Jet v2

Audit

Presented by:

OtterSec contact@osec.io

Harrison Green hgarrereyn@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

01 | Executive Summary

Overview

Jet Protocol engaged OtterSec to perform an assessment of the `jet-v2` program prior to deployment.

This assessment was conducted between March 28th and April 18th. We later performed followup patch confirmation between July 13th and July 24th.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation.

Key Findings

The following is a summary of the major findings in this audit.

- 12 findings total
- 4 vulnerabilities which could lead to loss of funds
    - OS-JET-ADV-00: Invalid authority check
    - OS-JET-ADV-01: Position `TokenAccount` zeroing
    - OS-JET-ADV-02: Improper `MarginPool` rounding
    - OS-JET-ADV-03: Untrusted program invocation in margin-swap

As part of this audit, we also provided proof of concepts for each vulnerability to prove exploitability and enable simple regression testing. These scripts can be found at https://osec.io/pocs/jet-v2. For a full list, see Appendix C.

We also observed the following:

- Team was very knowledgeable and responsive, pushing patches throughout the audit
- Code quality of the program was high and overall design was solid
- The use of Anchor and a comprehensive internal audit mitigated many implementation level bugs

02 | Scope

We received the program from Jet Protocol and began the audit March 28th. The source code was delivered to us in a git repository at https://github.com/jet-lab/jet-v2. This audit was performed against commit fdd9b91.

There were a total of six programs included in this audit. A brief description of the six programs is as follows. A full list of program files and hashes can be found in Appendix A.

Name	Description
control	High-level trusted instructions for registering adapters and creating pools.
margin	Core MarginAccount utilities and adapter program instructions.
margin-pool	Adapter program for token storage, deposits, withdrawals, borrowing in a trusted vault.
margin-serum	Adapter program for swaps through serum
margin-swap	Adapter program to provide an interface to the spl_token_swap program via margin pools.
metadata	Utilities for creating and configuring metadata accounts used in other programs.

As part of Jet's deployment sequence, we also received an internal audit report prepared by three engineers not involved in the implementation of the program.

03 | 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 onchain program. In other words, there is no way to steal tokens or deny service, ignoring any Solana specific quirks such as account ownership issues. An example of a design vulnerability would be an onchain oracle which could be manipulated by flash loans or large deposits.

On the other hand, auditing the implementation of the program requires a deep understanding of Solana's execution model. Some common implementation vulnerabilities include account ownership issues, arithmetic overflows, and rounding bugs. For a non-exhaustive list of security issues we check for, see Appendix B. One such issue that was identified here enables attackers to steal tokens from a margin pool by exploiting a rounding bug in deposit token/note conversion (OS-JET-ADV-02).

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. For example, we demonstrate that a lack of account checks in the margin-swap program allows an attacker to break security assumptions about cross-program invocation interactions and erase debt by burning loan notes (OS-JET-ADV-03).

We also aim to provide meaningful suggestions to reduce the overall program attack surface. For example, with OS-JET-SUG-00, we help mitigate the impact of adapter vulnerabilities by removing unnecessary signing of the margin account during accounting updates.

As we approach any new target, we strive to get a comprehensive understanding of the program first. In our audits, we always approach any target in a team of two. This allows us to share thoughts and collaborate, picking up on details that the other missed.

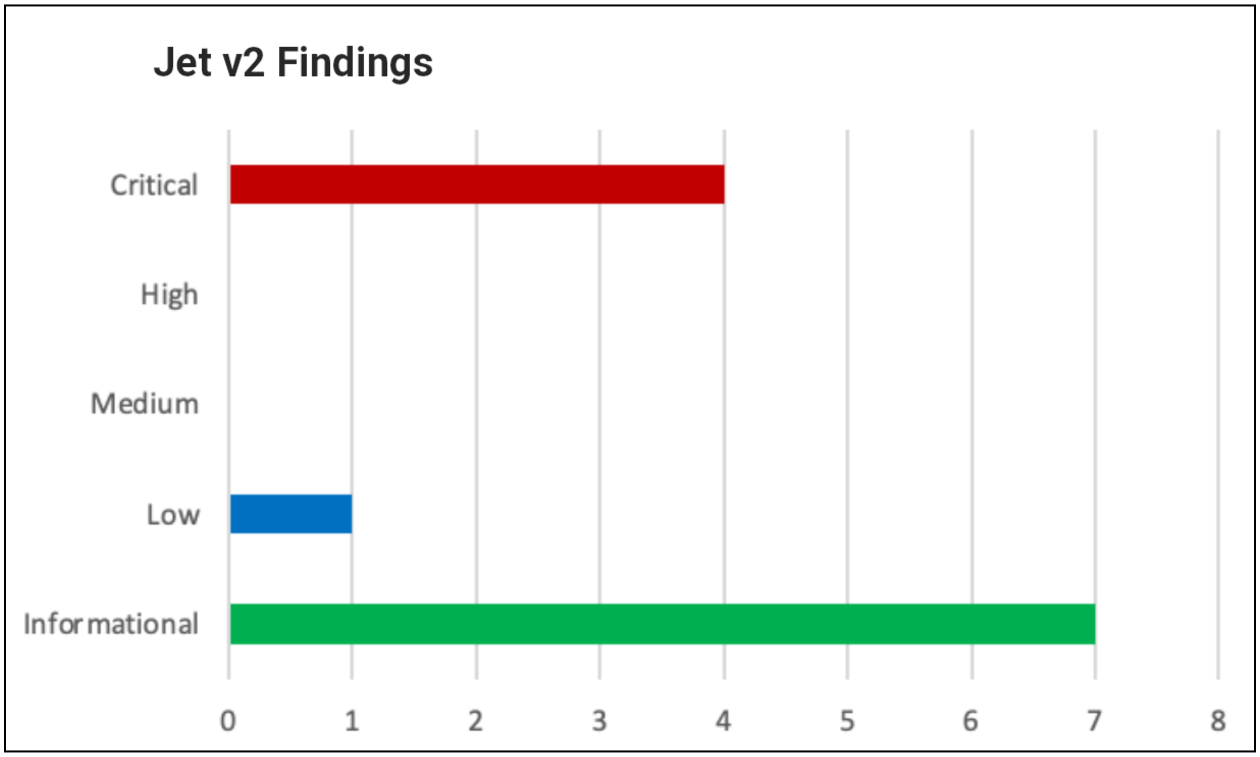
While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.

