hexens x ZHARTA

Security Review Report for Zharta

December 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This report covers the security review for Zharta, a new peer-to-peer lending protocol for ERC20 tokens. The protocol is highly permissionless and allows for very configurable loans between a lender and a borrower.

Our security assessment was a full review of the code, spanning a total of 2 weeks.

During our review, we did identified 3 High severity vulnerabilities, which could have resulted in a loss of user assets.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed or acknowledged by the development team and consequently validated by us.

We can confidently say that the overall security and code quality have increased after completion of our audit.

3. Security Review Details

- **Review Led by**

Jahyun Koo, Lead Security Researcher

- **Scope**

The analyzed resources are located on:

🔗 https://github.com/Zharta/lending-erc20-protocol/tree/cf0e9985213630adae7477f047d171b6b9545868

The issues described in this report were fixed in the following commits:

🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/bc954d5c866cc4592ece43452a7fe25e4a5e95b6
🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/dc9380324dc5320488aa22dd9025fb16a4b5bf15
🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/efc4f44b81a3f48295ad01c46549be0ec9726c3c
🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/68b0836ce27ed8a949c7730bad0793e84f2e7598
🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/5bc783c48a5150f9ac040e0d6342fd8f5801dcaf
🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/5decf48a79f94f4e7fffd4445a589a89e85e5161
🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/0f710ede23658e611bced9b665469eee56707068
🔗 - https://github.com/Zharta/lending-erc20-protocol/commit/bbd6a42d5fa1a4da21ebe2030a019ed7da390ead

▪ Changelog

8 December 2025	Audit start
23 December 2025	Initial report
7 January 2026	Revision received
9 January 2026	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

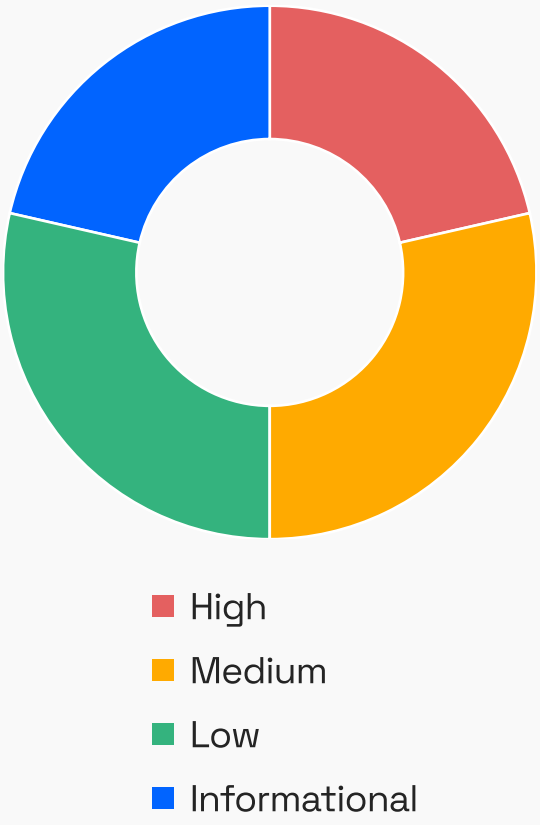
- **Issue Symbolic Codes**

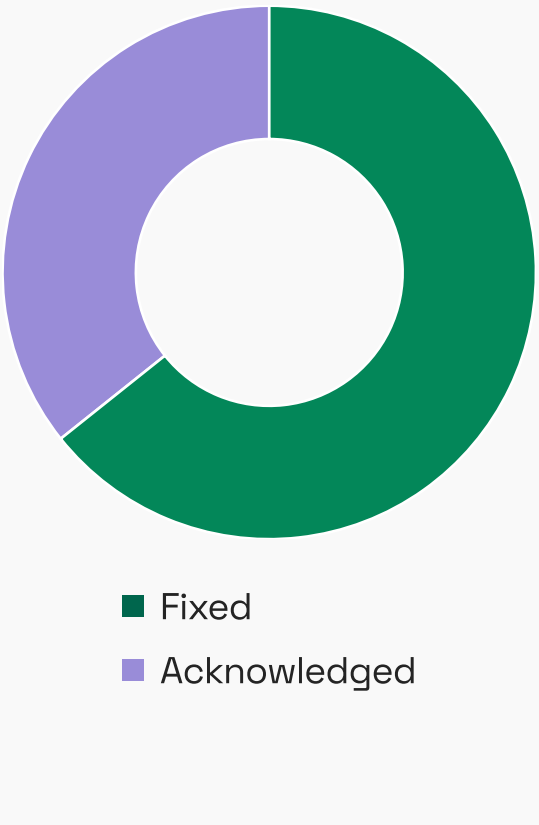
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	0
High	3
Medium	4
Low	4
Informational	3
Total:	**14**





6. Weaknesses

This section contains the list of discovered weaknesses.

