Honey

Audit

Presented by:

OtterSec contact@osec.io

William Wang defund@osec.io

Shiva Genji sh1v@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Honey Finance engaged OtterSec to perform an assessment of the `honey` program. This assessment was conducted between August 30th and September 19th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team over to streamline patches and confirm remediation.

We delivered final confirmation of the patches September 30th, 2022.

Key Findings

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

- 13 findings total
- 3 vulnerabilities which could lead to loss of funds
  - OS-HNY-ADV-00: The `WithdrawNFTSolvent` instruction is not admin-gated, which allows anyone to withdraw arbitrary NFTs.
  - OS-HNY-ADV-01 The `LiquidateSolvent` instruction is not admin-gated, which allows anyone to falsely mark their loan as repaid.
  - OS-HNY-ADV-02: Liquidators that partially repay loans will receive the NFT collateral nonetheless.

02 | Scope

The source code was delivered to us in a git repository at github.com/honey-labs/nftLendBorrow. This audit was performed against commit `1437e33`.

There was a total of 1 program included in this audit. A brief description of the program is as follows. A full list of program files and hashes can be found in Appendix A.

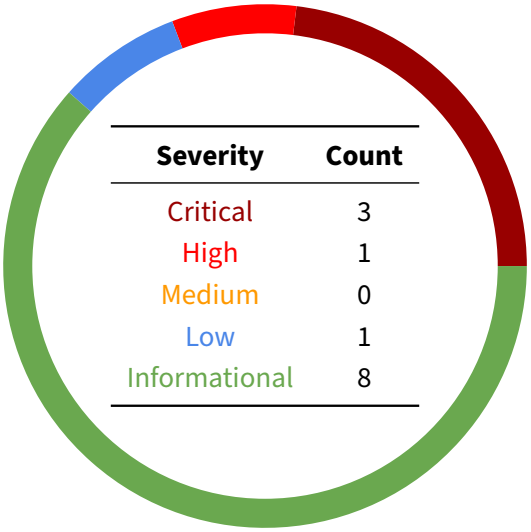
Name	Description
honey	Lending protocol where NFTs are used as collateral.

03 | Findings

Overall, we report 13 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.



Severity	Count
Critical	3
High	1
Medium	0
Low	1
Informational	8

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 D.

ID	Severity	Status	Description
OS-HNY-ADV-00	Critical	Resolved	The `WithdrawNFTSolvent` instruction is not admin-gated, which allows anyone to withdraw arbitrary NFTs.
OS-HNY-ADV-01	Critical	Resolved	The `LiquidateSolvent` instruction is not admin-gated, which allows anyone to falsely mark their loan as repaid.
OS-HNY-ADV-02	Critical	Resolved	Liquidators that partially repay loans will receive the NFT collateral nonetheless.
OS-HNY-ADV-03	High	Resolved	Initializing an NFT account permanently prevents any other obligation from using it collateral.
OS-HNY-ADV-04	Low	Resolved	The program checks that the market currently allows borrows when it should check for withdrawals instead.

OS-HNY-ADV-00 [crit] [resolved] | Solvent NFT Withdrawal Is Not Admin-Gated

Description

In addition to standard liquidations, Honey allows an administrator to “manually” liquidate a user’s obligation through the following process:

1. Forcefully withdraw the user’s NFT collateral.
2. Sell the NFT on Solvent.
3. Deposit the proceeds into Honey’s vault.
4. Forcefully update the user’s loan as repaid.

The first step is performed via the `WithdrawNFTSolvent` instruction. Notice that the instruction’s signer is not verified to be an admin-controlled address. By specifying their own `withdrawer` account, an attacker can withdraw NFTs from any obligation.

src/instructions/withdraw_nft_solvent.rs RUST

```
60  /// The admin who will own the nft
61  pub withdrawer: Signer<'info>,
```

Remediation

Use Anchor’s `address` constraint to match the `withdrawer` account against a hard-coded public key, which should be admin-controlled.

Patch

Fixed in 8407e3f.

OS-HNY-ADV-01 [crit] [resolved] | Solvent Liquidation Is Not Admin-Gated

Description

The fourth step of the process described in OS-HNY-ADV-00 is performed via the `LiquidateSolvent` instruction. Similarly, notice that the instruction's signer is not verified to be an admin-controlled address. By specifying their own `executor` account, an attacker can burn loan notes without transferring tokens.

src/instructions/liquidate_solvent.rs RUST

```rust
64  /// The admin/authority that has permission to execute solvent
        ↪ liquidation
65  #[account(mut)]
66  pub executor: Signer<'info>,
```

Remediation

Use Anchor's `address` constraint to match the `executor` account against a hard-coded public key, which should be admin-controlled.

Patch

Fixed in 53b1491.

