



Pashov Audit Group

KeySol Security Review

January 17th 2026 - January 18th 2026



Contents

1. About Pashov Audit Group	3
2. Disclaimer	3
3. Risk Classification	3
4. About KeySol	4
5. Executive Summary	4
6. Findings	5
Critical findings	6
[C-01] Sellers can grief buyers by using <code>seller_cancel</code>	6
[C-02] Users can redirect fees to themselves instead of the intended treasury account	8
[C-03] Buyer can steal USDC via <code>buyer_approve</code>	9
[C-04] Buyers can deposit worthless tokens and get away with it	12
[C-05] Buyers can deposit to their own token accounts instead of <code>escrow_token_account</code>	14
High findings	16
[H-01] Buyers can grief sellers	16
Low findings	18
[L-01] Approved and canceled can both be true	18
[L-02] Lack of close mechanism	18
[L-03] Creating listing can be closed	18
[L-04] Missing price validation allows zero-value listings	19
[L-05] Buyers can shut down as many listings as they want without paying a single USDC	20



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 KeySol

KeySol is a Solana escrow contract system where buyers deposit tokens for specific listings and both buyer and seller must approve before funds are released to the seller minus a 0.5% fee. The project provides cancellation and dispute resolution mechanisms with administrative oversight for contested transactions.

5. Executive Summary

A time-boxed security review of the **keysol-official/keysol** repository was done by Pashov Audit Group, during which **newspace**, **JoVi**, **ctrus** engaged to review **KeySol**. A total of 11 issues were uncovered.

Protocol Summary

Project Name	KeySol
Protocol Type	Escrow system
Timeline	January 17th 2026 - January 18th 2026

Review commit hash:

- [c7475532860cb5f0a1ab96364c4d8deb5c2a5cee](#)
(keysol-official/keysol)

Fixes review commit hash:

- [9bfab72e8685d812a1a2b62eb0676a357856898e](#)
(keysol-official/keysol)

Scope

lib.rs



6. Findings

Findings count

Severity	Amount
Critical	5
High	1
Low	5
Total findings	11

Summary of findings

ID	Title	Severity	Status
[C-01]	Sellers can grief buyers by using <code>seller_cancel</code>	Critical	Resolved
[C-02]	Users can redirect fees to themselves instead of the intended treasury account	Critical	Resolved
[C-03]	Buyer can steal USDC via <code>buyer_approve</code>	Critical	Resolved
[C-04]	Buyers can deposit worthless tokens and get away with it	Critical	Resolved
[C-05]	Buyers can deposit to their own token accounts instead of <code>escrow_token_account</code>	Critical	Resolved
[H-01]	Buyers can grief sellers	High	Resolved
[L-01]	Approved and canceled can both be true	Low	Resolved
[L-02]	Lack of close mechanism	Low	Resolved
[L-03]	Creating listing can be closed	Low	Resolved
[L-04]	Missing price validation allows zero-value listings	Low	Resolved
[L-05]	Buyers can shut down as many listings as they want without paying a single USDC	Low	Acknowledged



Critical findings

[C-01] Sellers can grief buyers by using `seller_cancel`

Severity

Impact: High

Likelihood: High

Description

The `seller_cancel` instruction lacks validation on both `buyer_token_account` and `escrow_token_account` parameters. When the buyer has already cancelled (`escrow.buyer_cancelled == true`), a malicious seller can exploit these missing checks to either steal the buyer's funds or grief the buyer by refunding a dust amount instead of the original deposit. In the `SellerCancel` account struct, neither account is validated:

```
#[derive(Accounts)]
pub struct SellerCancel<'info> {
    #[account(mut)]
    pub seller: Signer<'info>,

    #[account(
        mut,
        seeds = [b"escrow", escrow.listing_id.as_bytes()],
        bump = escrow.bump
    )]
    pub escrow: Account<'info, Escrow>,

    #[account(mut)]
    pub escrow_token_account: Account<'info, TokenAccount>, // No validation

    #[account(mut)]
    pub buyer_token_account: Account<'info, TokenAccount>, // No ownership validation

    pub token_program: Program<'info, Token>,
}
```