ZHAR2-6 | Lender can worsen a borrower's loan without giving additional funds

Fixed

Severity:	Probability:	Impact:
High	Unlikely	Critical

Path:

contracts/v2/P2PLendingV2Refinance.vy

Description:

The **replace_loan_lender** function allows a lender to replace a borrower's loan, either with a new lender or with themselves under new terms. The function is intended to prevent the borrower's repayment obligations, principal, interest, call eligibility, and LTV thresholds, from being worse under the new loan up to the original maturity.

checks include:

```
if offer.offer.liquidation_ltv > 0:
    assert offer.offer.liquidation_ltv > max_initial_ltv, "liquidation ltv le initial ltv"
    # required for soft liquidation: (1 + f) * iltv < 1
    assert (BPS + base.partial_liquidation_fee) * max_initial_ltv < BPS * BPS, "initial ltv too high"
```

```
if new_loan.liquidation_ltv > 0:
    assert new_loan.liquidation_ltv >= loan.liquidation_ltv, "liquidation ltv lt old loan"
```

```
if new_loan.liquidation_ltv > 0:
    assert initial_ltv <= current_ltv, "initial ltv gt old loan"
```

However, if the new loan's **liquidation_ltv** is set to 0, soft liquidation is disabled. In this case, the lender can manipulate the loan by:

1. Increasing the total loan value (TLV) to a higher amount.
2. Setting a high **origination_fee_amount** so that **borrower_compensation** and **borrower_delta** are zero.

This allows the lender to increase the borrower's debt without sending extra funds.

Example Scenario:

Old Loan:
Old loan 0.3 ETH
Old collateral 1 ETH
TLV 30% (super safe)
initial_ltv (when it gets liquidated): 70%
ORG_fee = 3%

New Loan:
new loan 0.9 ETH
old collateral 1 ETH
TLV 90%
initial_ltv 90%
ORG_fee_amount = 0.6 ETH (66.666% bps)

We calculate how much the borrower receives, the old lender receives, and the new lender has to send:

```
borrower_compensation: uint256 = convert(max(convert(max_interest_delta, int256), convert(outstanding_debt + new_loan.origination_fee_amount, int256) - convert(new_principal, int256)), uint256)
borrower_delta: int256 = convert(new_principal + borrower_compensation, int256) - convert(outstanding_debt + new_loan.origination_fee_amount, int256) # SHOULD BE 0
old_lender_delta: int256 = convert(outstanding_debt - protocol_settlement_fee_amount, int256) - convert(borrower_compensation, int256) # THIS SHOULD BE HIGHER
new_lender_delta: int256 = convert(new_loan.origination_fee_amount, int256) - convert(new_loan.amount + new_loan.protocol_upfront_fee_amount, int256)
```

borrower_compensation = 0.3 + 0.6 - 0.9 = **0**
borrower_delta = 0.9 + 0 - 0.3 + 0.6 = **0**
old_lender_delta = 0.3 - 0 = **0.3**
new_lender_delta = 0.6 - 0.9 = **-0.3**

This means that the borrower gets 0
lender has to send 0.3
lender received 0.3
meaning the lender didn't send anything because he send it to himself.

Impact:

- Borrower receives no additional funds, despite their debt increasing from 0.3 ETH to 0.9 ETH.
- Lender does not provide extra funds but shifts the loan to worse terms.
- Borrower's risk increases: they now have a larger debt against the same collateral, making

the loan much riskier.

- Collateral retrieval becomes much more expensive:
  - If the borrower wants to reclaim their collateral, they must pay 0.9 ETH instead of the original 0.3 ETH.
  - If the borrower is liquidated, they lose the entire collateral, because the loan is now over-leveraged.

Remediation:

Enforce borrower-protection invariants regardless of whether soft liquidation is enabled, including a hard requirement that the new loan's effective initial LTV and repayment burden do not increase. Additionally, constrain origination fees and principal changes in lender-initiated refinances so any debt increase must be matched by actual funds delivered to the borrower, and prevent using **liquidation_ltv = 0** to bypass these safeguards.

ZHAR2-9 | Vault withdrawal accounting breaks for void-return ERC20 tokens, leading to collateral withdrawal DoS

Acknowledged

Severity:	Probability:	Impact:
High	Likely	High

Description:

P2PLendingV2Base.vy and P2PLendingV2Vault.withdraw performs a low-level ERC20 **transfer** and validates success by converting the returned data to a boolean.

```
@external
def withdraw(amount: uint256, wallet: address):
    """
    @notice Withdraw tokens from the vault to a specified wallet.
    @dev Transfers tokens from the vault to the wallet and emits a Withdraw event.
    @param amount The amount of tokens to withdraw.
    @param wallet The address of the wallet to which tokens will be transferred.
    """
    assert msg.sender == self.caller, "unauthorized"
    assert amount + self.pending_transfers_total <= staticcall IERC20(self.token).balanceOf(self), "insufficient balance"

    success: bool = False
    response: Bytes[32] = b""

    success, response = raw_call(
        self.token,
        abi_encode(wallet, amount, method_id=method_id("transfer(address,uint256)")),
        max_outsize=32,
        revert_on_failure=False
    )

    if not success or not convert(response, bool):
        log TransferFailed(wallet=wallet, amount=amount)
        self.pending_transfers[wallet] += amount
        self.pending_transfers_total += amount
    else:
        log Withdraw(wallet=wallet, amount=amount)
```

For tokens that execute transfers successfully but return no data (void-return behavior such as USDT), the call can succeed while the vault treats it as a failure. In this case, the vault increments **pending_transfers** and **pending_transfers_total** even though the recipient already received the tokens and the vault balance has decreased. This desynchronizes internal accounting from the actual token balance. Since **withdraw** also enforces **amount + pending_transfers_total <= balanceOf(vault)**, the inflated **pending_transfers_total** can block subsequent withdrawals, preventing collateral from being returned to users and interfering with core flows that rely on vault withdrawals (e.g., settlement and liquidation).

