Pashov Audit Group

Enclave Security Review

October 25th 2025 - October 27th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

• **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
• **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
• **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

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

4. About Enclave Solver Fund Pool

Enclave is a Solana program that provides a fund pool allowing users to borrow tokens instantly with off-chain authorization and replay-protected signatures.

5. Executive Summary

A time-boxed security review of the **Enclave-Money/solver-fund-pool-anchor** repository was done by Pashov Audit Group, during which **0xAlix2**, **ZeroTrust01**, **shaflow**, **Johny** engaged to review **Enclave Solver Fund Pool**. A total of **6** issues were uncovered.

Protocol Summary

Project Name	Enclave Solver Fund Pool
Protocol Type	Lending Pool
Timeline	October 25th 2025 - October 27th 2025

Review commit hash:

- 668adb2eac1ecb81843ffda226db1fd1e1ed9396 (Enclave-Money/solver-fund-pool-anchor)

Fixes review commit hash:

- 3034df5741fb9e3a1c0c912a5ccd8c28f9d90a55 (Enclave-Money/solver-fund-pool-anchor)

Scope

`admin_management.rs` `borrow.rs` `initialize.rs` `mod.rs` `signer_management.rs` `error.rs` `lib.rs` `state.rs` `utils.rs`

6. Findings

Findings count

Severity	Amount
Critical	2
Low	4
Total findings	**6**

Summary of findings

ID	Title	Severity	Status
[C-01]	Funds can be redirected	Critical	**Resolved**
[C-02]	Wrong-Offset Ed25519 instruction introspection enables forged approvals	Critical	**Resolved**
[L-01]	Program does not support `Token-2022` mints	Low	**Resolved**
[L-02]	`borrow` lacks `has_one = user` constraint on `user_borrow_state`	Low	**Resolved**
[L-03]	Inconsistent usage of seed constants and byte literals	Low	**Resolved**
[L-04]	Frontrunnable Initialization	Low	Acknowledged

Critical findings

[C-01] Funds can be redirected

Severity

Impact: High

Likelihood: High

Description

```
#[account(mut)]
   pub user_token_account: Account<'info, TokenAccount>
```

In `programs/solver-fund-pool-anchor/src/instructions/borrow.rs:142-143`, the `user_token_account` is only required to be writable; the program never checks whether `owner == user.key()`. The signed message also omits the `user_token_account`, and the contract does not validate that `user_token_account.owner` equals `user`, nor that it is the user's associated token account (ATA).

As a result, an attacker can obtain a legitimate signature (which only binds `user/mint/amount/time window/nonce/...`) but replace `user_token_account` with their own token account, siphoning pool funds directly to themselves.

Recommendations

```
#[account(
    mut,
    constraint = user_token_account.mint == token_mint.key() @
ErrorCode::InvalidTokenAccountMint,
    constraint = user_token_account.owner == user.key() @ ErrorCode::InvalidTokenAccountOwner,
)]
pub user_token_account: Account<'info, TokenAccount>,
```

Or restrict to the user's ATA:

```
use anchor_spl::associated_token::get_associated_token_address;
require_keys_eq!(
    user_token_account.key(),
    get_associated_token_address(&Pubkey::try_from(user.key())?, &token_mint.key()),
    ErrorCode::NotUserATA
);
```

[C-02] Wrong-Offset Ed25519 instruction introspection enables forged approvals

Severity

Impact: High

Likelihood: High

Description

The program accepts off-chain approvals by “verifying” an Ed25519 signature via the instruction sysvar: `borrow` expects the previous instruction in the same transaction to be a native Ed25519 verification and then inspects that instruction’s raw data to decide whether the signature is valid.

However, the current verifier does not perform the essential validations that make this pattern sound:

- It does not assert that the previous instruction’s `program_id` is the native Ed25519 program.
- It does not enforce “inline mode” by requiring the three `*_instruction_index` fields in `Ed25519SignatureOffsets` to be `0xFFFF` (meaning use data from this very Ed25519 instruction).
- It relies on reading specific byte ranges inside the Ed25519 instruction data, but does not validate that the supplied offsets and sizes actually point to those ranges.

