Jet v2 Fixed Term

Audit

Presented by:

OtterSec contact@osec.io

Nicola Vella nick0ve@osec.io

Harrison Green hgarrereyn@osec.io

Contents

01 | Executive Summary

Overview

Jet Protocol engaged OtterSec to perform an assessment of the `fixed-term` program. This assessment was conducted between April 24th and May 2nd, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 9 findings total.

Specifically, we discovered a critical issue regarding the missing validation of term deposit markets, which may allow an attacker to exploit value differences between markets (OS-JFT-ADV-00). We also identified an issue with improper airspace validation, which may disrupt market operations and lead to rent-Lamport theft (OS-JFT-ADV-01).

Additionally, we provided recommendations around the enforcement of margin users' airspace validation during market interactions (OS-JFT-SUG-00), the prevention of possible type confusion during AAOB slab initialization (OS-JFT-SUG-01), and the necessity of validating term loans against the market during repayment procedures (OS-JFT-SUG-02).

Overall, we commend the Jet Protocol team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/jet-lab/jet-v2/tree/master/programs/fixed-term. This audit was performed against commit cdabc31.

`fixed-term` is a lending and borrowing protocol on Solana featuring an innovative product known as Jet Fixed Term, which offers fixed-term, fixed-rate lending and borrowing.

Users may interact with the protocol, lending their assets for fixed terms and earning interest or borrowing assets using an order book system for ticket issuance and exchange. All positions are over-collateralized, ensuring risk management through the protocol's automated liquidation process.

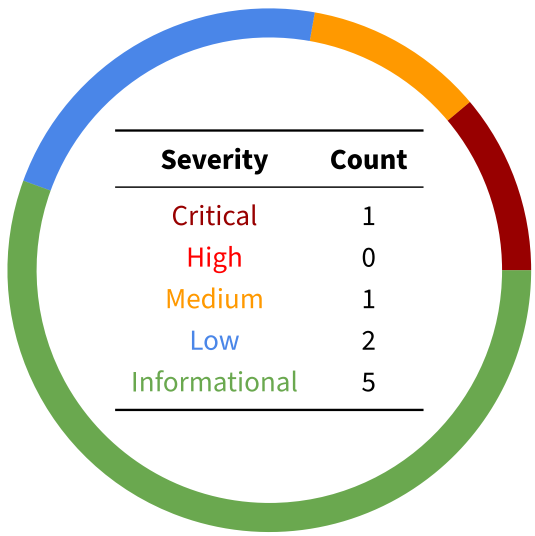
Its key features include:

1. Fixed-term, fixed-rate lending and borrowing through Jet Fixed Term.
2. A unique ticket system functions as perpetual, zero-strike American call options.
3. A rolling mechanism for extending obligations every tenor interval.
4. Fungibility of tickets for a particular principal token and tenor, facilitating their trade in a single order book.

03 | Findings

Overall, we reported 9 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 help mitigate future vulnerabilities.



Severity	Count
Critical	1
High	0
Medium	1
Low	2
Informational	5

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 A.

ID	Severity	Status	Description
OS-JFT-ADV-00	Critical	Resolved	An attacker may redeem deposits produced for one market in a different market.
OS-JFT-ADV-01	Medium	Resolved	The validation of airspace in the `AuthorizeCrank` and `RevokeCrank` instructions perform incorrectly.
OS-JFT-ADV-02	Low	Resolved	Processing an event on the orderbook may require initializing a `TermDeposit`.
OS-JFT-ADV-03	Low	Resolved	`InitializeMarginUser` does not correctly validate that `underlying_collateral_mint` belongs to the provided market.

OS-JFT-ADV-00 [crit] | Missing TermDeposit Market Validation

Description

`RedeemDeposit` does not validate that `TermDeposit`'s, creator is the Market. The creator of `TermDeposit` is relevant due to `TermDeposit` tracking the funds to redeem and the `Market` owning the funds for the withdrawal. The lack of validation is present in the following snippet of code:

```rust
pub struct RedeemDeposit<'info> {
    /// The tracking account for the deposit
    #[account(mut,
              close = payer,
              has_one = owner @ FixedTermErrorCode::WrongDepositOwner,
              has_one = payer
    )]
    pub deposit: Account<'info, TermDeposit>,
...
    pub market: AccountLoader<'info, Market>,
```

An attacker may exploit the lack of validation by creating a `TermDeposit` in a `Market` responsible for a cheaper asset. Redeeming the cheaper asset in a `Market` responsible for a more expensive asset nets the difference in the value of the two assets.

Proof of Concept

Suppose there are two markets: `marketA` trading $SOL and `marketB` trading $USDC. Notice that $SOL mint decimals are equal to nine, while $USDC mint decimals are equal to six. So, one $SOL is equivalent to 10^9 tickets while one $USDC is equivalent to 10^6 tickets.