The cancellation logic reads the amount from the passed `escrow_token_account` and transfers it to the passed `buyer_token_account`:

```
if ctx.accounts.escrow.buyer_cancelled {
    let amount = ctx.accounts.escrow_token_account.amount; // Reads from attacker-controlled account
    // ...
    let cpi_accounts = Transfer {
        from: ctx.accounts.escrow_token_account.to_account_info(),
        to: ctx.accounts.buyer_token_account.to_account_info(), // Attacker-controlled destination
        authority: ctx.accounts.escrow.to_account_info(),
    }
```



```
};  
token::transfer(cpi_ctx, amount)?;  
ctx.accounts.escrow.is_completed = true;  
}
```

Attack Scenario 1 - Direct Fund Theft:

- Buyer deposits 10,000 USDC into the legitimate escrow token account.
- Transaction falls through and buyer calls `buyer_cancel`, setting `[escrow.buyer_cancelled = true]`.
- Malicious seller calls `seller_cancel` but passes their own USDC token account as `buyer_token_account`.
- Since `[escrow.buyer_cancelled == true]`, the transfer executes.
- 10,000 USDC is transferred to the seller's account instead of refunding the buyer.
- `[escrow.is_completed = true]` prevents any further action (victim cannot even open a dispute).
- Buyer loses their entire deposit with no recourse.

Attack Scenario 2 - Refund Griefing:

- Buyer deposits 10,000 USDC into the legitimate escrow token account (Account A)
- Transaction falls through and buyer calls `buyer_cancel`
- Malicious seller: Creates a new token account (Account B) owned by the escrow PDA
Funds Account B with 1 USDC (dust amount)
- Seller calls `seller_cancel` passing Account B as `escrow_token_account` and the legitimate buyer token account
- The function reads `[amount = 1 USDC]` from Account B
- Buyer receives 1 USDC refund instead of 10,000 USDC
- `[escrow.is_completed = true]` finalizes the theft
- The original 10,000 USDC in Account A remains locked and Buyer loses 99.99% of their deposit The second attack is applicable on `BuyerCancel` as well; implement the same recommendations there as well.

Both attacks result in significant financial loss for the buyer who legitimately requested a cancellation, with no ability to dispute or recover funds since the escrow is marked as complete.

Recommendations

Store these token accounts in Escrow itself and then match them against user-supplied accounts.



[C-02] Users can redirect fees to themselves instead of the intended treasury account

Severity

Impact: High

Likelihood: High

Description

Multiple instructions ([seller_approve](#), [buyer_approve](#), [resolve_dispute](#)) transfer protocol fees to a `treasury_token_account` without validating that this account is actually owned by the intended treasury address defined in [TREASURY_PUBKEY](#). A malicious user can pass their own token account as the treasury account, redirecting all protocol fees to themselves.

The treasury public key is defined as a constant:

```
pub const TREASURY_PUBKEY: &str = "CrLuVTFN4TJ6eCB9t5DZppeVVHkoDbra3rxMUXMUTqn";
pub const FEE_BASIS_POINTS: u64 = 50;
```

However, in all account structs that use `treasury_token_account`, there is no validation:

```
#[derive(Accounts)]
pub struct SellerApprove<'info> {
    // ...
    #[account(mut)]
    pub treasury_token_account: Account<'info, TokenAccount>, // No ownership validation
    // ...
}

#[derive(Accounts)]
pub struct BuyerApprove<'info> {
    // ...
    #[account(mut)]
    pub treasury_token_account: Account<'info, TokenAccount>, // No ownership validation
    // ...
}

#[derive(Accounts)]
pub struct ResolveDispute<'info> {
    // ...
    #[account(mut)]
    pub treasury_token_account: Account<'info, TokenAccount>, // No ownership validation
    // ...
}
```