04 | Findings

Overall, we report 12 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 don't have an immediate impact but will help mitigate future vulnerabilities.

The below chart displays the findings by severity.



Proof of Concepts

For each vulnerability we created a proof of concept to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite. The proof of concept directory structure can be found in Appendix C.

These proof of concepts can be found at https://github.com/jet-lab/jet-v2.

To run a POC:

```
./run.sh <directory name>
```

For example,

```
./run.sh os-jet-adv-00
```

The relevant code for each POC can be found in `<dir>/patch`. These POCs make use of the existing test framework in Jet v2.

05 | Vulnerabilities

Here we present a technical analysis of the vulnerabilities we identified during our audit of the Jet v2 program. These vulnerabilities have **immediate** security implications, and we recommend remediation as soon as possible.

Rating criterion can be found in Appendix E.

ID	Severity	Status	Description
OS-JET-ADV-00	**Critical**	**Resolved**	Invalid authority checks allow unrestricted privileged configuration invocation.
OS-JET-ADV-01	**Critical**	**Resolved**	Lack of checks in UpdatePositionBalance can lead to token account zeroing and loss of funds.
OS-JET-ADV-02	**Critical**	**Resolved**	Improper rounding in margin-pool enables an attacker to steal tokens.
OS-JET-ADV-03	**Critical**	**Resolved**	Unrestricted CPI through margin-swap leads to loss of all pool funds.
OS-JET-ADV-04	**Low**	**Resolved**	It is not possible to close positions in a Margin account.

OS-JET-ADV-00 [Critical]: Invalid Authority Check

Description

Authority checks throughout the margin program are performed in an insecure manner.

As seen below in the metadata handler, the authority key is checked to be owned by the `CONTROL_PROGRAM_ID`:

metadata/src/lib.rs

```
    /// The authority that must sign to make this change
    #[cfg_attr(not(feature = "devnet"), account(owner =
CONTROL_PROGRAM_ID))]
    pub authority: Signer<'info>,

    /// The address paying the rent for the account
    #[account(mut)]
    pub payer: Signer<'info>,
```

This behavior is also found in the `margin-pool` creation and configuration handlers.

margin-pool/src/instructions/configure.rs

```
#[derive(Accounts)]
pub struct Configure<'info> {
    /// The pool to be configured
    #[account(mut)]
    pub margin_pool: Account<'info, MarginPool>,

    /// The authority allowed to modify the pool, which must sign
    #[account(owner = CONTROL_PROGRAM_ID)]
    pub authority: Signer<'info>,

    pub pyth_product: AccountInfo<'info>,
    pub pyth_price: AccountInfo<'info>,
}
```

margin-pool/src/instructions/create_pool.rs

```
#[derive(Accounts)]
pub struct CreatePool<'info> {
    ...
    /// The authority to create pools, which must sign
    #[account(owner = CONTROL_PROGRAM_ID)]
    pub authority: Signer<'info>,
}
```

In the expected flow, a user invokes `control::CreateAuthority` to create an `Authority` account owned by the control program.

Then a trusted user invokes instructions on the control program which performs additional validation on the calling user.

control/src/instructions/configure_token.rs

```
pub struct ConfigureToken<'info> {
    #[cfg_attr(not(feature = "devnet"), account(address =
crate::ROOT_AUTHORITY))]
    pub requester: Signer<'info>,
```

This adapter then performs a cross-program invocation to `margin-pool::CreatePool` and `margin-pool::Configure`, passing the provided `Authority` as proof that the instruction is being invoked from the control program.

In other words, the authority signer check is meant to ensure that the CPI is performed by the control program.

The ownership check is insufficient however.

In the Solana account model, any user can create an account and assign ownership to the Jet control program. This account can then be used to invoke `margin-pool::CreatePool` and `margin-pool::Configure` directly.

Proof of Concept

1. Attacker creates a new account and assigns ownership to the Jet control program.

2. Attacker invokes `margin-pool::Configure` on existing pools to compromise user funds (for example by changing the fee_destination/management_fee_rate and invoking `margin-pool::Collect`).

Remediation

The usage of the `Authority` account is intended to ensure that CreatePool/Configure are invoked as a cross-program invocation via the Control program.

There are several alternatives to enforce this:

1. **Enforce account type/ownership with Anchor:**

```
#[derive(Accounts)]
pub struct CreatePool<'info> {
    ...
    /// The authority to create pools, which must sign
    #[account(signer)]
    pub authority: Account<'info, Authority>,
}
```

This specification requires:

- authority is a signer
- authority is owned by control program (since Authority struct is defined in control)
- authority account **has the Authority type**

Account reassignment requires that the data is zeroed. Since this check validates the first 8 bytes in the account data, it is not possible for an attacker to forge authority via ownership reassignment.

2. **Use a signed PDA**

```
#[derive(Accounts)]
pub struct CreatePool<'info> {
    ...
    /// The authority to create pools, which must sign
    #[account(seeds = [
                b"CreatePool".as_ref()
              ],
              bump
              seeds::program = CONTROL_PROGRAM_ID)]
    pub authority: Signer<'info>,
}
```

This specification requires:

- authority is a PDA derived from the control program with seed “CreatePool”
- authority is a signer

Only the program that a PDA is derived from can sign for it and therefore we can be sure that this instruction was initiated by control. As an additional benefit, using a PDA ensures that signatures correspond to exactly one instruction. In other words, a control program’s signature for `CreatePool` cannot be reused for other instructions such as `Configure`.

