RED4SEC

Security Audit Report

05/31/2022

Zharta protocol-v1

RED4SEC

Content

Introduction 3
Disclaimer 3
Scope 4
Executive Summary 5
Conclusions 6
Vulnerabilities 7
List of vulnerabilities 7
Vulnerability details 8
Receive funds without collateral 9
Broken deposit 10
Uncontrolled activeLenders decrement 12
Operations prone to Front Running 13
Missing logic in the collateral whitelist removal 14
Lack of Inputs Validation 15
Invalidate method is prone to errors 17
Possible wrong getLoan logic 18
Dangerous default loan StartTime 19
Arrays with too low bounds 20
Reentrancy Patterns 21
Lack of Documentation 23
Outdated Compiler 24
Unused Libraries 25
GAS Optimization 26
Unsecured Ownership Transfer 31
Lack of Event Index 32
Emit Events on State Changes 33
Contracts Management Risks 34
Wrong Fallback Implementation 35
Beta Compiler 36
Annexes 37
Annex A – Vulnerabilities Severity 37

Introduction

Zharta is a platform for its users to transform their NFTs into instant loans. The objective is for the users to get instant loans in real-time with their NFT collection, without losing ownership over them.

With Zharta the user can select the assets they want to collaterize or the amount they need and get the loan instantly. It can also be used to lend, by depositing in one of Zharta's lending pools and earn interest. After the user lends, they can bid on the defaulted assets.

As solicited by **Zharta** and as part of the vulnerability review and management process, Red4Sec has been requested to perform a security code audit in order to evaluate the security of the **Zharta protocol-v1** project.

The report includes specifics retrieved from the audit for all the existing vulnerabilities of **Zharta protocol-v1**. The performed analysis shows that the smart contracts do not currently contain known vulnerabilities.

Disclaimer

This document only represents the results of the code audit conducted by Red4Sec Cybersecurity and should not be used in any way to make investment decisions or as investment advice on a project.

Likewise, the report should not be considered neither "endorsement" nor "disapproval" of the guarantee of the correct business model of the analyzed project, nor as guarantee on the operation or viability of the implemented financial product.

Red4Sec makes full effort and applies every resource available for each audit, however it does not warrant the function, nor the safety of the project and it cannot be deemed a sufficient assessment of the code's utility and safety, bug-free status, or any other declarations of the project. Additionally, Red4Sec makes no security assessments or judgments about the underlying business strategy, or the individuals involved in the project.

Blockchain technology and cryptographic assets come with its own new risks and challenges, where the ecosystem, platform, its programming language, and other software related to said technology can have vulnerabilities that could lead to exploits. As a result, the audit cannot guarantee the explicit security of the audited projects.

The audit reports can be used to improve the code quality of smart contracts, to help limit the vectors of attack and to lower the high level of risks associated with utilizing new and continually changing technologies such as cryptographic tokens and blockchain, but they are unable to detect any future security concerns with the related technologies.

Scope

Red4Sec Cybersecurity has made a thorough audit of the **Zharta protocol-v1** security level against attacks, identifying possible errors in the design, configuration or programming; therefore, guaranteeing the availability, integrity and confidentiality of the project and the possible assets treated and stored.

The scope of this evaluation includes the following items provided by **Zharta**:

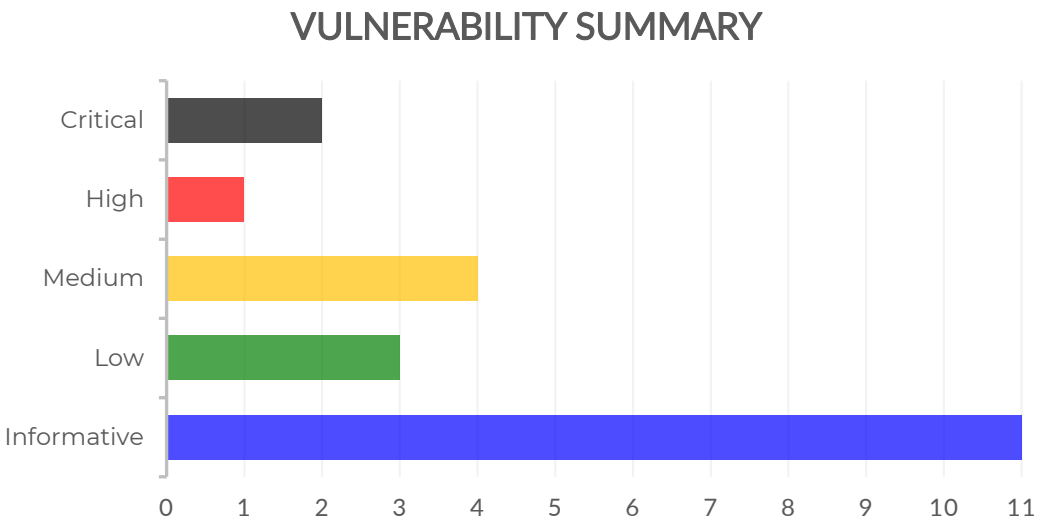
- https://gitlab.com/z106/smart-contracts-eth
  - `Branch: main`
  - `Commit: 2b1b3e7d99d2e1b76b57e65a2d92fddc806401b4`