1. An attacker exchanges one $SOL for 10^9 `ticketsA` on `marketA` by invoking `ExchangeTokens` instruction.
2. The attacker stakes their 10^9 `ticketsA` in `marketA`, receiving back a `TermDeposit` for the same amount by invoking `StakeTickets` instruction.
3. The attacker waits for `marketA.lend_tenor` and then redeems the `TermDeposit` on `marketB` by invoking `RedeemDeposit` instruction, netting $10^9/10^6$ $USDC.

Remediation

Validate for `TermDeposit.market` through anchor means.

DIFF

```diff
pub struct RedeemDeposit<'info> {
    /// The tracking account for the deposit
    #[account(mut,
              close = payer,
+             has_one = market,
              has_one = owner @ FixedTermErrorCode::WrongDepositOwner,
              has_one = payer
    )]
    pub deposit: Account<'info, TermDeposit>,
...
```

Patch

Resolved in 5f8559a by implementing the check described above.

OS-JFT-ADV-01 [med] | Improper Airspace Validation

Description

The validation of airspace in the `AuthorizeCrank` and `RevokeCrank` instructions is inadequate, enabling any airspace authority to craft `CrankAuthorizations` for any market and revoke them. This may allow an attacker to:

1. Craft `CrankAuthorizations` for any market.
2. Revoke `CrankAuthorizations` for any market.

Although, the ability to craft a `CrankAuthorization` for any market is not useful since the `ConsumeEvents` instruction checks the market and airspace. The following code snippet shows this:

src/orderbook/instructions/consume_events/accounts.rs RUST

```rust
#[derive(Accounts)]
#[derive(Accounts, MarketTokenManager)]
pub struct ConsumeEvents<'info> {
    ...
    #[account(
        has_one = crank @ FixedTermErrorCode::WrongCrankAuthority,
        constraint = crank_authorization.airspace ==
    ↪   market.load()?.airspace @
    ↪   FixedTermErrorCode::WrongAirspaceAuthorization,
        constraint = crank_authorization.market == market.key() @
    ↪   FixedTermErrorCode::WrongCrankAuthority,
    )]
    pub crank_authorization: Account<'info, CrankAuthorization>,
    ...
```

The ability to revoke them may cause significant harm, as revoking all of the `CrankAuthorizations` may halt the market's functionality and result in the theft of rent-Lamports.

Proof of Concept

1. An attacker with authority over `airspace1` invokes `fixed_term::RevokeCrank` over all of the authorized cranks in `airspace2`, stealing the rent.
2. Markets under `airspace2` cease to be operational since the events cannot be consumed by anyone through `ConsumeEvents`.

Remediation

Properly check that the `Airspace` account passed to `AuthorizeCrank` and `RevokeCrank` is of the same market under modification.

src/control/instructions/authorize_crank.rs DIFF

```diff
pub struct AuthorizeCrank<'info> {
+   #[account(has_one = airspace @ FixedTermErrorCode::WrongAirspace)]
    pub market: AccountLoader<'info, Market>,
    ...
}
```

src/control/instructions/revoke_crank.rs DIFF

```diff
#[derive(Accounts)]
pub struct RevokeCrank<'info> {
-   #[account(mut, close = receiver)]
+   #[account(mut, close = receiver, has_one = airspace @
     ↪  FixedTermErrorCode::WrongAirspace)]
    pub metadata_account: Account<'info, CrankAuthorization>,
    ...
}
```

Patch

The lack of checks in `AuthorizedCrank` was fixed in 5f8559a. Similarly, the lack of checks in `RevokeCrank` was fixed in 8f32ed8.

OS-JFT-ADV-02 [low] | Orderbook Event Queue Denial Of Service

Description

The account address of a TermDeposit is a PDA derived from a crank-supplied seed, where the seed does not matter for signer fills. Therefore, cranks are free to choose seeds that do not collide. However, processing the event is impossible if the account already exists. Users may also register these accounts outside of event processing through StakeTickets.

In exceptional scenarios, an attacker may continuously front-run the event resolution for an account they control to temporarily freeze the orderbook.

Proof of Concept

1. An attacker invokes StakeTickets repeatedly, registering TermDeposits with seeds that they know will be used by the cranks.
2. When the orderbook attempts to process an event and initialize a TermDeposit using one of these seeds, it fails because the account already exists.
3. The orderbook is effectively frozen until this issue resolves.

