )]
22      pub config: Box<Account<'info, SaleConfig>>,
23
24
25      ...
26  }
```

If an incorrect mint address is accidentally provided during initialization, all bucket ATAs, claim transfers, and burn operations would reference the wrong token instead of `XYBER`.

Since there is no on-chain validation, the misconfiguration would go undetected until participants attempt to claim and receive an unintended token in exchange for their SOL or stablecoin deposits.

BVSS

AO:S/AC:L/AX:H/R:P/S:U/C:N/A:C/I:C/D:M/Y:M (0.5)

Recommendation

It is recommended to add a hardcoded `BASE_MINT` constant in `constants.rs` following the same pattern used for `DEPLOYER`, and validate the `base_mint` account against it using an Anchor address constraint.

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes. A `XYBER_MINT` constant was added with a compile-time `check-mint` feature flag that enforces the base mint address constraint in production. The production build integrity is enforced via Solana Verified Build.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/f58eb691d25af4a9f2f002b02fe6616bfcf5e0cf

7.8 INCORRECT PARAMETERS IN QUOTE MINT CONFIGURATION MIGHT LEAD TO TEMPORAL DOS

// INFORMATIONAL

Description

In the `set_quote_mint` instruction, the `price` and `expo` parameters are directly assigned to `quote_config` without any validation, as shown in the code snippet below.

- The `price` parameter accepts any `i64` value, including zero and negative numbers.
- The `expo` parameter accepts any `i32` value, including extreme exponents that would cause overflow in downstream calculations.

These values are consumed by `convert_quote_to_sol` in `deposit_asset`, where `price` is used as a divisor and `expo` determines the magnitude of the numerator.

The `convert_quote_to_sol` function uses `price.unsigned_abs()` as the divisor and `10^expo.unsigned_abs()` as a multiplier in the numerator:

programs/xyber-sale/src/instructions/deposit_asset.rs

Copy Code

```
110 fn convert_quote_to_sol(quote_amount: u64, price: i64, expo: i32, quote_d
111     const SOL_DECIMALS: u32 = 9;
114
115     let numerator = (quote_amount as u128)
117         .and_then(|v| v.checked_mul(10u128.pow(SOL_DECIMALS)))
118         .expect("Overflow in numerator");
119
120     let denominator =
122
123     let sol_equivalent =
124         numerator.checked_div(denominator).expect("Error in convert_quote
125     ...
126 }
```

If `price` is set to zero, the denominator in `convert_quote_to_sol` becomes zero, causing a panic that blocks all `deposit_asset` operations for that quote mint. If `expo` is set to an extreme value, the numerator overflows and also causes a panic.
In both cases, the quote mint becomes non-functional until the multisig waits for

the 24-hour cooldown period to reconfigure it.

BVSS

AO:S/AC:L/AX:L/R:F/S:U/C:N/A:L/I:L/D:N/Y:N (0.2)

Recommendation

It is recommended to validate that `price` is strictly positive and that `expo` falls within a reasonable range before storing the values.

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes. Validation checks enforce `price > 0 && price <= MAX_PRICE` and `expo` within the range `[-12, 0]`, preventing division by zero and overflow in `convert_quote_to_sol`.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/ba699b4b44884af52937d908b3afeb5d73a0aa38

7.9 INCONSISTENT USE OF INLINE PDA SEED STRINGS INSTEAD OF CENTRALIZED CONSTANTS

// INFORMATIONAL

Description

The program defines several PDA seed constants in `constants.rs` (`SEED_ROOT`, `BUCKET_SEED`, `BUCKET_POOL_SEED`, `SALE_BUCKET_SEED`, `QUOTE_SEED`), but the majority of seed strings used across instructions are hardcoded as inline byte literals instead of referencing these constants.

The seeds `b"CONFIG"`, `b"ROUND"`, `b"VESTING_CONFIG"`, `b"VESTING_PLAN"`,

`b"BUCKET"`, and `b"BUCKET_POOL"` are used inline in account constraints across all 10 instruction files, as shown in the code snippets below.

In some cases, a constant already exists in `constants.rs` (e.g., `BUCKET_SEED` for `b"BUCKET"`, `BUCKET_POOL_SEED` for `b"BUCKET_POOL"`) but is not used in the instruction that declares the PDA.

For example, in `initialize.rs`, `b"BUCKET_POOL"` is used inline even though `BUCKET_POOL_SEED` is defined:

programs/xyber-sale/src/instructions/initialize.rs

Copy Code

```
27      #[account(
28          init_if_needed,
29          payer = admin,
30          space = 0,
31
32          ...
33      )]
34      pub bucket_pool: UncheckedAccount<'info>,
```

A typo in any inline seed literal (e.g., `b"CONFG"` instead of `b"CONFIG"`) would silently derive a different PDA, causing the instruction to fail or operate on an unintended account. Because the same string is duplicated across 10 files, the surface area for such errors is large and increases with future development.

The inconsistency between existing constants `BUCKET_SEED`, `BUCKET_POOL_SEED`) and their inline equivalents also creates confusion about which source of truth to use when adding or modifying instructions.

BVSS

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

Recommendation

Replace all inline usages with the centralized constants in the following files:

- `programs/xyber-sale/src/instructions/initialize.rs`
- `programs/xyber-sale/src/instructions/set_quote_mint.rs`

- programs/xyber-sale/src/instructions/setup_round.rs
- programs/xyber-sale/src/instructions/setup_bucket.rs
- programs/xyber-sale/src/instructions/setup_vesting_plan.rs
- programs/xyber-sale/src/instructions/setup_deterministic_vesting.rs
- programs/xyber-sale/src/instructions/deposit_sol.rs
- programs/xyber-sale/src/instructions/deposit_asset.rs
- programs/xyber-sale/src/instructions/claim.rs
- programs/xyber-sale/src/instructions/withdraw.rs

Remediation Comment

SOLVED: The **Xyber team** solved the issue by implementing the suggested changes.

Remediation Hash

https://github.com/Xyber-Labs/xyber-ico-solana/commit/b9bb3dd06d5a7b3b901a406a76b98fc1b52a6250

8. AUTOMATED TESTING

Description

Halborn used automated security scanners to assist with the 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 reviewers 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.

Results

ID	Package	Short Description
RUSTSEC-2024-0344	curve25519-dalek	Timing variability in curve25519-dalek's
RUSTSEC-2022-0093	ed25519-dalek	Double Public Key Signing Function Oracle Attack on `ed25519-dalek`

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.