Executive Summary

The security audit against **Zharta protocol-v1** has been conducted between the following dates: **05/16/2022** and **05/31/2022**.

Once the analysis of the technical aspects has been completed, the performed analysis shows that the audited source code does not contain known vulnerabilities.

During the analysis, a total of **21 vulnerabilities** were detected, these vulnerabilities have been classified by the following level of risks, defined in Annex A.



Conclusions

To this date, **31/05/2022**, the general conclusion resulting from the conducted audit, is that the **Zharta project is secure** and does not currently present known vulnerabilities that could compromise the security of the project.

The general conclusions of the performed audit are:

- **Critical and high-risk vulnerabilities** were detected during the security audit. These vulnerabilities posed a great risk for **Zharta** project and were promptly fixed.
- A few of the detected vulnerabilities were proven to affect the design of the product, so it was required a detailed analysis to find out if the current state satisfactorily met the objectives in a safe way.
- The project currently lacks any documentation and there is no technical documentation to check if the current implementation meets the needs and purpose of the project.
- It was detected that the logic of the contract **allowed the owner to alter certain values of the contract** at will, allowing to obtain an advantageous position in certain situations.However the **Zharta** team promptly corrected the issue.
- Due to the nature of the assets handled and the particularity of each NFT implementation, it becomes essential to study and audit each case before adding it to the available collaterals. Therefore, the contract must always operate with the whitelist option activated.
- Certain methods **did not make the necessary input checks** in order to guarantee the integrity and expected arguments format.
- It is important to highlight that Vyper language is a beta language, which is in current development, and it is prone to new bugs and breaking changes. Therefore, this audit is unable to detect any future security concerns with Vyper compiler.
- A **few low impact issues** were detected and classified only as informative, but they will continue to help Zharta improve the security and quality of its developments.
- All the proposed recommendations that were considered necessary by Red4Sec in order to improve the security of the project were properly applied by the Zharta team.

Vulnerabilities

In this section, you can find a detailed analysis of the vulnerabilities encountered upon the security audit.

List of vulnerabilities

Below, we have gathered a complete list of the vulnerabilities detected by Red4Sec, presented and summarized in a way that can be used for risk management and mitigation.

Table of vulnerabilities

ID	Vulnerability	Risk	State
ZH-01	Receive funds without collateral	**Critical**	**Fixed**
ZH-02	Broken deposit	**Critical**	**Fixed**
ZH-03	Uncontrolled activeLenders decrement	**High**	**Fixed**
ZH-04	Operations prone to Front Running	**Medium**	**Assumed**
ZH-05	Missing logic in the collateral whitelist removal	**Medium**	**Fixed**
ZH-06	Lack of Inputs Validation	**Medium**	**Fixed**
ZH-07	Invalidate method is prone to errors	**Medium**	**Fixed**
ZH-08	Possible wrong getLoan logic	**Low**	**Closed**
ZH-09	Dangerous default loan StartTime	**Low**	**Fixed**
ZH-10	Arrays with too low bounds	**Low**	**Fixed**
ZH-11	Reentrancy Patterns	**Informative**	**Fixed**
ZH-12	Lack of Documentation	**Informative**	**Assumed**
ZH-13	Outdated Compiler	**Informative**	**Fixed**
ZH-14	Unused Libraries	**Informative**	**Fixed**
ZH-15	GAS Optimization	**Informative**	**Fixed**
ZH-16	Unsecured Ownership Transfer	**Informative**	**Fixed**
ZH-17	Lack of Event Index	**Informative**	**Fixed**
ZH-18	Emit Events on State Changes	**Informative**	**Fixed**
ZH-19	Contracts Management Risks	**Informative**	**Fixed**
ZH-20	Wrong Fallback Implementation	**Informative**	**Fixed**
ZH-21	Beta Compiler	**Informative**	**Assumed**