Remediation

Ensure that the TermDeposit seeds for StakeTickets and event processing exist in disjoint seed namespaces to prevent attackers from blocking event processing. For instance, consider implementing the following seed namespaces:

1. ["term_deposit", <market>, <ticket_holder>, "stake", <seed>]
2. ["term_deposit", <market>, <ticket_holder>, "fill", <seed>]

This would ensure that the TermDeposits created by StakeTickets and during event processing do not collide.

Patch

Resolved in 5f8559a by changing the PDA derivation for user's tickets produced through StakeTickets.

OS-JFT-ADV-03 [low] | Missing Validation Of Underlying Collateral Mint

Description

`InitializeMarginUser` is responsible for setting up a user's account for margin trading. As part of this process, `InitializeMarginUser` verifies that the provided mints for claims and ticket collateral belong to the specified market.

However, the function does not perform this check for `underlying_collateral_mint`. The following code snippet demonstrates the current implementation:

```rust
pub struct InitializeMarginUser<'info> {
    ...
    #[account(
        has_one = claims_mint @ FixedTermErrorCode::WrongClaimMint,
        has_one = ticket_collateral_mint @
    ↪   FixedTermErrorCode::WrongCollateralMint
    )]
    pub market: AccountLoader<'info, Market>,
    pub claims_mint: Box<Account<'info, Mint>>,
    pub ticket_collateral_mint: Box<Account<'info, Mint>>,
    pub underlying_collateral_mint: Box<Account<'info, Mint>>,
    ...
```

Exploitation on this lack of validation may lead to initiating margin trades with incorrect underlying collateral, resulting in incorrect account balances.

Proof of Concept

1. An attacker invokes `InitializeMarginUser`, providing an `underlying_collateral_mint` that does not belong to the specified market.
2. The function proceeds without error, initializing the user's account for margin trading with the incorrect `underlying_collateral_mint`.

Remediation

Add a validation check to `InitializeMarginUser` to ensure that the provided `underlying_collateral_mint` belongs to the specified market. The fix may execute similarly to the existing checks for `claims_mint` and `ticket_collateral_mint`.

DIFF

```diff
pub struct InitializeMarginUser<'info> {
    ...
    #[account(
        has_one = claims_mint @ FixedTermErrorCode::WrongClaimMint,
        has_one = ticket_collateral_mint @
     ↪  FixedTermErrorCode::WrongCollateralMint,
+       has_one = underlying_collateral_mint
    )]
    pub market: AccountLoader<'info, Market>,
    ...
```

Patch

Resolved in 626cdb2 by implementing the proposed check.

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 lead to security issues in the future.

ID	Description
OS-JFT-SUG-00	When creating a `MarginUser` in a `Market`, the `Airspace` of the corresponding `MarginAccount` is not being explicitly checked to ensure that it matches the `Airspace` of the `Market`.
OS-JFT-SUG-01	It may be possible to pass an already-initialized anchor account as the bids or asks account when initializing the agnostic order book through `InitializeOrderbook`.
OS-JFT-SUG-02	Verify that the `TermLoan` being repaid belongs to the provided `Market`.
OS-JFT-SUG-03	The `Settle` instruction does not validate that the `margin_account` corresponds to the associated `margin_user`.
OS-JFT-SUG-04	Use `require_keys_eq` instead of `require_eq` when comparing `Pubkey` values to enhance error messages.

OS-JFT-SUG-00 | Enforce Market Airspace Validation For Margin Users

Description

For a `MarginAccount` to interact with a `Market`, it must create a `MarginUser` using the `InitializeMarginUser` instruction. However, this instruction does not explicitly check that the `MarginUser` belongs to the same `Airspace` as the `Market`. Instead, it relies on the `jet-margin` program to check it when attempting to register the position. This is shown below:

```rust
    pub fn register_position(
        &mut self,
        config: PositionConfigUpdate,
        approvals: &[Approver],
    ) -> AnchorResult<AccountPositionKey> {
        ...
        if self.airspace != config.airspace {
            return err!(ErrorCode::WrongAirspace);
        }
        ...
```

Remediation

Implement a check on the `InitializeMarginUser` instruction handler to ensure that the signing `MarginAccount` belongs to the same `Airspace` as the `Market`.

Patch

Resolved in 626cdb2.

OS-JFT-SUG-01 | Possible Type Confusion With AAOB Slab

Description

A check is implemented during the initialization of the agnostic order book state to ensure that the bids and asks accounts passed are uninitialized. However, this check implemented is insecure. The following code snippet shows that the check for equality with `AccountTag::Uninitialized` is performed only on the first byte of the account data.