Additionally, **withdraw_pending** relies on a strict ERC20 boolean return, which can make pending balances unclaimable for the same class of tokens.

```
@external
def withdraw_pending(amount: uint256):
    """
    @notice Withdraw tokens from the vault that are pending transfer to the sender.
    @dev Transfers tokens from the vault to the sender and emits a WithdrawPending event.
    @param amount The amount of tokens to withdraw.
    """
    assert self.pending_transfers[msg.sender] >= amount, "insufficient pending collateral"
    self.pending_transfers[msg.sender] -= amount
    self.pending_transfers_total -= amount
    assert extcall IERC20(self.token).transfer(msg.sender, amount), "transfer failed"
    log WithdrawPending(wallet=msg.sender, amount=amount)
```

Remediation:

Normalize ERC20 transfer handling in the vault to support both standard boolean-return tokens and void-return tokens. Treat a transfer as successful when the call succeeds and either returns no data or returns a truthy value, and apply the same success criteria consistently across **withdraw**, **deposit**, and **withdraw_pending**. Ensure pending transfer balances are only recorded when a transfer actually fails and that pending withdrawals remain claimable under the same token-compatibility rules.

Commentary from the client:

"Non standard contracts like USDT are currently not supported in the protocol (controlled by Zharta at deployment time). Also, no loans would be possible because calls to transferFrom used to receive funds would revert (RETURNDATASIZE < expected)."

ZHAR2-7 | Lender can Bypass Call Window Protection and liquidate borrowers

Fixed

Severity:	Probability:	Impact:
High	Unlikely	Critical

Description:

By chaining **replace_loan_lender** calls and temporarily disabling **call_eligibility**, a lender can set an extremely short **call_eligibility** & **call_window** (e.g., 1 block), enabling immediate liquidation of a borrower's collateral within 2 blocks.

The **replace_loan_lender** function contains a check to prevent the new loan to have a longer **call_window** (the time a borrower has to repay if the liquidator calls early repayment) than the previous loan:

```
if loan.call_eligibility > 0 and new_loan.call_eligibility > 0:
    assert new_loan.call_window >= loan.call_window, "call window lt old loan"
```

This is intended to protect the the borrower by ensuring that the lender cannot reduce the call window to a value shorter than the original loan.

However this check can be bypassed by calling replace loan twice:

In the first call the lender can set **call_eligibility** of the new loan to 0, this disables the check for the second call because the AND operator:

```
if loan.call_eligibility > 0 and new_loan.call_eligibility > 0:
    assert new_loan.call_window >= loan.call_window, "call window lt old loan"
```

Since **new_loan_call_eligibilty == 0** the condition is false, because the soft liquidation is disabled there is no need to check if the next call window has a value.

But the issue arises with the second call to **replace_loan_lender**:

```
if loan.call_eligibility > 0 and new_loan.call_eligibility > 0:
    assert new_loan.call_window >= loan.call_window, "call window lt old loan"
```

Because this check returns false again because the previous **call_eligibility** was 0, the caller can set any **call_window**.

Also because of a second check its needed the repay time is the same as previous one:

```
def _get_repayment_time(loan: base.Loan) -> uint256:
    if loan.call_eligibility == 0:
        return loan.maturity
    elif loan.call_time > 0:
        return min(loan.maturity,loan.call_time + loan.call_window)
    else:
        return min(loan.maturity, max(block.timestamp, loan.start_time + loan.call_eligibility) + loan.call_window)
```

```
    repayment_time_old_loan: uint256 = self._get_repayment_time(loan)
    repayment_time_new_loan: uint256 = self._get_repayment_time(new_loan)
    assert repayment_time_new_loan >= repayment_time_old_loan, "repayment time lt old loan" #edge case when already called.
```

This means that the older loan has to be longer than the previous one, but this can still be passed by passing time to +/- call_eligibility.

Example:

Original Loan:

- block.timestamp = 0
- duration = 180 seconds
- maturity = 0 + 180 = 180 seconds
- call_eligibility = 100 seconds
- call_window = 10 seconds

Original loan repayment time = 110 seconds

Foward 108 Seconds:

Loan v1:

- block.timestamp = 108 seconds
- duration = 2 seconds
- maturity = 108 + 10 = 118 seconds
- call_eligibility = 0 (disabled, no soft liquidation)
- call_window = 1 second

Loan v1 repayment time = 110 seconds (same repayment time as original loan)

Loan v2:

Now, update to Loan v2.