The fee transfer logic executes without any checks:

```
let cpi_accounts_fee = Transfer {
    from: ctx.accounts.escrow_token_account.to_account_info(),
    to: ctx.accounts.treasury_token_account.to_account_info(), // Attacker-controlled
```



```
    authority: ctx.accounts.escrow.to_account_info(),
  };
  let cpi_ctx_fee = CpiContext::new_with_signer(
    ctx.accounts.token_program.to_account_info(),
    cpi_accounts_fee,
    signer,
  );
  token::transfer(cpi_ctx_fee, fee)?;
```

An example attack scenario:

- A legitimate escrow is created with a 10,000 USDC price.
- Buyer deposits 10,000 USDC into the escrow.
- Seller calls [seller_approve](#) first, setting [escrow.seller_approved = true].
- Buyer calls [buyer_approve](#) but passes their own USDC token account as `treasury_token_account`.
- The transfer logic executes: 9,950 USDC (99.5%) goes to the seller. 50 USDC (0.5% fee) goes to the attacker's account instead of the treasury.
- The protocol loses all fee revenue.

This attack can be executed on every single transaction, resulting in a complete loss of protocol fees. Given the 0.5% fee rate, over time this represents significant losses for the protocol.

Recommendations

Add treasury ownership validation to all affected account structures.

[C-03] Buyer can steal USDC via `buyer_approve`

Severity

Impact: High

Likelihood: High

Description

The [buyer_approve](#) instruction does not validate that the `seller_token_account` is actually owned by the seller stored in the escrow. When the seller has already approved the transaction (`escrow.seller_approved == true`), a malicious buyer can pass their own token account as the `seller_token_account` parameter, redirecting the escrowed funds to themselves instead of the legitimate seller.

In the `BuyerApprove` account struct, the `seller_token_account` lacks any ownership constraint:



```
#[derive(Accounts)]
pub struct BuyerApprove<'info> {
    #[account(mut)]
    pub buyer: Signer<'info>,

    #[account(
        mut,
        seeds = [b"escrow", escrow.listing_id.as_bytes()],
        bump = escrow.bump
    )]
    pub escrow: Account<'info, Escrow>,

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

    #[account(mut)]
    pub seller_token_account: Account<'info, TokenAccount>, // No ownership validation

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

    pub token_program: Program<'info, Token>,
}
```

When the seller has already approved, the `buyer_approve` function executes the fund transfer:

```
if ctx.accounts.escrow.seller_approved {
    let amount = ctx.accounts.escrow_token_account.amount;
    let fee = amount * FEE_BASIS_POINTS / 10000;
    let seller_amount = amount - fee;

    // ...
    let cpi_accounts = Transfer {
        from: ctx.accounts.escrow_token_account.to_account_info(),
        to: ctx.accounts.seller_token_account.to_account_info(), // Attacker-controlled
        authority: ctx.accounts.escrow.to_account_info(),
    };
    // Transfer executes to attacker's account
    token::transfer(cpi_ctx, seller_amount)?;
    // ...
}
```



Here is an example attack scenario:

- Seller creates a listing for 1000 USDC
- Buyer deposits 1000 USDC into the legitimate escrow token account
- Seller calls `seller_approve`, setting `[escrow.seller_approved = true]`
- Seller expects the buyer to approve so they receive payment
- Malicious buyer calls `buyer_approve` but passes their own USDC token account as `seller_token_account`
- The condition `[escrow.seller_approved == true]` is satisfied, triggering the transfer 995 USDC (minus 0.5% fee) is transferred to the buyer's own account instead of the seller
- 5 USDC fee goes to the treasury
- `[escrow.is_completed]` is set to true, finalizing the theft
- The buyer receives both their deposited funds back and retains whatever asset/service the seller provided

Recommendations

Add ownership validation constraints in `BuyerApprove` as well as `SellerApprove`.

```
#[derive(Accounts)]
pub struct BuyerApprove<'info> {
    #[account(mut)]
    pub buyer: Signer<'info>,

    #[account(
        mut,
        seeds = [b"escrow", escrow.listing_id.as_bytes()],
        bump = escrow.bump
    )]
    pub escrow: Account<'info, Escrow>,

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

    #[account(
        mut,
        constraint = seller_token_account.owner == escrow.seller @
KeySolError::InvalidSellerTokenAccount
    )]
    pub seller_token_account: Account<'info, TokenAccount>,

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

    pub token_program: Program<'info, Token>,
}
```



[C-04] Buyers can deposit worthless tokens and get away with it

Severity

Impact: High

Likelihood: High

Description

The `deposit` instruction does not validate that the `escrow_token_account` holds the intended token mint (USDC). A malicious buyer can create a token account owned by the escrow PDA but holding a worthless or attacker-controlled token mint, effectively paying nothing of value while marking the escrow as funded.