Vulnerability details

In this section, we provide the details of each of the detected vulnerabilities indicating the following aspects:

- Category
- Active
- Risk
- Description
- Recommendations

Receive funds without collateral

Identifier	Category	Risk	State
ZH-01	Unprotected Withdrawal	**Critical**	**Fixed**

There has been a failure found in the `Loans` contract through which an attacker could receive the fund of an authorized loan and also recover the collateral.

It has been detected that the `cancelPendingLoan` method does not check that the loan is pending authorization, it only checks that it exists.

```
@external
def cancelPendingLoan(_loanId: uint256):
    assert self.loansCore.isLoanCreated(msg.sender, _loanId), "The sender has not created a loan with the given ID"
```

This allows a malicious user to call the `pay` method of the `Loans` contract, and then call the `cancelPendingLoan` method. The user will receive the collateral (NFT) and will also obtain the previously approved loan.

Recommendations

- Modify the `cancelPendingLoan` method to ensure that the `Loan` has not been initiated or invalidated.

Source Code References

- contracts/Loans.vy#L450

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/39

Broken deposit

Identifier	Category	Risk	State
ZH-02	Business Logic Errors	**Critical**	**Fixed**

It is possible to drain the tokens from the `LendingPoolCore` contract by taking advantage of a flaw in the logic of the `deposit` method.

The `deposit` method of the `LendingPoolCore` contract, contains a condition that allows a user to directly call it . This method is intended to be invoked by `lendingPoolPeripheral`, so it only registers the user's deposit without performing the `transferFrom` of the tokens.
The contract is designed for the user to call `LendingPoolPeripheral.deposit`, and this method does transfer the tokens of the users. However, the `or msg.sender == _lender` condition of the `deposit` method allows it to be called directly as long as the `msg.sender` is specified as `_lender`.

```
@external
def deposit(_lender: address, _amount: uint256) -> bool:
    # _amount should be passed in wei

    assert msg.sender == self.lendingPoolPeripheral or msg.sender == _lender, "Only defined lending pool peripheral or the lender can deposit"
    assert _amount > 0, "Amount deposited has to be higher than 0"
```

The same conditional allows calling the `withdraw` method of `LendingPoolCore`, which will transfer the tokens to the user based on their previous deposits. Although in this case it is also possible to withdraw the tokens using the `LendingPoolPeripheral.withdraw` method, since the deposits are already registered in the contract.

```
@external
def withdraw(_lender: address, _amount: uint256) -> bool:
    # _amount should be passed in wei

    assert msg.sender == self.lendingPoolPeripheral or msg.sender == _lender, "Only defined lending pool peripheral or the lender can withdraw"
    assert self._computeWithdrawableAmount(_lender) >= _amount, "The lender has less funds deposited than the amount requested"
    assert self.fundsAvailable >= _amount, "Not enough funds in the pool to be withdrawn"

    newDepositAmount: uint256 = self._computeWithdrawableAmount(_lender) - _amount
    newLenderSharesAmount: uint256 = self._computeShares(newDepositAmount)
    self.totalSharesBasisPoints -= (self.funds[_lender].sharesBasisPoints - newLenderSharesAmount)

    self.funds[_lender] = InvestorFunds(
        {
            currentAmountDeposited: newDepositAmount,
            totalAmountDeposited: self.funds[_lender].totalAmountDeposited,
            totalAmountWithdrawn: self.funds[_lender].totalAmountWithdrawn + _amount,
            sharesBasisPoints: newLenderSharesAmount,
            activeForRewards: True
        }
    )

    if self.funds[_lender].currentAmountDeposited == 0:
        self.funds[_lender].activeForRewards = False
        self.activeLenders -= 1

    self.fundsAvailable -= _amount

    if not IERC20Token(self.erc20TokenContract).transfer(_lender, _amount):
        raise "Withdrawal transfer error"

    return True
```

Therefore, if an attacker makes a `deposit` calling directly to the `LendingPoolCore` contract and later makes a `withdraw`, in any of its contracts, the user will receive the arbitrary amount defined in the `deposit` method without any cost or collateral.