- block.timestamp = 116 seconds
- duration = 2 seconds
- maturity = 108 + 10 = 118 seconds
- call_eligibility = 1 (enabled, soft liquidation allowed)
- call_window = 1 second

Loan v2 repayment time = 110 seconds

Impact:

- Original Loan (180 seconds total):
  - Early repayment can be called after 100 seconds.
  - The lender has 10 seconds to act after early repayment is called.
- Malicious Loan (180 seconds total):
  - Early repayment can be called after 1 second. (we are already on the 108th second)
  - The lender only has 1 second to act after early repayment is called.

In this scenario, the borrower would not have enough time to react and would lose their collateral.

Remediation:

Consider not allowing soft liquidation in replace loan if the previous loan didn't allow it.

ZHAR2-1 | Missing committed_liquidity reduction could lead to loan DoS

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	High

Description:

In the **liquidate_loan** function, there is no code that decreases **commited_liquidity** when calling **_reduce_commited_liquidity**.

```
assert commited_liquidity + amount <= offer.offer.available_liquidity, "offer fully utilized"
```

If multiple liquidations occur, **commited_liquidity** may accumulate, and this can cause future loan executions to fail due to the validation inside the **_check_and_update_offer_state** function.

```
@external
def liquidate_loan(
    loan: base.Loan,

    payment_token: address,
    collateral_token: address,
    oracle_addr: address,
    oracle_reverse: bool,
    kyc_validator_addr: address,
    collateral_token_decimals: uint256,
    payment_token_decimals: uint256,
    offer_sig_domain_separator: bytes32,
    vault_impl_addr: address,
):

    """
    @notice Fully liquidates a defaulted loan. Can be called by anyone.
    @param loan The loan to be soft liquidated.
    """

    assert base._is_loan_valid(loan), "invalid loan"
    # assert base._is_loan_defaulted(loan), "loan not defaulted"
    liquidator: address = msg.sender if not base.authorized_proxies[msg.sender] else tx.origin
```

```
    if not base._is_loan_defaulted(loan):

        current_interest: uint256 = base._compute_settlement_interest(loan)
        convertion_rate: base.UInt256Rational = base._get_oracle_rate(oracle_addr, oracle_reverse)
        current_ltv: uint256 = base._compute_ltv(loan.collateral_amount, loan.amount + current_interest,
convertion_rate, payment_token_decimals, collateral_token_decimals)

        assert loan.liquidation_ltv > 0, "not defaulted, partial disabled"
        assert current_ltv >= loan.liquidation_ltv, "not defaulted, ltv lt partial"

        principal_written_off: uint256 = 0
        collateral_claimed: uint256 = 0
        liquidation_fee: uint256 = 0
        principal_written_off, collateral_claimed, liquidation_fee = base._compute_partial_liquidation(
            loan.collateral_amount,
            loan.amount + current_interest,
            loan.initial_ltv,
            loan.partial_liquidation_fee,
            convertion_rate,
            payment_token_decimals,
            collateral_token_decimals
        )

        assert principal_written_off >= loan.amount + current_interest, "not defaulted, partial possible"

    current_interest: uint256 = base._compute_settlement_interest(loan)
    outstanding_debt: uint256 = loan.amount + current_interest
    rate: base.UInt256Rational = base._get_oracle_rate(oracle_addr, oracle_reverse)

    liquidation_fee_collateral: uint256 = min(loan.collateral_amount, outstanding_debt * base.full_liquidation_fee
* rate.denominator * collateral_token_decimals // (BPS * rate.numerator * payment_token_decimals))
    collateral_for_debt: uint256 = outstanding_debt * rate.denominator * collateral_token_decimals //
(rate.numerator * payment_token_decimals)
    remaining_collateral: uint256 = loan.collateral_amount - liquidation_fee_collateral
    remaining_collateral_value: uint256 = remaining_collateral * rate.numerator * payment_token_decimals //
(rate.denominator * collateral_token_decimals)
    protocol_settlement_fee_amount: uint256 = min(loan.protocol_settlement_fee * current_interest // BPS,
remaining_collateral_value)
    shortfall: uint256 = outstanding_debt - remaining_collateral_value if remaining_collateral_value <
outstanding_debt else 0
    _vault: vault.Vault = base._get_vault(loan.borrower, vault_impl_addr)

    if remaining_collateral_value >= outstanding_debt:
        base._receive_funds(liquidator, outstanding_debt, payment_token)
```

```
        base._send_collateral(liquidator, collateral_for_debt + liquidation_fee_collateral, _vault)
        if remaining_collateral > collateral_for_debt:
            base._send_collateral(loan.borrower, remaining_collateral - collateral_for_debt, _vault)
        base._send_funds(loan.lender, outstanding_debt - protocol_settlement_fee_amount, payment_token)
        base._send_funds(base.protocol_wallet, protocol_settlement_fee_amount, payment_token)

    else:
        base._receive_funds(liquidator, remaining_collateral_value, payment_token)
        base._send_collateral(liquidator, loan.collateral_amount, _vault)
        base._send_funds(loan.lender, remaining_collateral_value - protocol_settlement_fee_amount,
payment_token)
        base._send_funds(base.protocol_wallet, protocol_settlement_fee_amount, payment_token)

    base.loans[loan.id] = empty(bytes32)

    log main.LoanLiquidated(
        id=loan.id,
        borrower=loan.borrower,
        lender=loan.lender,
        liquidator=liquidator,
        outstanding_debt=outstanding_debt,
        collateral_for_debt=collateral_for_debt,
        remaining_collateral=remaining_collateral,
        remaining_collateral_value=remaining_collateral_value,
        shortfall=shortfall,
        liquidation_fee=liquidation_fee_collateral,
        protocol_settlement_fee_amount=protocol_settlement_fee_amount
    )
```