In the `Deposit` account struct, there is no constraint verifying the mint of the token accounts:

```
#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub buyer: Signer<'info>,

    #[account(
        mut,
        seeds = [b"escrow", escrow.listing_id.as_bytes()],
        bump = escrow.bump
    )]
    pub escrow: Account<'info, Escrow>,

    #[account(mut)]
    pub buyer_token_account: Account<'info, TokenAccount>, // No mint validation

    #[account(mut)]
    pub escrow_token_account: Account<'info, TokenAccount>, // No mint validation

    pub token_program: Program<'info, Token>,
}
```

The transfer succeeds as long as both token accounts share the same mint, but there is no enforcement that this mint is USDC or any legitimate payment token.



Here is an example attack scenario:

- Seller creates a listing expecting payment in USDC with `price = 1000000000`
- Attacker creates a new worthless token mint they control
- Attacker creates a token account for the escrow PDA with their worthless mint as the token type
- Attacker creates their own token account with the same worthless mint and mints arbitrary amounts to it
- Attacker calls `deposit`, passing both token accounts with the worthless mint
- The transfer of `escrow.price` amount succeeds (transferring worthless tokens; this could be way less than 1000 USDC) `escrow.deposited` is set to true, indicating funds are transferred
- When both parties approve, the seller receives worthless tokens instead of USDC
- The seller loses their asset/service while the buyer pays nothing of value

Note: This vulnerability is also present in and affects the approval and cancellation flows since they read `escrow_token_account.amount` without verifying the mint, allowing manipulation of fee calculations and transfer amounts.

Recommendations

Add appropriate mint validations.

```
pub const USDC_MINT: &str = "EPjFWdd5AufqSSqeM2qN1xzybapC8G4wEGGkZwyTDt1v"; // Mainnet USDC

#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub buyer: Signer<'info>,

    #[account(
        mut,
        seeds = [b"escrow", escrow.listing_id.as_bytes()],
        bump = escrow.bump
    )]
    pub escrow: Account<'info, Escrow>,

    /// The USDC mint account
    #[account(
        address = USDC_MINT.parse::<Pubkey>().unwrap() @ KeySolError::InvalidMint
    )]
    pub usdc_mint: Account<'info, Mint>,

    #[account(
        mut,
        constraint = buyer_token_account.mint == usdc_mint.key() @ KeySolError::InvalidMint,
        constraint = buyer_token_account.owner == buyer.key() @
KeySolError::InvalidTokenAccountOwner
    )]
    pub buyer_token_account: Account<'info, TokenAccount>,
```



```
#[account(
    mut,
    constraint = escrow_token_account.mint == usdc_mint.key() @ KeySolError::InvalidMint,
    constraint = escrow_token_account.owner == escrow.key() @
KeySolError::InvalidEscrowTokenAccount
)]
pub escrow_token_account: Account<'info, TokenAccount>,

pub token_program: Program<'info, Token>,
}
```

alternate solution could be to store both mint address and escrow token account address in escrow itself and then match the user-passed token account against those stored addresses in escrow.

[C-05] Buyers can deposit to their own token accounts instead of `escrow_token_account`

Severity

Impact: High

Likelihood: High

Description

The `deposit` instruction lacks validation on the `escrow_token_account` parameter, allowing a malicious buyer to pass their own controlled token account instead of the legitimate escrow-owned token account.