Recommendations

- It is convenient to remove the `or msg.sender == _lender` condition from the `deposit` and `withdraw` methods in the `LendingPoolCore` contract.
- Move the `transferFrom` logic from the contract `LendingPoolPeripheral` to the `LendingPoolCore` contract.

Source Code References

- LendingPoolCore.vy#L112
- LendingPoolCore.vy#L152
- LendingPoolPeripheral.vy#L272

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/40

Uncontrolled activeLenders decrement

Identifier	Category	Risk	State
ZH-03	Denial of Service	**High**	**Fixed**

It has been possible to verify that there is a logic error in the `LendingPoolCore` contract which may affect the functionality and stability of the contract and lead to a denial of service.

The `withdraw` method of the `LendingPoolCore` contract does not check that the set `_amount` is greater than 0. In case of invoking with the following arguments `_amount = 0` and `_lender = msg.sender`, all the initial asserts are satisfied and then `InvestorFunds.activeForRewards = false` is set and `self.activeLenders -= 1` is subtracted.

```
@external
def withdraw(_lender: address, _amount: uint256) -> bool:
    # _amount should be passed in wei

    assert msg.sender == self.lendingPoolPeripheral or msg.sender == _lender, "Only defined lending pool peripheral or the lender can withdraw"
    assert self._computeWithdrawableAmount(_lender) >= _amount, "The lender has less funds deposited than the amount requested"
    assert self.fundsAvailable >= _amount, "Not enough funds in the pool to be withdrawn"
```

To perform this operation, it is not necessary to have previously invested, so if an attacker makes multiple calls with `_amount = 0`, it will reduce `activeLenders` and this will reflect an incorrect value that is inconsistent with the expected value.

```
if self.funds[_lender].currentAmountDeposited == 0:
    self.funds[_lender].activeForRewards = False
    self.activeLenders -= 1
```

The greatest problem occurs when this counter is artificially decremented to zero and due to Vyper's native protection against underflow operations, it prevents legitimate users with active loans from being able to withdraw all of their funds.

Recommendations

- Verify that `_amount` is greater than 0.

Source Code References

- LendingPoolCore.vy#L175

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/41

Operations prone to Front Running

Identifier	Category	Risk	State
ZH-04	Business Logic Errors	**Medium**	**Assumed**

The design of the workflows of the **Zharta** project are prone to a front running attack.

For the correct functioning of the project, the continuous intervention of the owner of the contract is necessary to modify the whitelist, change the maximum and minimum amounts of the loans, among others... but mainly to validate, invalidate, settle and cancel loans.

Due to market fluctuations, the collaterals of the loans can change in value on very short periods, which would imply the need for the owner to update the affected loans.

To carry out these operations, the owner will have to broadcast as many transactions as the loans it wants to modify, which gives a window of time in which users can anticipate these transactions and operate with their loans with the values prior to market changes.

Recommendations

- It is convenient that the **Zharta** team studies in depth the possible implications of this issue and implement the necessary protections.

References

- https://swcregistry.io/docs/SWC-114

Source Code References

- contracts/Loans.vy#L339
- contracts/Loans.vy#L361
- contracts/Loans.vy#L422
- contracts/Loans.vy#L450
- contracts/Loans.vy#L192
- contracts/Loans.vy#L204
- contracts/LoansCore.vy#L357
- contracts/LoansCore.vy#L367
- contracts/LoansCore.vy#L375
- contracts/LoansCore.vy#L387
- contracts/LoansCore.vy#L403
- contracts/LoansCore.vy#L411
- contracts/LoansCore.vy#L419
- contracts/LoansCore.vy#L427
- contracts/LoansCore.vy#L435

Fixes Review

Zharta Notes: *That is a risk the protocol assumes, and it is encoded in our risk management engine. It is impossible for a borrower (owner of the loan) to cancel an ongoing loan. An ongoing loan can only either be paid or liquidated at the end of the maturity date if it was not repaid.*

Missing logic in the collateral whitelist removal

Identifier	Category	Risk	State
ZH-05	Business Logic Errors	**Medium**	**Fixed**

It has been possible to verify that there is a logic error in the `Loans` contract that can affect the functionality and stability of the contract.