With this approach, the control program would need to be modified to invoke `CreatePool` with a PDA and sign for it.

This method also removes the need for an on-chain account and the `control::CreateAuthority` command.

Patch

A ControlAuthority PDA owned by the program is now used to verify cross-program invocation originating from the control program. These instructions ensure the ControlAuthority is both a signer and is owned by the control program.

OS-JET-ADV-01 [Critical]: Position Collateral Zeroing

Description

The update_position_balance_handle method does not properly check to make sure that the TokenAccount is registered in the MarginAccount.

margin/src/instructions/update_position_balance.rs

```
#[derive(Accounts)]
pub struct UpdatePositionBalance<'info> {
    /// The account to update
    #[account(mut)]
    pub margin_account: AccountLoader<'info, MarginAccount>,

    /// The token account to update the balance for
    #[account(constraint = token_account.owner ==
margin_account.key())]
    pub token_account: Account<'info, TokenAccount>,
}

pub fn update_position_balance_handler(ctx:
Context<UpdatePositionBalance>) -> Result<()> {
    let mut margin_account = ctx.accounts.margin_account.load_mut()?;
    let token_account = &ctx.accounts.token_account;

    margin_account.set_position_balance(&token_account.mint,
token_account.amount)?;

    Ok(())
}
```

The root cause of this vulnerability is a mistaken assumption that each TokenAccount is unique per user-mint pair.

This is true in the context of a MarginAccount where the token_account is a PDA derived from the mint. However, this is not true for SPL TokenAccounts in general.

As described in the motivation behind the associated token account program

> **A user may own arbitrarily many token accounts belonging to the same mint** which makes it difficult for other users to know which account they should send tokens to and introduces friction into many other aspects of token management.

In particular,

> In addition, **it allows a user to send tokens to another user even if the beneficiary does not yet have a token account for that mint**. Unlike a system transfer, for a token transfer to succeed the recipient must have a token account with the compatible mint already, and somebody needs to fund that token account.

In other words, a malicious user is able to create a token account for our victim and then update their position balance with this newly initialized token account. The margin program will think that the user's position with respect to that token is now zero.

The malicious user can then perform a liquidation to steal a victim's collateral.

Proof of Concept

Consider the following steps

1. The victim deposits 100 USDC and 100 USDT of collateral and borrows $100 of SOL.
2. The attacker creates a USDC token account for the victim. Note that this token account is initialized with 0 USDC.
3. The attacker updates the victim's position with respect to USDC. The margin program now thinks the user is undercollateralized.
4. The attacker atomically (in the same transaction) liquidates the victim, getting the 100 USDT.

Remediation

It is critical that the correct TokenAccount is passed into `UpdatePositionBalance`.

One way to do this would be to ensures that the provided token_account matches the expected PDA for that margin_account/mint pair as generated in `RegisterPosition`:

margin/src/instructions/register_position.rs

```
#[derive(Accounts)]
pub struct RegisterPosition<'info> {
    ...
    /// The mint for the position token being registered
    pub position_token_mint: Account<'info, Mint>,

    ...
    /// The token account to store hold the position assets in the
custody of the
    /// margin account.
    #[account(init,
              seeds = [
                  margin_account.key().as_ref(),
                  position_token_mint.key().as_ref()
              ],
              bump,
              payer = payer,
              token::mint = position_token_mint,
              token::authority = margin_account
    )]
    pub token_account: Account<'info, TokenAccount>,
    ...
}
```

Specifically, consider an UpdatePositionBalance instruction such as:

```
#[derive(Accounts)]
pub struct UpdatePositionBalance<'info> {
    /// The account to update
    #[account(mut)]
    pub margin_account: AccountLoader<'info, MarginAccount>,

    /// The mint for the position token
    pub position_token_mint: Account<'info, Mint>,

    /// The token account to update the balance for
    #[account(seeds = [
                margin_account.key().as_ref(),
                position_token_mint.key().as_ref()
             ],
             bump,
             token::mint = position_token_mint,
             token::authority = margin_account)]
    pub token_account: Account<'info, TokenAccount>,
}
```

With this approach, we verify that token_account passed into position updates is a PDA derived from the margin program with the same seed as used to initialize the token_account.

This makes it impossible for an attacker to pass in a malicious token account.

Patch

The user's token account address is now verified against the registered position, preventing this vulnerability.

OS-JET-ADV-02 [Critical]: Improper MarginPool Rounding

Description

There is a rounding bug in `MarginPool::convert_amount` that enables an attacker to withdraw tokens from a margin pool without owning any deposit notes.

margin-pool/src/state.rs

```
fn convert_amount(&self, amount: Amount, exchange_rate: Number) ->
FullAmount {
    match amount.kind {
        AmountKind::Tokens => FullAmount {
            tokens: amount.value,
            notes: (Number::from(amount.value) /
exchange_rate).as_u64(0),
        },

        AmountKind::Notes => FullAmount {
            notes: amount.value,
            tokens: (Number::from(amount.value) *
exchange_rate).as_u64(0),
        },
    }
}
```

When converting from a token amount to a notes amount (i.e. amount.kind is AmountKind::Token), the function divides the token amount by the current exchange_rate (and implicitly rounds down to the nearest integer).

Given an exchange_rate > 1, a token amount of 1 will be converted to a notes amount of 0.