In the `Deposit` account struct, the `escrow token account` is defined without any ownership or authority constraints:

```
#[derive(Accounts)]
pub struct Deposit<'info> {
    #[account(mut)]
    pub buyer: Signer<'info>,

    #[account(
        mut,
        seeds = [b"escrow", escrow.listing_id.as_bytes()],
        bump = escrow.bump
    )]
    pub escrow: Account<'info, Escrow>,

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

    #[account(mut)]
    pub escrow_token_account: Account<'info, TokenAccount>, // No ownership validation
```



```
pub token_program: Program<'info, Token>,
}
```

The transfer logic then moves tokens from the buyer's account to whatever account is passed as `escrow_token_account` :

```
let cpi_accounts = Transfer {
  from: ctx.accounts.buyer_token_account.to_account_info(),
  to: ctx.accounts.escrow_token_account.to_account_info(),
  authority: ctx.accounts.buyer.to_account_info(),
};
```

Here is a simple attack scenario:

- A seller creates a legitimate listing with a valid escrow PDA.
- A malicious buyer calls `deposit` but passes their own token account (or any account they control) as `escrow_token_account` .
- The buyer effectively transfers tokens to themselves while the escrow's `deposited` flag is set to true.
- The escrow state now shows funds are deposited, but the actual escrow token account controlled by the escrow PDA has a zero balance. To avoid failing the transaction, the buyer can transfer a few USDC to this original escrow token account; now it has a non-zero balance but still very little compared to what was intended.
- When the transaction is approved by both parties, the transfer from the escrow token account will result in the seller receiving less than intended, leaving the seller with less payment (what the buyer later transfers to the intended token account in step 4).

Recommendations

implement an ownership check that makes sure `escrow_token_account` is owned by Escrow PDA.

```
#[account(
  mut,
  constraint = escrow_token_account.owner == escrow.key() @
  KeySolError::InvalidEscrowTokenAccount
)]
pub escrow_token_account: Account<'info, TokenAccount>,
```



High findings

[H-01] Buyers can grief sellers

Severity

Impact: High

Likelihood: Medium

Description

The [buyer_approve](#) and [seller_approve](#) instructions do not validate that the `escrow_token_account` is the legitimate token account where the buyer originally deposited funds. A malicious party can pass a different token account (owned by the escrow PDA but funded with dust amounts), causing the seller to receive significantly less than the agreed price.

In both approval structs, there is no validation linking `escrow_token_account` to the original deposit:

```
#[derive(Accounts)]
pub struct BuyerApprove<'info> {
    // ...
    #[account(mut)]
    pub escrow_token_account: Account<'info, TokenAccount>, // No validation against original
    deposit
    // ...
}
```

The transfer amount is derived directly from the passed account's balance:

```
if ctx.accounts.escrow.seller_approved {
    let amount = ctx.accounts.escrow_token_account.amount; // Reads from attacker-controlled
    account
    let fee = amount * FEE_BASIS_POINTS / 10000;
    let seller_amount = amount - fee;
    // ...
    token::transfer(cpi_ctx, seller_amount)?; // Transfers dust amount
}
```



An example scenario:

- Seller creates a listing for 10,000 USDC
- Buyer deposits 10,000 USDC into the legitimate escrow token account (Account A)
- Buyer receives the item/service in real life
- Seller calls `seller_approve`, setting `[escrow.seller_approved = true]`
- Malicious buyer: Creates a new token account (Account B) with the escrow PDA as owner
Funds Account B with 1 USDC (dust amount)
- Buyer calls `buyer_approve`, passing Account B as `escrow_token_account`
- The function reads `amount = 1 USDC`
- Seller receives 0.995 USDC instead of 9,950 USDC
- Treasury receives 0.005 USDC fee
- `[escrow.is_completed = true]`, preventing any dispute; seller cannot raise a dispute because escrow has completed already, seller lost his entitled 10,000 USDC.
- The original 10,000 USDC in Account A remains locked forever

Recommendations

Store the escrow token account address in the `Escrow` struct during the deposit and validate the user's passed account against it.



Low findings

[L-01] Approved and canceled can both be true

Description

The contract allows both approval and cancellation flags to be `true` simultaneously for the same party, creating a logically inconsistent state. There are no mutual exclusion checks between approve and cancel functions.