The `addCollateralToWhitelist` method of the `Loans` contract sets the address received in the `whitelistedCollaterals` mapping to `True`. Additionally, if it is not contained in the `whitelistedCollateralsAddresses` array, it is added.

```
@external
def addCollateralToWhitelist(_address: address) -> bool:
    assert msg.sender == self.owner, "Only the contract owner can add collateral addresses to the whitelist"
    assert _address.is_contract == True, "The _address sent does not have a contract deployed"

    self.whitelistedCollaterals[_address] = True
    if _address not in self.whitelistedCollateralsAddresses:
        self.whitelistedCollateralsAddresses.append(_address)

    return True


@external
def removeCollateralFromWhitelist(_address: address) -> bool:
    assert msg.sender == self.owner, "Only the contract owner can add collateral addresses to the whitelist"
    assert _address in self.whitelistedCollateralsAddresses, "The collateral is not whitelisted"

    self.whitelistedCollaterals[_address] = False

    return True
```

However, we can see how the `removeCollateralFromWhitelist` function removes that flag from the `whitelistedCollaterals` mapping, but does not delete it from the `whitelistedCollateralsAddresses` array, this will cause the mapping and the array to be out of sync. Therefore, if any user or dApp queries said array using the `getWhitelistedCollateralsAddresses` method or directly accessing the exposed property, you will get values that do not correspond with the reality, this affects the expected logic and functionality of the contract.

Recommendations

- Add the necessary logic to keep the `whitelistedCollaterals` and `whitelistedCollateralsAddresses` registries synchronized during the collateral whitelist update processes.

Source Code References

- contracts/Loans.vy#L206

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/51

Lack of Inputs Validation

Identifier	Category	Risk	State
ZH-06	Improper Input Validation	**Medium**	**Fixed**

Certain methods of the different contracts in the `Zharta` project do not properly check the arguments, which can lead to major errors. Below we list the most significant examples.

The logic established in the `reserve` method of the `Loans` contract apparently allows the user to decide his own interest through the `_interest` argument, if so, it is important to check that the value stipulated by the user is within the expected range and in no case exceeds the 100%.

- contracts/Loans.vy#L300

The `addWhitelistedAddress` method of the `LendingPoolPeripheral` contract allows adding addresses to the whitelist, however, when accepting a Non-Fungible Token (NFT) it may be convenient to add additional checks to verify that the destination contract is really an NFT, for this the ERC-165 (Interface Detection) standard can be used and verify that the destination supports the `ERC721nonFungibleContract.supportsInterface(type(IERC721).interfaceId)` standard. Furthermore, it is not verified that the provided addresses sent by arguments are valid, for example, it allows `address(0)`.

This behavior has been observed in:

- contracts/LendingPoolPeripheral.vy#L222

During the execution of the `__init__` constructor in the `LoansCore`, `LendingPoolCore` and `LendingPoolPeripheral` contracts, it is not verified that the provided addresses sent through the `_loansPeripheral`, `_lendingPoolPeripheral`, `_erc20TokenContract`, `_loansContract`, `_erc20TokenContract` and `_protocolWallet` arguments are valid, so it allows the address to be set to `address(0)`, it is advisable to use the same verifications in the constructor as in the methods that make the changes to the values. Otherwise, unexpected errors could occur.

- contracts/LoansCore.vy#L124
- contracts/LendingPoolCore.vy#L94-95
- contracts/LendingPoolPeripheral.vy#L120-122
- contracts/LendingPoolPeripheral.vy#L158
- contracts/LendingPoolPeripheral.vy#L192

In the same way, during the logic of the `__init__` constructor in the `Loans` contract, none of the received arguments are verified. For example, `_maxAllowedLoans`, `_maxAllowedLoanDuration`, `_minLoanAmount`, `_maxLoanAmount` should be valid, as for instance, be greater than zero. It is advisable to use the same verifications in the constructor as in the methods that make the changes to the values. Otherwise, unexpected errors could occur.

- contracts/Loans.vy#L119-122
- contracts/Loans.vy#L174
- contracts/Loans.vy#L183
- contracts/Loans.vy#L214
- contracts/Loans.vy#L224

The same problem has been detected in the constructor of the `LendingPoolPeripheral` contract, none of the received arguments are verified. For example, `protocolFeesShare` and `maxCapitalEfficienty` should be valid, as for instance, be greater than zero and **within the expected range**. It is advisable to use the same verifications in the constructor as in the methods that make the changes to the values. Otherwise, unexpected errors could occur.