OS-HNY-ADV-02 [crit] [resolved] | Partial Liquidation Should Not Be Allowed

Description

In the `ExecuteLiquidateBid` instruction, the liquidator specifies how many loan notes to burn with the `amount` argument. This is subsequently used to calculate how many tokens they must transfer.

src/instructions/execute_liquidate_bid.rs RUST

```
240 let payoff_notes = amount.as_loan_notes(reserve_info, Rounding::Down)?;
241 let payoff_notes = std::cmp::min(
242     payoff_notes,
243     token::accessor::amount(&loan_account.to_account_info())?,
244 );
245 let payoff_tokens = std::cmp::min(
246     reserve_info.loan_notes_to_tokens(payoff_notes, Rounding::Up),
247     reserve.unwrap_outstanding_debt(clock.slot).as_u64(0),
248 );
```

Later, the program transfers the NFT collateral, assuming the loan has been fully repaid. But this is not necessarily true, since liquidators may repay an arbitrarily small amount. In this case, the obligation ends up in an unhealthy state: there are outstanding loans, yet no collateral.

src/instructions/execute_liquidate_bid.rs RUST

```
286 // 7. remove the NFT from the obligation
287 obligation.unregister_nft(accounts.nft_mint.key())?;
288
289 // 8. send the NFT to the bidder's new account
290 token::transfer(
291     accounts
292         .transfer_nft_context()
293         .with_signer(&[&market.authority_seeds()]),
294     1,
295 )?;
```

This is indicative of a broader design issue: since NFT transfer is “atomic,” liquidation must also happen all at once, across multiple reserves. In contrast, `ExecuteLiquidateBid` is only designed to repay a single reserve.

Remediation

Limit obligations to a single loan, and enforce that the loan’s balance is fully repaid during a liquidation.

Patch

Fixed in b1c2127 and 5dbff6a.

OS-HNY-ADV-03 [high] [resolved] | Denial of Service Affecting NFT Deposit

Description

In the `InitializeNFTAccount` instruction, `collateral_account` is created to hold an NFT as collateral. Notice that it is an associated token account, whose address is a PDA derived from the mint and authority. In particular, anyone attempting to initialize an NFT account will be required to use the same address. On the other hand, the `init` constraint will throw an error if the account already exists.

RUST

```rust
/// The account that will store the deposit notes
#[account(init,
          associated_token::mint = deposit_nft_mint,
          associated_token::authority = market_authority,
          payer = owner)]
pub collateral_account: Box<Account<'info, TokenAccount>>,
```

This effectively means that an NFT can only be used as collateral once. An attacker can also cause denial of service attack by intentionally invoking `InitializeNFTAccount` for arbitrary NFTs in the market.

Remediation

Replace the `init` constraint with `init_if_needed`, so that `collateral_account` may be reused.

Patch

Fixed in 9f04e0c.

OS-HNY-ADV-04 [low] [resolved] | Incorrect Market Flag Check

Description

The `WithdrawNFT` and `WithdrawNFTSolvent` instructions check that the market currently allows borrows. However, it is performing an NFT withdrawal, not a borrow.

src/instructions/withdraw_nft_solvent.rs RUST

```
162  market.verify_ability_borrow()?;
```

Remediation

Use the `Market::verify_ability_deposit_withdraw` method instead.

Patch

Fixed in 3f8e965 and 786409d.

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 do represent antipatterns and could introduce a vulnerability in the future.

ID	Status	Description
OS-HNY-SUG-00	Resolved	The `InitializeReserve` instruction has an unused account.
OS-HNY-SUG-01	Resolved	The checks enforced while registering a new position are ineffective.
OS-HNY-SUG-02	Resolved	The checks enforced while registering an NFT as collateral are ineffective.
OS-HNY-SUG-03	Resolved	Iterating over market reserves may theoretically omit some.
OS-HNY-SUG-04	Resolved	Revoking a bid should always transfer the escrow account's full balance.
OS-HNY-SUG-05	Resolved	The `Deposit` and `Withdraw` instructions are unnecessary.
OS-HNY-SUG-06	Resolved	Metadata validation logic should be refactored into a helper function.
OS-HNY-SUG-07	Resolved	Liquidators should not be able to repay loans for healthy obligations.

OS-HNY-SUG-00 [resolved] | Unused Quote Token Mint

Description

The `InitializeReserve` does not do anything meaningful with the `quote_token_mint` account.

Remediation

All references to `quote_token_mint` should be removed from the `InitializeReserve` instruction.

Patch

Fixed in 4cff36a.