This token/note conversion logic is used by the withdraw function:

margin-pool/src/instructions/withdraw.rs

```
pub fn withdraw_handler(ctx: Context<Withdraw>, amount: Amount) ->
Result<()> {
    let pool = &mut ctx.accounts.margin_pool;
    let clock = Clock::get()?;
    // Make sure interest accrual is up-to-date
    if !pool.accrue_interest(clock.unix_timestamp) {
        msg!("interest accrual is too far behind");
        return Err(ErrorCode::InterestAccrualBehind.into());
    }

    let deposit_amount = pool.convert_deposit_amount(amount); // [0]
    pool.withdraw(&deposit_amount)?;

    let pool = &ctx.accounts.margin_pool;
    let signer = [&pool.signer_seeds()?[..]];

    token::transfer(
        ctx.accounts.transfer_context().with_signer(&signer),
        deposit_amount.tokens,
    )?;
    token::burn(
        ctx.accounts.burn_note_context().with_signer(&signer),
        deposit_amount.notes,
    )?;

    Ok(())
}
```

Specifically, the withdraw handler takes the *user-provided amount value* and calls `MarginPool::convert_deposit_amount` which internally calls `Pool::convert_amount` [0].

Importantly, because the withdraw handler lets a user specify *either a token amount or a notes amount*, an attacker can specify a token amount of 1 which may be incorrectly converted to a notes amount of 0. Such a transaction would let the attacker withdraw 1 deposit token while spending 0 deposit notes.

Since this attack requires an exchange_rate > 1, when is this possible?

The deposit exchange rate is computed by MarginPool::deposit_note_exchange_rate:

margin-pool/src/state.rs

```
fn deposit_note_exchange_rate(&self) -> Number {
    let deposit_notes = std::cmp::max(1, self.deposit_notes);
    let total_value = std::cmp::max(Number::ONE, self.total_value());
    (total_value - *self.total_uncollected_fees()) /
Number::from(deposit_notes)
}
```

In short, the formula used is:

$$\frac{max(1, borrowed\_tokens + deposit\_tokens) - uncollected\_fees}{max(1, deposit\_notes)}$$

Initially the ratio is 1. The core logic in the `Deposit`, `Withdraw`, `Borrow` and `Repay` instructions does not affect this ratio. However, if time passes while a user has borrowed funds, these instructions will invoke `MarginPool::accrue_interest` which will increase both `borrowed_tokens` and `uncollected_fees`, thereby increasing the ratio above 1.

Proof of Concept

Exploitation requires several steps:

1. User A deposits tokens into a pool
2. User B borrows tokens from the pool
3. Time passes and interest accrues on B's borrow (even 1 second is enough)
4. User C can issue "free" Withdraw instructions for 1 token repeatedly to steal all available funds in the pool.

Remediation

Rounding is usually a necessity with value conversions. However, it should never be *advantageous for the user*. Rounding bugs that are advantageous for the user can be exploited to steal money (as demonstrated here).

In consideration of the `Deposit`/`Withdraw` flow, the following principles should be used:

- In `Deposit`, users specify a *token amount*. Rounding may cause users to receive *fewer* notes (but never more).
- In `Withdraw`, users specify a *notes amount*. Rounding may cause users to receive *fewer* tokens (but never more).

The current issue is that in the `Withdraw` instruction, the amount can be specified as either notes or tokens; and specifying a token amount may lead to advantageous rounding.

Patch

Based on the pool operation (deposit, withdraw, borrow, repay) and base unit (tokens or notes), a `RoundingDirection` is constructed to enforce defensive rounding. In all cases, value amounts are rounded away from the user thereby preventing these incremental rounding attacks.

This rounding system has been deployed across the whole margin-pool program.

OS-JET-ADV-03 [Critical]: Unrestricted MarginSwap CPI

Description

The margin-swap program does not perform sufficient verification on the provided accounts.

margin-swap/src/lib.rs

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

...

    pub swap_program: UncheckedAccount<'info>, // FIXME: need program
id ?
}
```

An attacker can supply a malicious swap_program argument to the MarginSwap instruction to obtain a cross-program invocation to an attacker-controlled program along with a *signed* margin_account.

With a signed margin_account, an attacker can act as the authority for the associated token accounts and issue further instructions to the SPL Token program.

For instance, an attacker could use this control to burn loan notes for a user, effectively erasing debt and enabling unrestricted borrowing.

A normal CPI flow for an invocation of the swap program looks like:

1. jet_control::AdapterInvoke
    a. jet_margin_swap::MarginSwap
        i. jet_margin_pool::Withdraw
        ii. spl_token_swap::Swap
        iii. jet_margin_pool::Deposit

To exploit this vulnerability, an attacker can replace the spl_token_swap program with an attacker-controlled program to issue instructions on behalf of the proxied margin_account:

1. jet_control::AdapterInvoke
    a. jet_margin_swap::MarginSwap
        i. jet_margin_pool::Withdraw
        ii. <attacker_program>
            - spl_token::Burn
        iii. jet_margin_pool::Deposit

Proof of Concept

1. User A deposits $10,000,000 USDC
2. User B deposits $40,000 worth of TSOL.
3. Repeat 250 times:
    a. User B borrows $40,000 worth of USDC
    b. User B exploits MarginSwap to erase $40,000 worth of loan notes. Note that because the loan notes are erased, the program lets user B borrow again.
4. User B withdraws $10,000,000 USDC

An example of an attacker-controlled program that conforms to `spl_token_swap::Swap` and invokes `spl_token::Burn` is provided below:

```rust
entrypoint!(entry);
pub fn entry(
    program_id: &Pubkey,
    accounts: &[AccountInfo],
    instruction_data: &[u8],
) -> ProgramResult {
    let acc_iter = &mut accounts.iter();

    let _swap_pool: &AccountInfo = next_account_info(acc_iter)?;
    let burn_to: &AccountInfo = next_account_info(acc_iter)?;
    let margin_account: &AccountInfo = next_account_info(acc_iter)?;
    let _transit_source: &AccountInfo = next_account_info(acc_iter)?;
    let _vault_into: &AccountInfo = next_account_info(acc_iter)?;
    let _vault_from: &AccountInfo = next_account_info(acc_iter)?;
    let _transit_destination: &AccountInfo =
next_account_info(acc_iter)?;
    let token_mint: &AccountInfo = next_account_info(acc_iter)?;
    let _fee_account: &AccountInfo = next_account_info(acc_iter)?;
    let token_program: &AccountInfo = next_account_info(acc_iter)?;
    token::burn(
        CpiContext::new(
            token_program.clone(),
            Burn {
                mint: token_mint.clone(),
                to: burn_to.clone(),
                authority: margin_account.clone(),
            },
        ),
        40000000000,
    )?;