- contracts/LendingPoolPeripheral.vy#L131-132
- contracts/LendingPoolPeripheral.vy#L149
- contracts/LendingPoolPeripheral.vy#L167

In the `LoansCore` contract, the `addLoan` method does not verify that the value sent through the `_interest` argument is valid, so it allows to establish an interest above 100%.

- contracts/LoansCore.vy#L326

The `setLoansCoreAddress` method of the `Loans` contract allows to modify the address of the `LoansCore` contract asoiciated to the `loansCoreAddress` variable.
However, it is not verified that the new address sent by `argument` is valid, additionally, it is highly recommended to check that the `Loans` contract is the owner of the new `LoansCore` contract before modifying the value of the `loansCoreAddress` variable.

- contracts/Loans.vy#L238-239

During the execution of the following methods, it is not verified that the provided addresses sent by arguments are valid, so it allows `address(0)`.

- contracts/LendingPoolPeripheral.vy#L297
- contracts/Loans.vy#L165
- contracts/Loans.vy#L192
- contracts/Loans.vy#L204
- contracts/Loans.vy#L234
- contracts/Loans.vy#L244

References

- https://eips.ethereum.org/EIPS/eip-165

Fixes Review

This issue was addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/42

Invalidate method is prone to errors

Identifier	Category	Risk	State
ZH-07	Improper Input Validation	**Medium**	**Fixed**

The `Loans` contract does not include the necessary verifications to prevent incorrect administration, which can lead to canceling loans and returning collateral without ensuring loan payment.

Although this may be the expected behavior, the Red4Sec team does not have the necessary documentation to determine it, so it is recommended that Zharta's team studies the case in depth.

The `invalidate` method contains incorrect logic, since it returns the collateral to the `_borrower` and invalidates the contract, however it does not check that the amount has been returned by the user, nor does it check that the user's collateral is available so it can be returned.

```
@external
def invalidate(_borrower: address, _loanId: uint256):
    assert msg.sender == self.owner, "Only the contract owner can invalidate loans"
    assert self.loansCore.isLoanCreated(_borrower, _loanId), "This loan has not been created for the borrower"
```

In the event that the borrower does not deposit the collateral of the loan, it will never be invalidated by the admin because the `safeTransferFrom` statement will fail.

In case the user does deposit the collateral of the loan and it is validated by the admin through the `validate` method, he will have received the amount of the loan, but if by mistake, an administrator invalidates a loan initiated, validated and paid through the `invalidate` method. The user will receive the collateral (NFTs) and will also maintain the balance initially received from the loan.

Additionally, it is worth mentioning that it is possible to validate a previously invalidated loan, which would provide the funds to a previously invalidated loan.
The methods that we consider also should verify that the loan is already invalidated are: `validate`, `pay`, `cancelPendingLoan`.

Recommendations

- Modify the `invalidate` method to ensure that the `Loan` has not been initiated or that the loan has been returned.

Source Code References

- contracts/Loans.vy#L450
- contracts/Loans.vy#L361

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/43

Possible wrong getLoan logic

Identifier	Category	Risk	State
ZH-08	Design Weaknesses	**Low**	**Closed**

It has been possible to verify that there is a logic error in the `LoansCore` contract that can affect the functionality and integrity of the services that rely on the information stored in the contract.

The `getLoan` method of the `LoansCore` contract returns the details of the `Loan` of `borrower` with the `id` specified in the argument. In order to return it, it checks that the `Loan` has been initiated or that it has been invalidated.

```
@view
@external
def getLoan(_borrower: address, _loanId: uint256) -> Loan:
  if self._isLoanStarted(_borrower, _loanId) or self._isLoanInvalidated(_borrower, _loanId):
    return self.loans[_borrower][_loanId]
  return empty(Loan)
```

We believe that the expected logic is that the `Loan` has been initiated **and not** invalidated, although the structure of the `Loan` itself already returns whether it has been invalidated or not, this condition may be unnecessary. The doubts about the expected logic, increased by the lack of documentation in this regard, make it convenient to review the logic of said method.

Source Code References

- contracts/LoansCore.vy#L258

Fixes Review

It has been verified that the necessary verifications are implicit in the state of the loan.