On Solana, the Ed25519 verifier is flexible: the instruction can specify offsets and instruction indexes for where to read the signature, public key, and message. If a contract does not validate those fields, an attacker can craft a preceding instruction that causes the Ed25519 program to verify a signature over attacker-controlled data, while placing benign-looking bytes (the expected signer pubkey, message hash, signature) at the locations your program later checks. Your verifier will then return success even though no signature from an authorized signer was actually validated.

As a result, any caller can submit a transaction that includes a maliciously constructed “verification” instruction immediately before `borrow` and have the program accept it as valid. That enables unauthorized borrows from the pool and increments the user’s nonce, potentially blocking the legitimate user from later redeeming a genuine approval.

Recommendations

Harden the Ed25519 verification to treat the previous instruction as valid evidence only if all of the following are true:

- The previous instruction’s `program_id` is the native Ed25519 program.
- The instruction encodes exactly one signature and carries no auxiliary accounts.
- All three `*_instruction_index` fields in `Ed25519SignatureOffsets` are `0xFFFF` (inline mode), so the signature, public key, and message are taken from the same Ed25519 instruction.
- All offsets and sizes are bounds-checked against the instruction data length before slicing.
- The inline public key, message, and signature bytes exactly match the expected authorized signer and message.
- If `borrow` is the first instruction, reject (there is no “previous” verification to inspect).

Low findings

[L-01] Program does not support `Token-2022` mints

The program exclusively imports and uses the legacy SPL Token interface:

```
use anchor_spl::token::{self, Mint, Token, TokenAccount, Transfer};
pub token_program: Program<'info, Token>;
```

If a Token-2022 mint (owned by `spl_token_2022::ID`) is passed, CPIs will fail because the runtime enforces the correct program ID for each account type.

Recommendations:

Consider switching to Anchor's **token interface** layer, which is compatible with both SPL Token (legacy) and Token-2022 programs.

[L-02] `borrow` lacks `has_one = user` constraint on `user_borrow_state`

The `Borrow` accounts struct (`programs/solver-fund-pool-anchor/src/instructions/borrow.rs:109-128`) derives the user-specific state PDA with seeds `[USER_BORROW_SEED, user.key().as_ref()]` , but it never enforces that the stored `user` field inside `UserBorrowState` actually matches the `user` account provided in the instruction. If the account data is ever modified (e.g., via unchecked CPI or data corruption), the program will happily accept a mismatched tuple, causing the nonce owner stored on-chain to differ from the receiver used for token transfers and signature verification. This breaks critical assumptions about who owns the borrow nonce and can complicate replay protection or downstream accounting.

Recommendations Add `has_one = user` to the `user_borrow_state` account constraint so Anchor verifies the persisted `user` field matches the supplied `user` account, tightening the consistency checks around the PDA.

[L-03] Inconsistent usage of seed constants and byte literals

When checking the `fund_pool` address, the seeds used are inconsistent. Some places use the constant `FUND_POOL_SEED` , while others directly use the literal `b"fund_pool"` .

```
    #[account(
        mut,
        seeds = [b"fund_pool"],
        bump = fund_pool.bump,
        constraint = fund_pool.is_admin(&admin.key()) @ ErrorCode::Unauthorized
    )]
    pub fund_pool: Account<'info, FundPoolState>,
    #[account(
        seeds = [FUND_POOL_SEED],
        bump = fund_pool.bump
    )]
    pub fund_pool: Account<'info, FundPoolState>,
```

Although both are currently equivalent, accidental modification to one of them during upgrades and maintenance may cause inconsistencies.

It is recommended to consistently use the `FUND_POOL_SEED` constant to improve maintainability and reduce risks introduced by upgrades.

[L-04] Frontrunnable Initialization

The `initialize` instruction creates the global `fund_pool` PDA (`seed = b"fund_pool"`) and sets `initial_admin` to an arbitrary public key supplied by the caller. There is no access control restricting who may invoke this first-use initializer. An attacker can front-run deployment, initialize the pool, and seize control over all admin- and signer-gated operations for the lifetime of the program (until redeploy).

Recommendations

Implement a robust access control mechanism that ensures only a trusted entity, such as the program's deployer or a predefined address, can call the `initialize` function. This restriction can be enforced by verifying the caller's identity or using a specific signature during initialization.