    Ok(())
}
```

Remediation
Ensure that *trusted adapter programs* are not capable of invoking arbitrary programs. Any program that gets passed the MarginAccount signature must be fully trusted. This includes all adapter programs and subsequent CPI calls.

In this case, verification of the program can be done with Anchor's address constraint:

```
#[derive(Accounts)]
pub struct SwapInfo<'info> {
    ...
    #[account(address = spl_token_swap::ID)]
    pub swap_program: Program<'info>,
}
```

Patch
The margin-swap program now enforces a runtime check on the swap_program argument. The program is limited to `spl_token_swap_v2`, `orca_swap_v1` or `orca_swap_v2`. This check prevents attackers from gaining direct access to a signed MarginAccount.

Additionally, the adapter program system has been extremely hardened. In normal usage of the margin programs, instructions can be proxied through the core margin program to adapter programs (margin-pool, margin-swap, …) via the `AdapterInvoke` instruction. In these cases, it is expected that the Margin program is the direct parent. Crucially, since account balance checks occur in the core Margin program, the security of adapter programs relies on the fact that the return values will be checked and enforced by the Margin program. For example, the ability to invoke margin-pool instructions directly would allow an attacker to violate collateral requirements.

In order to enforce strict cross-program interaction rules. An `invocation` field has been added to the `MarginAccount` state which acts as a sentinel for adapter programs. The Margin program sets this field, marking its call height before it proxies an action. Adapter programs can check this field to ensure that they are the direct child of the margin core program (and are not a target of a man-in-the-middle attack). Importantly, since this field is part of the MarginAccount which is owned by the Margin program, other programs cannot forge this data even if the MarginAccount is passed as a signer.

OS-JET-ADV-04 [Low]: Invalid Position Closing

Description

A new token account is created when a user opens a position in their margin account. This token account is used to store the balance of the position.

When the balance of the position is zero, a user should be able to close the position and the token account created for the position. Margin program defines a `ClosePosition` instruction to allow a user to close the position when it's no longer required and its balance is zero.

Incorrect implementation of the `ClosePosition` instruction handler makes it impossible to close the position and token account. A user becomes unable to reclaim the funds used for rent exemption of the position token_account.

margin/src/instructions/close_position.rs

```
pub fn close_position_handler(ctx: Context<ClosePosition>) -> Result<()> {
   let mut account = ctx.accounts.margin_account.load_mut()?;
   account.unregister_position(&ctx.accounts.token_account.key())?;

   token::close_account(
       ctx.accounts
           .close_token_account_ctx()
           .with_signer(&[&account.signer_seeds()]),
   )?;

   Ok(())
}
```

This is because when the account gets registered, the token mint gets passed in as the key [0].

```
pub fn register_position_handler(ctx: Context<RegisterPosition>) ->
Result<()> {
...
    account.register_position(
        position_token.key(), // [0]
        position_token.decimals,
        address,
        metadata.adapter_program,
        kind,
        metadata.collateral_weight,
        metadata.collateral_max_staleness,
    )?;
```

`MarginAccount::unregister_position` expects the mint address of the position's token as a parameter. Here, instead of mint, the address of token_account is passed. As there won't be a position with token_account address in the margin account, the call to this instruction will fail.

Proof of Concept

1. Register a position
2. Attempt to close that position

Remediation

Use `&ctx.accounts.token_account.mint` instead of `&ctx.accounts.token_account.key()`.

Patch

The correct pubkey is now being passed in.

06 | General Findings

Here we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they do represent antipatterns and could introduce a vulnerability in the future.

ID	Description
OS-JET-SUG-00	Unnecessary accounting invoke signature
OS-JET-SUG-01	Avoid use of UncheckedAccount when not required.
OS-JET-SUG-02	Remove InitAccount from margin core program.
OS-JET-SUG-03	Use LiquidatorAdapterMetadata instead of MarginAdapterMetadata for liquidator invoke instruction.
OS-JET-SUG-04	Margin account doesn't need to be writable for calls to adapter programs.
OS-JET-SUG-05	Consider requiring authentication for the RegisterToken instruction in the control program.
OS-JET-SUG-06	Stronger CPI proxy guarantees

OS-JET-SUG-00: Remove Accounting MarginAccount Signature

Description

As part of our analysis of the Margin program's overall attack surface, we noted how the accounting invoke handler assigns excess permissions by signing the `MarginAccount` through the same `adapter::invoke` handler as `adapter_invoke`.

margin/src/adapter.rs

```
    let mut accounts = vec![AccountMeta {
        pubkey: ctx.margin_account.key(),
        is_signer: true,
        is_writable: true,
    }];
```

This is an unprivileged instruction. Anybody can perform an accounting update in order to refresh prices for the relevant MarginAccount.

In order to prevent arbitrary modifications of balance information, the accounting handler then checks to ensure that `NewBalanceChange` was not returned by the adapter, which would indicate a balance change.

margin/src/instructions/accounting_invoke.rs

```
    match result {
        AdapterResult::NewBalanceChange(_) => {
            msg!("New balance changes may only be realized through
either adapter_invoke or liquidate_invoke, depending on context.");
            err!(ErrorCode::UnauthorizedInvocation)
        }
        AdapterResult::PriceChange(_) => Ok(()),
        AdapterResult::PriorBalanceChange(_) => Ok(()),
    }
```

From a security design perspective, it becomes critical to return `NewBalanceChange` whenever there is modification of any state associated with the `MarginAccount`.

This could be a potentially difficult security boundary to enforce. It might not be immediately clear what state is associated with the `MarginAccount`. An easy fix for this would be to remove the signature from the passed in `MarginAccount` for accounting updates.

This works because an unsigned `MarginAccount` adapter invocation would be equivalent to just invoking the adapter programs directly via CPI. Price updates should not require privileged access to any particular `MarginAccount`.

Remediation

Similar to OS-JET-SUG-04, consider decreasing the permissions on the passed `MarginAccount`.

For example, having a parameter which denotes if the invocation represents an accounting invoke.

margin/src/adapter.rs