OS-HNY-SUG-01 [resolved] | Ineffective Checks in Position Registration

Description

When registering a new position on an obligation side, its data is validated against the existing positions. In particular, the token account and reserve cannot already be in use. Notice that the program assigns to the first empty slot it encounters, then returns immediately. If there are later positions in the array (this can occur if a position is unregistered), the checks are ineffective.

src/state/obligation.rs RUST

```rust
fn register(&mut self, new: Position) -> Result<()> {
    for position in self.positions.iter_mut() {
        if position.account == new.account.key() {
            panic!(...);
        }

        if position.reserve_index == new.reserve_index &&
    ↪   position.account != Pubkey::default()
        {
           panic!(...);
        }

        if position.account != Pubkey::default() {
            continue;
        }

        *position = new;

        return Ok(());
    }

    err!(ErrorCode::NoFreeObligation)
}
```

Remediation

Check over the entire `positions` array before assigning the new position.

Patch

Fixed in e41f114.

OS-HNY-SUG-02 [resolved] | Ineffective Checks in Collateral Registration

Description

Honey is designed so that each obligation holds at most one NFT as collateral. However, the explicit check is incorrect; the program should verify that the collateral array is empty before registering a new position.

src/state/obligation.rs RUST

```rust
// 1 nft per market per wallet
if self.collateral().iter().count() > 1 {
    return err!(ErrorCode::CollateralExists);
}
```

Remediation

Check that the collateral array is empty, instead of having at most one element.

Patch

Implicitly fixed by 9f04e0c, which stops using the collateral array altogether.

OS-HNY-SUG-03 [resolved] | Incorrect Market Reserves Iteration

Description

The `MarketReserves::iter` and `MarketReserves::iter_mut` methods use `take_while` to ignore empty entries in the array. However, this will stop at the first empty entry and ignore the remaining entries, even if they contain reserves. This scenario can occur if a reserve is unregistered.

src/state/market.rs RUST

```rust
pub fn iter(&self) -> impl Iterator<Item = &ReserveInfo> {
    self.reserve_info
        .iter()
        .take_while(|r| r.reserve != Pubkey::default())
}

pub fn iter_mut(&mut self) -> impl Iterator<Item = &mut ReserveInfo> {
    self.reserve_info
        .iter_mut()
        .take_while(|r| r.reserve != Pubkey::default())
}
```

Remediation

Use the `filter` method to ignore uninitialized entries.

Patch

Fixed in 1345a9a.

OS-HNY-SUG-04 [resolved] | Transfer Full Balance While Revoking Bid

Description

In the `RevokeLiquidateBid` instruction, the bidder specifies how many tokens to withdraw with the `withdraw_amount` argument. It transfers them from the `bid_escrow` token account, which is subsequently closed. Notice that the argument is unnecessary, since the bidder should always transfer the full balance. Moreover, token accounts cannot be closed while holding a non-zero balance.

Remediation

Remove the `withdraw_amount` argument, and always transfers the full balance of `bid_escrow`.

Patch

Fixed in 1570470.

OS-HNY-SUG-05 [resolved] | Unnecessary Deposit and Withdraw Instructions

Description

Honey is a fork of Jet v1, which supports both the `Deposit` and `DepositTokens` instructions for backward compatibility. In reality, only one instruction is necessary; the same applies for the `Withdraw` and `WithdrawTokens` instructions.

Remediation

Remove the `Deposit` and `Withdraw` instructions. Update the `DepositTokens` and `WithdrawTokens` instructions to use PDA seeds for the `deposit_note_account` account.

Patch

Fixed in 52339a3.

OS-HNY-SUG-06 [resolved] | Refactor Metadata Validation

Description

Honey uses Metaplex to verify that NFTs belong to a whitelisted collection. This involves complex validation logic, which is repeated in numerous instructions:

- `InitializeNFTAccount`
- `DepositNFT`
- `WithdrawNFT`
- `WithdrawNFTSolvent`

Remediation

Refactor the `validate` method and `is_valid_metadata` computation from the instructions into helper functions.

Patch

Fixed in e41f114.

OS-HNY-SUG-07 [resolved] | Healthy Obligations Can Be Liquidated

Description

The `ExecuteLiquidateBid` should only be invoked on obligations which are deemed unhealthy. However, this check is commented out on the locked commit. As it stands, this represents a vulnerability which could lead to loss of funds.

src/instructions/execute_liquidate_bid.rs RUST

```
229 // 1. Verify the obligation is unhealthy
230 // if obligation.is_healthy(market_reserves, clock.slot) {
231 //     return Err(ErrorCode::ObligationHealthy.into());
232 // }
```

Remediation

Uncomment the check.

Patch

Fixed in e29eaf5.