Remediation:

Consider adding this code at end of **liquidate_loan** function.

```
base._reduce_commited_liquidity(loan.lender, loan.offer_tracing_id, loan.amount)
```

ZHAR2-16 | Missing committed_liquidity reduction in partially_liquidate_loan could lead to loan DoS

Acknowledged

Severity:	Probability:	Impact:
Medium	Unlikely	High

Description:

commited_liquidity Not Reduced When Lender Self-Liquidates Can Block New Loans

In **partially_liquidate_loan**, **commited_liquidity** is reduced only when the liquidator is not the lender

```
assert commited_liquidity + amount <= offer.offer.available_liquidity, "offer fully utilized"
```

Because **commited_liquidity** is not decreased in the lender-self-liquidation path, the system's internal accounting can become inflated relative to the actual available liquidity. Over time, this can cause the validation above to fail, incorrectly reverting with "**offer fully utilized**" and preventing further loans from being created.

```
@external
def partially_liquidate_loan(
    loan: base.Loan,

    payment_token: address,
    collateral_token: address,
    oracle_addr: address,
    oracle_reverse: bool,
    kyc_validator_addr: address,
    collateral_token_decimals: uint256,
    payment_token_decimals: uint256,
    offer_sig_domain_separator: bytes32,
    vault_impl_addr: address,
):
 ...
if liquidator != loan.lender:
  base._transfer_funds(liquidator, loan.lender, principal_written_off, payment_token)
  base._reduce_commited_liquidity(loan.lender, loan.offer_tracing_id, principal_written_off)
```

Remediation:

Consider moving **base._reduce_commited_liquidity** outside of the if statement.

Commentary from the client:

"The commited liquidity can not be reduced when the liquidator is the lender, as no liquidity is transferred back to the lender."

ZHAR2-2 | transfer_loan always reverts due to swapped vault addresses

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/v2/P2PLendingV2Erc20.vy
contracts/v2/P2PLendingV2Base.vy

Description:

P2PLendingV2Erc20.transfer_loan, the vault arguments are reversed when moving collateral. The function calls **_send_collateral** with the new borrower's vault as the source and the old borrower's vault address as the destination, so the withdrawal always reverts because the new vault is empty. As a result, authorized transfer agents cannot move loans for recovery or legal transfer scenarios. Funds are not misdirected, but the transfer feature is effectively DOSed.

```
def transfer_loan(loan: base.Loan, new_borrower: address, new_borrower_kyc: base.SignedWalletValidation):
    ...
    updated_loan.id = base._compute_loan_id(updated_loan)
    base.loans[updated_loan.id] = base._loan_state_hash(updated_loan)
    base.loans[loan.id] = empty(bytes32)

    base._send_collateral(
        base._wallet_to_vault(loan.borrower, vault_impl_addr),
        loan.collateral_amount,
        base._create_vault_if_needed(new_borrower, vault_impl_addr, collateral_token)
    )
    ...
```

```
def _send_collateral(wallet: address, _amount: uint256, _vault: vault.Vault):
    extcall _vault.withdraw(_amount, wallet)
```

Remediation:

Update **transfer_loan** to withdraw collateral from the old borrower's vault and send it to the new borrower's vault. Ensure the vault used as the source is the old borrower's vault and the destination is the newly created (or existing) vault for the new borrower.

ZHAR2-18 | Reentrancy in claimInterest Enables Uncollateralized Loans

Fixed

Severity:	Probability:	Impact:
Medium	Rare	Critical

Description:

The **claimInterest** function performs an external call to the user-supplied **interest_payment** address:

```
extcall ProfitInterestPayment(interest_payment).claimInterest(amount)
```

Because this external call happens before the function finishes its critical accounting checks and token transfers, a malicious **interest_payment** contract can re-enter the system during the callback and execute state-changing logic (e.g., creating a loan). This enables a reentrancy attack that can be leveraged to obtain an effectively uncollateralized loan.

In particular, the function relies on this invariant:

```
assert IERC20(payment_token).balanceOf(self) - balance_before == amount
```

However, during reentrancy the attacker can manipulate the vault's token balance (e.g., by depositing collateral into the vault through another code path), making the balance delta appear valid even though the increase did not come from legitimate interest.
This scenario may occur if the P2PLendingV2VaultProfitr implementation is used in the loan logic and a valid KYC signature can be generated for a specific contract address, or a forwarder is registered in authorized_proxies by an administrator.