```
    let mut accounts = vec![AccountMeta {
        pubkey: ctx.margin_account.key(),
        is_signer: !is_accounting_invoke,
        is_writable: false,
    }];
```

Patch

The Margin adapter program system has been extremely hardened to prevent these types of attacks. See detailed notes in the Patch section for OS-JET-SUG-04.

OS-JET-SUG-01: Stricter Account Validation

Description

Use of `UncheckedAccount` in Anchor instruction declarations can lead to type confusion vulnerabilities as no checks are performed on the provided accounts.

For example, in margin-pool Deposit:

margin-pool/src/instructions/deposit.rs

```
#[derive(Accounts)]
pub struct Deposit<'info> {
    ...
    /// The source of the tokens to be deposited
    #[account(mut)]
    pub source: UncheckedAccount<'info>,

    /// The destination of the deposit notes
    #[account(mut)]
    pub destination: UncheckedAccount<'info>,
    ...
}
```

Both source and destination will be implicitly verified during the CPI calls to `spl::Transfer` and `spl::MintTo` so this is not a security vulnerability.

Remediation

Clarity about the account could be improved by explicitly specifying the `TokenAccount` type.

```
#[derive(Accounts)]
pub struct Deposit<'info> {
    ...
    /// The source of the tokens to be deposited
    #[account(mut)]
    pub source: Account<'info, TokenAccount>,

    /// The destination of the deposit notes
    #[account(mut)]
    pub destination: Account<'info, TokenAccount>,
    ...
}
```

In general, use of typed-Accounts rather than `UncheckedAccount` can help mitigate potential type confusion vulnerabilities ahead of time.

Patch

Account checks have been improved across the suite of margin programs.

OS-JET-SUG-02: Remove InitAccount

Description

The margin core program currently has two ways to create a `MarginAccount`: `InitAccount` and `CreateAccount` which use slightly different techniques. Currently `CreateAccount` is used internally by the Jet simulation framework. For consistency, consider removing `InitAccount` to avoid code duplication and/or confusion.

Remediation

Remove the use of `InitAccount`.

Patch

The `InitAccount` instruction has been removed.

OS-JET-SUG-03: Use LiquidatorAdapterMetadata

Description

`liquidator_invoke` of Margin Program invokes an adapter program with the given instruction data and accounts. The call is signed by `margin_account`. it is checked that the adapter program is present in one of the MarginAdapterProgram metadata accounts.

margin/src/instructions/liquidator_invoke.rs

```
#[derive(Accounts)]
pub struct LiquidatorInvoke<'info> {
    ...
    /// The metadata about the proxy program
    #[account(has_one = adapter_program)]
    pub adapter_metadata: Account<'info, MarginAdapterMetadata>,
}
```

However, the metadata program defines an account to store adapter programs which can be invoked by a liquidator.

metadata/src/lib.rs

```
/// An account that references a program that's allowed to be invoked
by
/// proxy via a margin account for liquidation purposes.
#[account]
#[derive(Default)]
pub struct LiquidatorAdapterMetadata {
   /// The address of the allowed program
   pub adapter_program: Pubkey,
}
```

It makes more sense to use LiquidatorAdapterMetadata to impose additional constraints on the types of operations a liquidator can perform.

Remediation

As a first step, ensure the `liquidator_invoke` instruction handler verifies the adapter program with a `LiquidatorAdapterMetadata` instead of `MarginAdapterMetadata`.

It might also make sense to further constrain the operations a liquidator can perform on an instruction level, for example, by using a similar per-instruction signature scheme as mentioned in OS-JET-ADV-00.

Patch

`LiquidatorAdapterMetadata` has been removed and both invoke variants use `MarginAdapterMetadata` for consistency.

OS-JET-SUG-04: Unnecessary Writable MarginAccount

Description

All calls to the adapter program from the Margin program are signed by the margin account on which the adapter program operates on. The margin account is also set as writable in the call which is unnecessary.

margin/src/adapter.rs

```
pub fn invoke(
    ctx: &InvokeAdapter,
    account_metas: Vec<CompactAccountMeta>,
    data: Vec<u8>,
) -> Result<AdapterResult> {
    let mut accounts = vec![AccountMeta {
        pubkey: ctx.margin_account.key(),
        is_signer: true,
        is_writable: true,
    }];
    ...
}
```

The `margin_account` is owned by the margin program. Thus, only the margin program can write to the margin account.

As long as the adapter program is not the margin program then it doesn't matter if the margin account is writable or not as non self-recursion reentrancy is prohibited in Solana.

In general, the adapter programs should not need to directly modify the data in the margin account. Thus, it does not make sense to pass it in as writable.

Remediation

Similar to OS-JET-SUG-00, refrain from setting `is_writable` for adapter invocations.

Patch

The AdapterInvoke process has been considerably reworked and security-hardened. This suggestion no longer applies.

OS-JET-SUG-05: Unprivileged RegisterToken

Description

Anyone can invoke the `RegisterToken` instruction of the Jet Control program to create a Margin pool for the given token. Allowing anyone to create a valid margin pool for a token is a vulnerability. A malicious actor can take advantage of it to attack the protocol.

For example, an attacker could mint any number of tokens and use them as collateral to withdraw other valuable tokens from other Margin pools.

At the moment this is not a security issue because margin pool creation is not complete until the `ConfigureToken` instruction is called. Note that the `ConfigureToken` instruction requires the root authority as signer of the instruction which ensures that only the root authority can create a valid margin pool.

However, it seems unnecessary to make this an unprivileged instruction.

Remediation

Allow only the root authority to register a token for margin pool. For example, by changing:

control/src/instructions/register_token.rs