Dangerous default loan StartTime

Identifier	Category	Risk	State
ZH-09	Design Weaknesses	**Low**	**Fixed**

It is possible that the rationale around the logic of `getLoanStartTime` in the `LoansCore` contract is prone to errors.

The `getLoanStartTime` method of the `LoansCore` contract returns the `startTime` value and in the case the `Loan` does not find a 0.

```
@view
@external
def getLoanStartTime(_borrower: address, _loanId: uint256) -> uint256:
    if _loanId < len(self.loans[_borrower]):
        return self.loans[_borrower][_loanId].startTime
    return 0
```

From Red4Sec we consider that the default value in this case must be `uint2560.max`, or even better, to revert the execution for non-existent loans.

Keep in mind that the method intends to return the start date. The most logical purpose is to know if it has been initialized or not, and a default value of 0 could mean that something, that doesn't even exist, has already been initiated. Additionally, this makes it impossible to differentiate when a `Loan` is created using the `addLoan` method, since it also sets that value to 0.

Recommendations

- It is advisable to study reversing the transaction or changing the default value.

Source Code References

- contracts/LoansCore.vy#L196
- contracts/LoansCore.vy#L338

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/45

Arrays with too low bounds

Identifier	Category	Risk	State
ZH-10	Denial of Service	**Low**	**Fixed**

One of the benefits or problems associated with the Vyper language is that the definition of dynamic arrays and fixed strings must have a maximum size defined but, if this size is not defined with a wide margin, it can cause denial of service problems in the future.

The most notable example of this issue can be seen in the `erc20TokenSymbol` method, where the output is defined as a `string[10]`. This prevents the use of a token whose `symbol` is greater than 10 characters, since otherwise the call to this method would fail during execution in the EVM.

```
@view
@external
def erc20TokenSymbol() -> String[10]:
    return IERC20(self.lendingPool.erc20TokenContract()).symbol()
```

Another example is the number of `collaterals` that `Loan` allows, currently limited to 10, and which may be insufficient. Therefore, it is recommended to increase it to avoid encountering usage limits in the future.

Recommendations

- Consider increasing the maximums established for the mentioned variables.

References

- https://vyper.readthedocs.io/en/stable/types.html#dynamic-arrays
- https://vyper.readthedocs.io/en/stable/types.html#strings

Source Code References

- contracts/Loans.vy#L281
- contracts/Loans.vy#L40
- contracts/LoansCore.vy#L17
- contracts/LoansCore.vy#L185-188

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/46

Reentrancy Patterns

Identifier	Category	Risk	State
ZH-11	Reentrancy	**Informative**	**Fixed**

The Reentrancy attack is a vulnerability that occurs when external contract calls can make new calls to the calling contract, before the initial execution is completed. For a function, this means that the state of the contract could change in the middle of its execution as a result of a call to an untrusted contract or the use of a low-level function with an external address.

Certain tokens such as the `ERC777` facilitate reentrancy attacks, and in the case of sending ether, depending on the process, it can also facilitate attacks of this type.

Therefore, transfers to contracts could invoke the execution of the payment method of the recipient, and the recipient may redirect the execution to himself or to another contract. Making adequate changes to the storage before external calls will prevent reentrance-type attacks.

In the case of the `reserve` method in the `Loans` contract, the values are updated after the external calls are made so the call can be redirected to any method before updating the values. In this situation, no problem has been detected, but undoubtedly it is a completely unadvised practice that can open new vectors of attacks.

```
assert self.ongoingLoans[msg.sender] < self.maxAllowedLoans, "Max number of loans for borrower already reached"
assert _amount >= self.minLoanAmount, "Loan amount is less than the min loan amount"
assert _amount <= self.maxLoanAmount, "Loan amount is more than the max loan amount"

newLoanId: uint256 = self.loansCore.addLoan(
    msg.sender,
    _amount,
    _interest,
    _maturity,
    _collaterals
)

for collateral in  collaterals:
    ICollateral(collateral.contractAddress).transferFrom(msg.sender, self, collateral.tokenId)

    self.loansCore.addCollateralToLoan(msg.sender, collateral, newLoanId)

    self.loansCore.updateCollaterals(collateral, False)

self.ongoingLoans[msg.sender] += 1
```

The risk of this vulnerability has been reduced from critical to informative, because it currently does not have a clear exploit path.