The escrow state could be:

```
buyer_approved = true  
buyer_cancelled = true
```

This is semantically meaningless; the buyer cannot simultaneously approve and cancel the transaction.

[L-02] Lack of close mechanism

Description

The contract lacks a close function. Once created, escrow accounts remain forever, even if they are completed or created by mistake.

- The seller may want to close the escrow before any deposits are made, but currently, they cannot. As a result, multiple buyers can still deposit into it, even though the seller no longer intends to proceed with the sale, making any further operations unnecessary.
- Old completed escrows cannot be cleaned up, which leads to unnecessary rent costs and storage bloat.

Recommendation

Add a close instruction.

[L-03] Creating listing can be closed

Description

The `create_listing` function derives the escrow PDA using only the `listing_id` string, without incorporating the seller's public key. This allows malicious actors to frontrun listing creation and create escrows with the same `[listing_id]` before the legitimate seller's transaction is processed.

Recommendation:



Include the seller's public key in the PDA seeds to ensure each seller has their own namespace for listing IDs:

```
#[derive(Accounts)]
#[instruction(listing_id: String)]
pub struct CreateListing<'info> {
    #[account(mut)]
    pub seller: Signer<'info>,

    #[account(
        init,
        payer = seller,
        space = 8 + Escrow::INIT_SPACE,
        seeds = [b"escrow", seller.key().as_ref(), listing_id.as_bytes()],
        bump
    )]
    pub escrow: Account<'info, Escrow>,

    pub system_program: Program<'info, System>,
}
```

[L-04] Missing price validation allows zero-value listings

Description

The [create_listing](#) function does not validate that the `price` parameter is greater than zero. Given that there are no fees to create a listing, this allows sellers to create listings with a price of 0, which can spam the platform with garbage listings.

Add a minimum price check:

```
pub const MINIMUM_PRICE: u64 = 1_000_000; // 1 USDC

pub fn create_listing(
    ctx: Context<CreateListing>,
    listing_id: String,
    price: u64,
) -> Result<()> {
    require!(price >= MINIMUM_PRICE, KeySolError::PriceTooLow);

    let escrow = &mut ctx.accounts.escrow;
    escrow.listing_id = listing_id;
    escrow.seller = ctx.accounts.seller.key();
    // ...existing code...
}

#[error_code]
pub enum KeySolError {
    // ...existing code...
    #[msg("Price must be greater than minimum allowed")]
    PriceTooLow,
}
```



[L-05] Buyers can shut down as many listings as they want without paying a single USDC

Description

The escrow system allows any buyer to deposit funds into any listing, then immediately initiate a cancellation or dispute process to close the listing permanently. Since the `[is_completed]` flag is set to true after resolution, the listing becomes permanently unusable. This attack costs the buyer nothing (they get their full deposit back) while forcing sellers to lose their rent payment of escrow creation and recreate new listings for the same goods.

Even in the case when the seller does not agree to cancel the request, the buyer can always open a dispute and get back the full deposit and hence complete the listing.

Attack Scenario:

- Seller creates a listing, paying rent (~0.002 SOL for the escrow account)
- Attacker deposits the exact `[escrow.price]` in USDC to become the buyer
- Attacker immediately calls `[buyer_cancel]`, setting `buyer_cancelled` = true
- If seller cooperates: Seller calls `[seller_cancel]`, buyer gets a full refund, listing is permanently closed
- If seller does not cooperate: Attacker calls `open_dispute`, then waits for the admin to resolve. Admin has no choice but to refund the buyer (since no goods were exchanged), setting `[is_completed] = true`
- In both cases, the listing is now permanently unusable
- Seller loses rent payment and must create a new listing
- Attacker repeats this on hundreds of listings at zero cost (only transaction fees)

This is effectively destroying the entire listing system and core purpose of the protocol as malicious buyers with huge amounts of money can keep many listings in a limbo state, hence preventing genuine users from using the protocol in the intended manner.

Recommendations

Add a non-refundable deposit fee for buyers, a small amount, just big enough to act as a barrier.