```
pub struct RegisterToken<'info> {
    #[account(mut)]
    requester: Signer<'info>,
    ...
}
```

to

```
pub struct RegisterToken<'info> {
    #[cfg_attr(not(feature = "devnet"), account(address =
crate::ROOT_AUTHORITY))]
    pub requester: Signer<'info>,
    ...
}
```

Patch

The `RegisterToken` instruction has been removed and margin-pool creation and configuration is now restricted to the root authority.

OS-JET-SUG-06: Stronger CPI Proxy Requirements

Description

A large portion of the security of margin-pool stems from the fact that several instructions, namely `Borrow`, `Repay`, `MarginRefreshPosition`, and `Withdraw` are "proxy-only" instructions. Balance and account health checks happen in the *parent program* by observing potential balance changes set with `jet_margin::write_adapter_result`.

This is critical for security, as instructions that lead to unhealthy accounts fail in later checks and are aborted.

margin/src/instructions/adapter_invoke.rs

```
    match result {
        AdapterResult::NewBalanceChange(_) =>
margin_account.verify_healthy_positions()?,
        AdapterResult::PriceChange(_) => (),
        AdapterResult::PriorBalanceChange(_) => (),
    }
```

An attacker who could invoke adapter instructions directly without using the standard proxy could simply ignore the adapter result and bypass account health checks.

Currently, the only requirement for invoking these instructions is that they must be invoked with a signed `MarginAccount` (which only the margin program can provide). However, this same MarginAccount is the authority for many additional accounts.

In OS-JET-ADV-03, we demonstrate how an attacker can leverage a flawed adapter program to invoke custom, attacker-controlled instructions. Specifically, the lack of account verification on the `swap_program` enables the attacker to invoke any subsequent instruction with a signed `MarginAccount`.

In theory, an attacker could leverage this control to invoke any of these trusted instructions. However, current CPI depth limits of 4 prevented this specific vector due to the fact that these margin-pool instructions attempt to invoke SPL Token instructions internally.

Remediation

Consider requiring additional proof in proxy-only instructions such that adapter programs can be sure that they are being invoked directly by margin core and as such, guarantee that the account health will be enforced.

One option is to use signed PDA's provided by the margin core program. For example, MarginBorrow could be refactored as:

```
#[derive(Accounts)]
pub struct MarginBorrow<'info> {
    /// The margin account being executed on
    #[account(signer)]
    pub margin_account: AccountLoader<'info, MarginAccount>,

    #[account(signer,
              seeds = [
                  margin_account.key().as_ref(),
                  b"AdapterInvoke".as_ref()
              ],
              bump,
              seeds::program = jet_margin::ID)]
    pub control_proof: AccountInfo<'info>,
    ...
}
```

Only the margin core program can sign for this PDA.

The adapter programs can then omit the signed `control_proof`, while providing the signed `margin_account` to other CPI calls. This helps prevent the invocation of arbitrary adapter programs.

Patch

The Margin adapter program system has been extremely hardened to prevent these types of attacks. See detailed notes in the Patch section for OS-JET-SUG-04.

07 | Appendix

Appendix A: Program Files

Below are the files in scope for this audit and their truncated sha256 hashes.