Attack scenario

1. The attacker (who is also **borrower** and **vault.owner**) calls: **vault.claimInterest(malicious, WETH, amount)**.
2. **balance_before** records the vault's WETH balance.
3. **claimInterest** calls **malicious.claimInterest(amount)**, entering the attacker-controlled callback.
4. Inside the callback, the attacker calls **P2P.create_loan()** → which triggers **_receive_collateral** → causing collateral to be deposited into the vault.
5. The callback returns. Now balance_after - balance_before == amount passes because the collateral deposit increased the vault's WETH balance, not because real interest was paid.
6. claimInterest executes transfer(self.owner, amount), sending amount (funded by the collateral) directly to the vault owner (the attacker).

7. Result: the attacker ends up holding both the loan principal and the collateral, achieving an uncollateralized loan.

```
@external
def claimInterest(interest_payment: address, payment_token: address, amount: uint256):
    balance_before: uint256 = staticcall IERC20(payment_token).balanceOf(self)
    extcall ProfitInterestPayment(interest_payment).claimInterest(amount)
    assert (staticcall IERC20(payment_token).balanceOf(self)) - balance_before == amount, "incorrect interest amount"
    assert extcall IERC20(payment_token).transfer(self.owner, amount), "transfer failed"
```

Remediation:

Consider adding a reentrancy modifier to the Vault's claimInterest, deposit, and withdraw functions.

ZHAR2-3 | Missing validation for oracle responses in _get_oracle_rate

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/v2/P2PLendingV2Base.vy

Description:

The **_get_oracle_rate** function in **P2PLendingV2Base.vy** retrieves price data from an oracle via **latestRoundData()** but does not validate the returned values. The function directly uses the **answer** field without checking whether it is positive, non-zero, or stale.

The **latestRoundData()** call returns multiple fields including **answer**, **updatedAt**, and **answeredInRound**, which should be validated before use. Without these checks, the function may operate on invalid or outdated price data.

```
def _get_oracle_rate(oracle_addr: address, oracle_reverse: bool) -> UInt256Rational:
    convertion_rate_numerator: uint256 = 0
    convertion_rate_denominator: uint256 = 0
    if oracle_reverse:
        return UInt256Rational(
            numerator=10 ** convert(staticcall AggregatorV3Interface(oracle_addr).decimals(), uint256),
            denominator=convert((staticcall AggregatorV3Interface(oracle_addr).latestRoundData()).answer,
uint256)
        )
    else:
        return UInt256Rational(
            numerator=convert((staticcall AggregatorV3Interface(oracle_addr).latestRoundData()).answer,
uint256),
            denominator=10 ** convert(staticcall AggregatorV3Interface(oracle_addr).decimals(), uint256)
        )
```

Remediation:

Add validation checks after calling **latestRoundData()** to ensure the **answer** is positive and the data is not stale. Consider checking that updatedAt is recent relative to a defined threshold and that **answeredInRound** is greater than or equal to **roundId**.

Commentary from the client:

“The code in _get_oracle_rate will revert for negative values. Other than that, is assumed that the answer is the most up to date, even if not updated recently. Otherwise it could block actions on all loans leading to a DoS situation.”

ZHAR2-8 | Liquidation Fee Uses Wrong Source During Loan Execution

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

/contracts/v2/P2PLendingV2Liquidation.vy

Description:

In **create_loan**, the contract sets the **full_liquidation_fee** from the **base** values (base.full_liquidation_fee) when creating the loan. However, during **liquidate_loan**, the fees are read from **base** again instead of using the values stored in the loan struct (**loan.full_liquidation_fee**)

If the **base** fees are changed after loan creation, existing loans could be liquidated with different fees than intended

```
liquidation_fee_collateral: uint256 = min(
    loan.collateral_amount,
    outstanding_debt * base.full_liquidation_fee * rate.denominator * collateral_token_decimals // (BPS * rate.numerator * payment_token_decimals)
)
```

```
    liquidation_fee_collateral: uint256 = min(loan.collateral_amount, outstanding_debt * base.full_liquidation_fee * rate.denominator * collateral_token_decimals // (BPS * rate.numerator * payment_token_decimals))
```

Remediation:

Consider using **loan.full_liquidation_fee** instead of **base.full_liquidation_fee**:

```
liquidation_fee_collateral: uint256 = min(
    loan.collateral_amount,
--  outstanding_debt * base.full_liquidation_fee * rate.denominator * collateral_token_decimals // (BPS * rate.numerator * payment_token_decimals)
++  outstanding_debt * loan.full_liquidation_fee * rate.denominator * collateral_token_decimals // (BPS * rate.numerator * payment_token_decimals)
)
```

ZHAR2-13 | Inaccurate interest delta calculation for called loans

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Description:

The **_max_interest_delta** function calculates the remaining loan duration based solely on **loan.maturity**. It does not account for loans that have been called, where the effective repayment deadline is reduced to **call_time + call_window**. As a result, when refinancing a called loan, the function overestimates the remaining time, leading to an incorrect **borrower_compensation** calculation.

```
def _max_interest_delta(loan: base.Loan, offer: base.Offer, new_principal: uint256) -> uint256:
    return convert(
        max(
            0,
            (convert(new_principal * offer.apr, int256) - convert(loan.amount * loan.apr, int256)) *
convert(loan.maturity - block.timestamp, int256) // convert(365 * 86400 * BPS, int256)
        ),
        uint256
    )
```

Remediation:

Modify **_max_interest_delta** to consider the loan's call status. When calculating the time delta, use the earlier of the original maturity or the call expiration timestamp.