A | Program Files

Below are the files in scope for this audit and their corresponding SHA256 hashes.

```
Cargo.toml                                  27e20cd53bc0f435528e6108df8cd299fb7bf042dee9438186394077e1abd4f6
Xargo.toml                                  815f2dfb6197712a703a8e1f75b03c6991721e9eb7c40dfaec8b0b49da4aa629
src
  common.rs                                 dc854d643d637a71a657b7697b47405eed276f8555d3fc79b9b0d9e0e2d38c67
  errors.rs                                 1895fcc00bb3d8345dabd7ddb84500f8b2110a0f39ed90fd320144a86226e9aa
  lib.rs                                    df81bb4155e08cbb51321a59d8f314feb72bc9a101d48635186a7ae218d38236
  utils.rs                                  bb4eecac42660eef5f70d6199d0377f100e697067d97781a8fac81375e5469b6
  instructions
    borrow.rs                               0fd508a839af0579041befbca1c70c5a1010ef1ee6f5f16687014f4609d3d981
    deposit.rs                              e69e920903443f6d00487bf191610924257cf423bb4db00277518341d572e006
    deposit_nft.rs                          8f218cc52c425f44a600457f8a1a583beada017528046d91518c36214bb40a49
    deposit_tokens.rs                       3cbfbd38e0403efb6918c390c2488aadf6331e82b9695dea56f6fd2739d53fd1
    execute_liquidate_bid.rs                591995dafa96e5dca3dc7b88a89312858611789af83c72b1e51a0d01643c886d
    increase_liquidate_bid.rs               3806b2575111f4fb366919a55f51292ec9303fa3a07e502eef1efd87169d49fe
    init_deposit_account.rs                 455638cd984991226802fc248b451043f78d09319744255dd7d6bda5d525f6ca
    init_loan_account.rs                    f883e0c5ed2e56e72f6aa58f137666d6842d92b25c6355b618a59a492fcb641e
    init_market.rs                          57596b963d5a98d4ed66faeff66ab80a2e711c48726c8a830af833b9a5dadac0
    init_nft_account.rs                     77c29f7d12e75059295eb990a6f99cb9ee09a01a3492c85b033893ea1a6fa5c0
    init_obligation.rs                      8de056205b695b32d89ec970294e23ed0868b879f67e6633cd22ab4f19a677d6
    init_reserve.rs                         03162855c768606f47205e9422e55299ac377c1f0e3bbf376f94caab2b1f1172
    liquidate_solvent.rs                    2a5f05babc21891ee0a52a3e66ad8fd21049b710afedcd24cadab50d63956e98
    mod.rs                                  2ebc21eab06afa3ae2bf0efd2d22032e0f0553ecb853244fd1bb0cb96998e430
    place_liquidate_bid.rs                  fce82566d0b355893314e60bcc5d9223e3812baaed49776c593d650e9a5e8f35
    refresh_reserve.rs                      ad1fe95bda09dfbd807d7d344386a71c6287e0738a4ea75d5f3752b35a89bb8f
    repay.rs                                8b1ecf2d395463cbf88ca84f62436f5369c11ac3311da6790cf7787e10f52a3e
    revoke_liquidate_bid.rs                 e02bce0c17168af12c90d83a6494664e928387a90c33d8fb763facbe69c5b0d5
    set_market_flags.rs                     6bbbe3f046383cdc4cefc22d26bb488a00c07c30826cdb39c5a77d4111d5db2b
    set_market_owner.rs                     ca3cbee5bab5c572047a2966585dba0b802287bcd1eab8a2e8518c84851718e8
    update_reserve_config.rs                26ef26e97f09cccf1e11f1496d6613a73927eea9a4f93178463dfc7c741e6cbf
    withdraw.rs                             1ec78cd14b28d6a6a0b4973c93f34b889c538b5b19dbaf226fc6254167404189
    withdraw_nft.rs                         725734cd0e5eedaaf509e47ac4197669ac4fa078e55beb907759329c6a29b6c5
    withdraw_nft_solvent.rs                 295cd72d97aaab4fd34920461b879254a7ec63041b1b56c799ade4033fe7b882
    withdraw_tokens.rs                      b165114ed1647a69ecb645b15ff12530b885cb19a8da3daf8fb95f5fd3969c7f
  state
    bid.rs                                  d84ac9e93e30913cc68fa55f78d897413fe358e8dd8077253d9847c49c79ab83
    cache.rs                                4b6b3cc8f67c58535f748e5078894522101416 23f7ca23dd99ebdc207a26a6b9
    market.rs                               c9f17ad53cd212b333d259837b16fe3ad9f6954e172829c7114d09679cf48e7c
    mod.rs                                  8671afce4dcff3c0eb9408432d3b238e59d5b4e5367f63913eeb21b47c5bd78d
    obligation.rs                           9eedd8045ed6c76cd5410113bc077e2fcd66e1cf06a1ee7fc8ac17127da65ca5
    reserve.rs                              80505278bdbc903e85224b4c22e61cabac262775dc1fd181effbe0e42a85f0f4
```

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 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 C.

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 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.

C | 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-patterns.

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)