```
control
  Cargo.toml                            1829c93f8b04a5269415b916ac46e310
  Xargo.toml                            815f2dfb6197712a703a8e1f75b03c69
  src
    instructions.rs                     fe343a9d9033a3407fc35dd2ec405f25
    lib.rs                              99cd5156bcb70c40f72513ed15cc8cb1
    instructions
      configure_token.rs                da1554b99a83b9b556ab952b9c39dee2
      create_authority.rs               166f96cd536d883a6b9c9405190edee6
      register_adapter.rs               d76876284f5ac05f99354913da8162af
      register_token.rs                 774b19fd34082d1323abc78efac2de41
drift-labs-pyth
  Cargo.toml                            24dba07122d4cba479eee8604c1559c8
  Xargo.toml                            a4292e0c9687ede7dc40c108682afb35
  src
    lib.rs                              0d30af75de01093df99f051b7baf50f3
    pc.rs                               f03fd33f3c55230fcb32247cbd430158
margin
  Cargo.toml                            16a414123cc7dbba5ff82256bb0129c0
  Xargo.toml                            815f2dfb6197712a703a8e1f75b03c69
  src
    adapter.rs                          8f561b22714589207160523bc5d20c39
    instructions.rs                     962f452bf66a9093f358484c325f0a1d
    lib.rs                              38082f804b7e9a9ebbeb788938eb7bac
    state.rs                            36eb9c3301c29440ed477e6b62e72ec5
    instructions
      accounting_invoke.rs              429584eefec6314676234addcbbe1768
      adapter_invoke.rs                 ecb69e4b76f17ae7b15261a480503333
      close_account.rs                  7b6f6cae0f6f87be2f874de065d69d66
      close_position.rs                 5ab789e6383e0584dd326da6c00483e0
      create_account.rs                 9ccb438f440e0eab6a92d46f5115ca4f
      init_account.rs                   00fbd61edabee17a0c2652bf7a77c71a
      liquidate_begin.rs                ee45b4820e3cff43f1916cc2512b6776
      liquidate_end.rs                  695e4afd675f864ddb8f522fb270264d
      liquidator_invoke.rs              dbce29029b1c638c1bcebde03b8523d9
      register_position.rs              fc67b115879b13fd36eb51ef8df7e0c5
      update_position_balance.rs        a223ad75891283945a4f4766c92ea693
      verify_healthy.rs                 e6595e86f35be74cb58b6baa5facc703
margin-pool
  Cargo.toml                            d745e254bd10cd14fee8a8608c1c4b37
  Xargo.toml                            815f2dfb6197712a703a8e1f75b03c69
  src
    instructions.rs                 6f4b7c6c02191be383a7e3870e86fe01
    lib.rs                          cf5ac383b9bb44d0c1a8fdfd5a8bc747
    state.rs                        23f0ed27512a4fe247a7ad54adc06b3f
    util.rs                         a50437360dd0e54c92ef3843a231988b
    instructions
      collect.rs                    8027a11bc2ad3cb8de1d961abf533906
      configure.rs                  08ef48d9434b705f555073e6ff0b7dd9
      create_pool.rs                f542df51b234a032ad22323851779176
      deposit.rs                    f6edf2f05ddcea32160adb6f5b613358
      margin_borrow.rs              e64b6fb5c6efbeace18fd1165ce803e2
      margin_refresh_position.rs    c34df4287882a7911c913411ed3ecb32
      margin_repay.rs               15e7739ce1ee5354fed6ed872657fcf1
      margin_withdraw.rs            80b084b1751e90fd645d2c181d067b1b
      withdraw.rs                   bc66655dd10a686b4cbb2494900af48d
margin-serum
  Cargo.toml                        1ad8da6a88d5fd4db722292f9372995c
  Xargo.toml                        815f2dfb6197712a703a8e1f75b03c69
  src
    instructions.rs                 c64d6e473790faabf7fdb27289891431
    lib.rs                          5bdf0159a6291186cc7d3edda90432a0
    state.rs                        e3b0c44298fc1c149afbf4c8996fb924
    util.rs                         e3b0c44298fc1c149afbf4c8996fb924
    instructions
      cancel_order_by_client_id_v2.rs   f330b5f080722457ca7cce620edafff3
      cancel_order_v2.rs            d0b0c06e58fc324efe086a76098f40dd
      close_open_orders.rs          42867ac65e7423aaefd3619a345a07f0
      consume_events.rs             32fd908342814a64e5648dd903fee5dd
      match_orders.rs               644b3fb1c020d045644df6cf0096159b
      new_order_v3.rs               3c6c950438953c36bbaebe3f4c5305ee
      settle_funds.rs               e35f388bd7f4fe88d622174b2d3409d2
margin-swap
  Cargo.toml                        fe4e07e9b4b1ce1855f42954378bc047
  Xargo.toml                        815f2dfb6197712a703a8e1f75b03c69
  src
    lib.rs                          d18b2e79464cbb5b83587025cfd45246
metadata
  Cargo.toml                        7e0cf5fa4b3086ab987a12562a5eac09
  Xargo.toml                        815f2dfb6197712a703a8e1f75b03c69
  src
    lib.rs                          1c1bba7adcef13cc72daed5dc09f72b7
```

Appendix B: Implementation Security Checklist

Unsafe arithmetic

Integer underflows or overflows	Unconstrained input sizes could lead to integer over or underflows, causing potentially unexpected behavior. Ensure that for unchecked arithmetic, all integers are properly bounded.
Rounding	Rounding should always be done against the user to avoid potentially exploitable off-by-one vulnerabilities.
Conversions	Rust `as` conversions can cause truncation if the source value does not fit into the destination type. While this is not undefined behavior, such truncation could still lead to unexpected behavior by the program.

Account security

Account Ownership	Account ownership should be properly checked to avoid type confusion attacks. For Anchor, the safety of unchecked accounts should be clearly justified and immediately obvious.
Accounts	For non-Anchor programs, the type of the account should be explicitly validated to avoid type confusion attacks.
Signer Checks	Privileged operations should ensure that the operation is signed by the correct accounts.
PDA Seeds	PDA seeds are uniquely chosen to differentiate between different object classes, avoiding collision.

Input Validation

Timestamps	Timestamp inputs should be properly validated against the current clock time. Timestamps which are meant to be in the future should be explicitly validated so.
Numbers	Sane limits should be put on numerical input data to mitigate the risk of unexpected over and underflows. Input data should be constrained to the smallest size type possible, and upcasted for unchecked arithmetic.
Strings	Strings should have sane size restrictions to prevent denial of service conditions

Internal State	If there is internal state, ensure that there is explicit validation on the input account's state before engaging in any state transitions. For example, only open accounts should be eligible for closing.

Miscellaneous

Libraries	Out of date libraries should not include any publicly disclosed vulnerabilities
Clippy	`cargo clippy` is an effective linter to detect potential anti-practices.

Appendix C: Proof of Concepts

Below are the provided proof of concept files and their truncated sha256 hashes.

```
README.md                                74116f9c517f769d3e592c3abb424b56
run.sh                                   1c893c756b44271daee874505e71edd0
os-jet-adv-00
  patch                                  ca6b6c2f56ec4b4f696e34ab208dd759
  run.sh                                 c88ed3f909ae3d47f05489451e18dda9
os-jet-adv-01
  patch                                  bc1745795271ba098b8ebfd6ce45f8f2
  run.sh                                 c88ed3f909ae3d47f05489451e18dda9
os-jet-adv-02
  patch                                  76509f0aaedcaf8300e549f563822a2c
  run.sh                                 3bba5a972732d6a6a041bfc015a1d8ac
os-jet-adv-03
  patch                                  6d7d400fd22ec1466a680d4887a88d42
  run.sh                                 20d207ffc1f3ca3871123267b574d20a
os-jet-adv-04
  patch                                  819ddb2e141db2922db8adde6a5ffb5e
  run.sh                                 c88ed3f909ae3d47f05489451e18dda9
```

Appendix D: Vulnerability Rating Scale

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

Critical	Vulnerabilities which **immediately** lead to loss of user funds with minimal preconditions Examples: - Misconfigured authority/token account validation - Rounding errors on token transfers
High	Vulnerabilities which **could** lead to 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 which could lead to denial of service scenarios or degraded usability. Examples: - Malicious input cause computation limit exhaustion - Forced exceptions preventing normal use
Low	Low probability vulnerabilities which could still be 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 - Uncaught Rust errors (vector out of bounds indexing)