RUST
```rust
pub fn initialize(asks_data: &mut [u8], bids_data: &mut [u8]) ->
↪ Result<(), ProgramError> {
    if asks_data[0] != AccountTag::Uninitialized as u8
        || bids_data[0] != AccountTag::Uninitialized as u8
    {
        return Err(ProgramError::AccountAlreadyInitialized);
    }
    asks_data[0] = AccountTag::Asks as u8;
    bids_data[0] = AccountTag::Bids as u8;
    Ok(())
}
```

As `AccountTag::Uninitialized` is defined as 0x00, it may be possible to bypass this check by passing any anchor account containing the first byte of the discriminator equal to a null byte.

Currently, this weakness is not exploitable because discriminators for accounts in the fixed term never start with a null byte, as shown below:

RUST
```rust
Market = [219, 190, 213, 55, 0, 227, 198, 154]
CrankAuthorization = [218, 255, 159, 109, 236, 131, 215, 201]
MarginUser = [182, 149, 190, 42, 115, 239, 142, 128]
TermLoan = [72, 47, 66, 203, 56, 242, 85, 202]
EventAdapterMetadata = [195, 178, 194, 30, 225, 98, 200, 170]
TermDeposit = [158, 97, 90, 10, 119, 192, 33, 160]
```

Remediation

Modify `Slab::initialize` to perform the check on all 8 bytes of the discriminator, as already done in `EventQueue::from_buffer` to mitigate this risk.

Patch

Resolved in 626cdb2 by using anchor.

```diff
##[derive(Accounts)]
pub struct InitializeOrderbook<'info> {
    ...
-   #[account(mut)]
+   #[account(zero)]
    pub event_queue: AccountInfo<'info>,
-   #[account(mut)]
+   #[account(zero)]
    pub bids: AccountInfo<'info>,
-   #[account(mut)]
+   #[account(zero)]
    pub asks: AccountInfo<'info>,
    ...
}
```


OS-JFT-SUG-02 | Validate TermLoan Against Market In Repay

Description

In the Repay functionality, TermLoan being repaid is not validated to be associated with the Market. In the current state of the code, this does not pose a security risk because the association is indirectly enforced by the presence of the claims account, which must have the proper claims_mint for the transfer to succeed.

Remediation

Include an explicit verification step within the Repay functionality. This verification should confirm that the TermLoan being repaid indeed belongs to the provided Market. This can be implemented through anchor.

```diff
pub struct Repay<'info> {
    /// The account tracking information related to this particular user
    #[account(mut, has_one = claims @
     ↪  FixedTermErrorCode::WrongClaimAccount)]
    pub margin_user: Account<'info, MarginUser>,
    #[account(
        mut,
+       has_one = market,
        has_one = margin_user @ FixedTermErrorCode::UserNotInMarket,
        has_one = payer,
        constraint = term_loan.sequence_number
            == margin_user.next_term_loan_to_repay().unwrap()
            @ FixedTermErrorCode::TermLoanHasWrongSequenceNumber
    )]
    pub term_loan: Account<'info, TermLoan>,
```

Patch

Resolved in 626cdb2.

OS-JFT-SUG-03 | Validate Margin Account Against Margin User

Description

The `Settle` instruction does not validate that the `margin_account` corresponds to the associated `margin_user`. Although the ATA computation partially addresses this verification, it is crucial to enforce this association to prevent potential unauthorized operations.

Remediation

Explicitly verify the association between the `margin_account` and the corresponding `margin_user` within the `Settle` operation, using anchor.

DIFF

```diff
pub struct Settle<'info> {
    /// The account tracking information related to this particular user
    #[account(mut,
+       has_one = margin_account,
        has_one = market @ FixedTermErrorCode::UserNotInMarket,
        has_one = claims @ FixedTermErrorCode::WrongClaimAccount,
        has_one = ticket_collateral @
    ↪   FixedTermErrorCode::WrongTicketCollateralAccount,
        has_one = underlying_collateral @
    ↪   FixedTermErrorCode::WrongUnderlyingCollateralAccount,
    )]
    pub margin_user: Box<Account<'info, MarginUser>>,

    /// use accounting_invoke
    pub margin_account: AccountLoader<'info, MarginAccount>,
```

Patch

Resolved in 626cdb2.

OS-JFT-SUG-04 | Improve Error Messages

Description

The codebase uses `require_eq` for comparing `Pubkey` values in methods such as:

- `OrderbookMut::cancel_order`
- `QueueIterator::maybe_adaptor`
- `RepayAccounts::repay`

However, `require_eq` does not provide detailed error messages when the comparison fails, which may allow troubleshooting and debugging to be more challenging.

Remediation

Replace the usage of `require_eq` with `require_keys_eq` for comparing `Pubkey` values. This alternative function provides more informative error messages, including the actual values under comparison.

Patch

Resolved in 626cdb2.

A | 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 that immediately lead to 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 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 that could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
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

B | 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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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.