Commentary from the client:

"Refinances are not available for defaulted loans (including called loans)."

ZHAR2-15 | Missing Upper Bound Check on Loan Duration

Acknowledged

Severity:	Probability:	Impact:
Low	Rare	Low

Description:

The **_check_offer_validity** function validates that a loan duration is greater than zero but does not enforce any upper bound on the duration value:

```
assert offer.offer.duration > 0, "duration is 0"
```

As a result, a lender may accidentally specify a large duration (e.g., due to a typo or a extra zero). Once the offer is accepted, the resulting loan may last decades or longer than intended.

Because loan duration is immutable after acceptance, this creates a permanent and unrecoverable misconfiguration risk for the lender.

Remediation:

Introduce a maximum allowed loan duration and enforce it during offer validation. For example 4 years:

```
++ MAX_LOAN_DURATION = 365 * 24 * 60 * 60  # 4 year

assert offer.offer.duration > 0, "duration is 0"
++ assert offer.offer.duration <= MAX_LOAN_DURATION, "duration exceeds maximum"
```

Commentary from the client:

“There's no hard limit for loans durations. Mistakes are mitigated via the app UI.”

ZHAR2-12 | Missing Protocol Fee Initialization in Constructor

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

In the **__init__** function of the **P2PLendingV2Erc20** contract, the parameters **_protocol_upfront_fee** and **_protocol_settlement_fee** are accepted but not assigned to their corresponding storage variables in P2PLendingV2Base (**base.protocol_upfront_fee** and **base.protocol_settlement_fee**). As a result, the protocol fees default to zero upon deployment, regardless of the values passed during initialization.

```
def __init__(
    _payment_token: address,
    _collateral_token: address,
    _oracle_addr: address,
    _oracle_reverse: bool,
    _kyc_validator_addr: address,
    _protocol_upfront_fee: uint256,
    _protocol_settlement_fee: uint256,
    _protocol_wallet: address,
    _max_protocol_upfront_fee: uint256,
    _max_protocol_settlement_fee: uint256,
    _partial_liquidation_fee: uint256,
    _full_liquidation_fee: uint256,
    _refinance_addr: address,
    _liquidation_addr: address,
    _vault_impl_addr: address,
    _transfer_agent: address
):

    """
    @notice Initialize the contract with the given parameters.
    @param _payment_token The address of the payment token.
    @param _collateral_token The address of the collateral token.
    @param _oracle_addr The address of the oracle contract for collateral valuation.
    @param _oracle_reverse Whether the oracle returns the collateral price in reverse (i.e., 1 / price).
    @param _protocol_upfront_fee The percentage (bps) of the principal paid to the protocol at origination.
    @param _protocol_settlement_fee The percentage (bps) of the interest paid to the protocol at settlement.
    @param _protocol_wallet The address where the protocol fees are accrued.
    @param _max_protocol_upfront_fee The maximum percentage (bps) of the principal that can be charged as protocol upfront fee.
```

```
    @param _max_protocol_settlement_fee The maximum percentage (bps) of the interest that can be charged as protocol settlement fee.
    @param _partial_liquidation_fee The percentage (bps) of the principal that is charged as a liquidation fee when a loan is partially liquidated.
    @param _full_liquidation_fee The percentage (bps) of the principal that is charged as a liquidation fee when a loan is fully liquidated.
    @param _refinance_addr The address of the facet contract implementing the refinance functionality.
    @param _liquidation_addr The address of the facet contract implementing the liquidation functionality.
    @param _vault_impl_addr The address of the vault implementation contract.
    @param _transfer_agent The wallet address for the transfer agent role.
    """

    base.__init__()

    payment_token = _payment_token
    collateral_token = _collateral_token
    oracle_addr = _oracle_addr
    oracle_reverse = _oracle_reverse
    kyc_validator_addr = _kyc_validator_addr
    max_protocol_upfront_fee = _max_protocol_upfront_fee
    max_protocol_settlement_fee = _max_protocol_settlement_fee
    refinance_addr = _refinance_addr
    liquidation_addr = _liquidation_addr
    vault_impl_addr = _vault_impl_addr
    collateral_token_decimals = 10 ** convert(staticcall IERC20Detailed(_collateral_token).decimals(), uint256)
    payment_token_decimals = 10 ** convert(staticcall IERC20Detailed(_payment_token).decimals(), uint256)
    base.protocol_wallet = _protocol_wallet
    base.transfer_agent = _transfer_agent
    base.partial_liquidation_fee = _partial_liquidation_fee
    base.full_liquidation_fee = _full_liquidation_fee

    offer_sig_domain_separator = keccak256(
        abi_encode(
            base.DOMAIN_TYPE_HASH,
            keccak256(base.ZHARTA_DOMAIN_NAME),
            keccak256(base.ZHARTA_DOMAIN_VERSION),
            chain.id,
            self
        )
    )
```

Remediation:

Update the constructor to correctly assign the **_protocol_upfront_fee** and **_protocol_settlement_fee** arguments to the respective storage variables in the base contract.

ZHAR2-10 | Create Loan Can Overcommit Lender and Cause Revert via Protocol Upfront Fee

Fixed

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Description:

In **create_loan**, the protocol upfront fee is added on top of the loan principal when transferring funds from the lender:

```
self._transfer_funds(loan.lender, loan.borrower, loan.amount - loan.origination_fee_amount)

if loan.protocol_upfront_fee_amount > 0:
    self._transfer_funds(loan.lender, base.protocol_wallet, loan.protocol_upfront_fee_amount)
```

However, the lender's committed liquidity is only reduced by **principal** in **_check_and_update_offer_state**. This allows the contract to transfer the principal + protocol upfront fee even if the lender approved liquidity only equal to the principal.

Impact:

- The lender can be overcommitted beyond their approved liquidity.
- Loan creation may fail if the lender's approved liquidity is insufficient to cover the principal plus the protocol fee.

Scenario:

- Lender offers **1.5 ETH** liquidity.
- Borrower takes a loan of **1.5 ETH** with **origination_fee = 0** and **protocol_upfront_fee = 1%**.
- **_check_and_update_offer_state** only checks **principal <= available_liquidity.**
- **create_loan** tries to transfer **1.5 ETH + 0.015 ETH (protocol fee)** from lender.
- Transaction reverts because lender's approved liquidity is insufficient to cover the fee.
  OR
  Transactions successful but the lender send more funds than their approved liquidity of 1.5 eth.

Remediation:

Clarify in the documents that **available_liquidity/commited_liquidity** track principal only, while the lender also pays the protocol upfront fee at loan creation. Explicitly state that lenders must approve/hold funds for principal + protocol_upfront_fee_amount − origination_fee_amount, otherwise **create_loan** will revert.

ZHAR2-14 | Inconsistent Documentation Regarding Loan Default Status in liquidate_loan

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Description:

The **liquidate_loan** function in **P2PLendingV2Liquidation.vy** contains a discrepancy between its documentation and implementation. The function's docstring explicitly states that it "Fully liquidates a defaulted loan." However, the assertion checking for the default status (**base._is_loan_defaulted**) is commented out, and the code includes specific logic to handle the full liquidation of non-defaulted loans if certain Loan-to-Value (LTV) conditions are met. This inconsistency makes the intended behavior ambiguous, as the code explicitly permits pre-maturity liquidation while the documentation implies it is restricted to defaulted loans.

```
def liquidate_loan(
    loan: base.Loan,

    payment_token: address,
    collateral_token: address,
    oracle_addr: address,
    oracle_reverse: bool,
    kyc_validator_addr: address,
    collateral_token_decimals: uint256,
    payment_token_decimals: uint256,
    offer_sig_domain_separator: bytes32,
    vault_impl_addr: address,
):

    """
    @notice Fully liquidates a defaulted loan. Can be called by anyone.
    @param loan The loan to be soft liquidated.
    """

    assert base._is_loan_valid(loan), "invalid loan"
    # assert base._is_loan_defaulted(loan), "loan not defaulted"
    liquidator: address = msg.sender if not base.authorized_proxies[msg.sender] else tx.origin

    if not base._is_loan_defaulted(loan):

        current_interest: uint256 = base._compute_settlement_interest(loan)
```

```
    convertion_rate: base.UInt256Rational = base._get_oracle_rate(oracle_addr, oracle_reverse)
    current_ltv: uint256 = base._compute_ltv(loan.collateral_amount, loan.amount + current_interest,
convertion_rate, payment_token_decimals, collateral_token_decimals)

    assert loan.liquidation_ltv > 0, "not defaulted, partial disabled"
    assert current_ltv >= loan.liquidation_ltv, "not defaulted, ltv lt partial"

    principal_written_off: uint256 = 0
    collateral_claimed: uint256 = 0
    liquidation_fee: uint256 = 0
    principal_written_off, collateral_claimed, liquidation_fee = base._compute_partial_liquidation(
        loan.collateral_amount,
        loan.amount + current_interest,
        loan.initial_ltv,
        loan.partial_liquidation_fee,
        convertion_rate,
        payment_token_decimals,
        collateral_token_decimals
    )

    assert principal_written_off >= loan.amount + current_interest, "not defaulted, partial possible"
```

Remediation:

Clarify the intended business logic for full liquidations. If the protocol intends to allow full liquidation before maturity when LTV thresholds are breached, update the documentation to accurately reflect this capability.


hexens
x
ZHARTA