Recommendations

It is **essential to always make the state changes in the storage before making transfers or calls to external contracts**, in addition to implementing the necessary measures to avoid the reentrancy behavior.

References

- https://eips.ethereum.org/EIPS/eip-777

Source Code References

- contracts/Loans.vy#L324-329

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/47

Lack of Documentation

Identifier	Category	Risk	State
ZH-12	Bad Coding Practices	**Informative**	**Assumed**

The **Zharta** project does not contain technical documentation of any sort, nor execution flow diagrams, class diagrams or code properly commented. This is a bad practice that complicates understanding the project and makes it difficult for the team of auditors to analyze the functionalities, since there is no technical documentation to check if the current implementation meets the needs and purpose of the project.

Having updated and accurate documentation of the project is an essential aspect for open-source projects, in fact it is closely related with the adoption and contribution of the project by the community.

Recommendations

Documentation is an integral part of the Secure Software Development Lifecycle (SSDLC), and it helps to improve the quality of the project, so it is recommended to add the code's documentation with the according descriptions of the functionalities, classes and public methods.

References

- https://snyk.io/learn/secure-sdlc/
- https://www.freecodecamp.org/news/why-documentation-matters-and-why-you-should-include-it-in-your-code-41ef62dd5c2f

Outdated Compiler

Identifier	Category	Risk	State
ZH-13	Outdated Software	**Informative**	**Fixed**

Vyper frequently launches new versions of the compiler. Using an outdated version of the compiler can be problematic, especially if there are errors that have been made public or known vulnerabilities that affect this version. It is always of good policy to use the most up to date version of the compiler.

We have detected that the audited contract uses the following version of Vyper `0.3.2`:

The `0.3.3` version of compiler it is a bugfix release, and, in this case, it has important bug fixes in the storage allocation mechanism for dynamic arrays.

Recommendations

- Use the latest and more stable compiler version, at any possible time.

References

- https://github.com/vyperlang/vyper/releases

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/48

Unused Libraries

Identifier	Category	Risk	State
ZH-14	Unused Code	**Informative**	**Fixed**

The existence of the OpenZeppelin libraries in the project has been detected, however these are not necessary since Vyper is used in the development.

Currently the OpenZeppelin libraries are not compatible with Vyper, they are developed exclusively for Solidity and therefore the contracts cannot inherit these functionalities, since Vyper language has no code reusability mechanism (due to lack of inheritance).

This issue does not imply a vulnerability, but it is always advisable to leave the project as clean and organized as possible to facilitate the understanding of future users.

Recommendations

- Remove unused third-party libraries.

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/49

GAS Optimization

Identifier	Category	Risk	State
ZH-15	Bad Coding Practices	**Informative**	**Fixed**

Software optimization is the process of modifying a software system to make an aspect of it work more efficiently or use less resources. This premise must be applied to smart contracts as well, so that they execute faster or in order to save GAS.

On Ethereum blockchain, GAS is an execution fee which is used to compensate miners for the computational resources required to power smart contracts. If the network usage is increasing, so will the value of GAS optimization.

These are some of the requirements that must be met to reduce GAS consumption:

- Short-circuiting.
- Remove redundant or dead code.
- Delete unnecessary libraries.
- Explicit function visibility.
- Use of proper data types.
- Use hard-coded CONSTANT instead of state variables.
- Avoid expensive operations in a loop.
- Pay special attention to arithmetical operations and comparisons.

Dead code

In programming, a part of the source code that is never used is known as dead code. The execution of this type of code consumes more GAS during deployment in something that is not necessary.

The following methods or events of the contract are not used during the execution of the said contract, so it would be convenient to either remove them or to put them at use.

This behavior has been observed in:

LendingPoolCore.vy:

- `Compound` event
  - LendingPoolCore.vy#L26
- `_fundsAreAllowed` method
  - LendingPoolCore.vy#L55

Optimize require messages

Ethereum Virtual Machine operates under a 32-byte word memory model where an additional gas cost is paid by any operation that expands the memory that is in use.

Therefore, exceeding error messages of this length means increasing the number of slots necessary to process the require, reducing the error messages to 32 bytes or less would lead to saving gas.

This behavior has been observed in:

LoansCore.vy:

- contracts/LoansCore.vy#L129-130
- contracts/LoansCore.vy#L139-140
- contracts/LoansCore.vy#L301-302
- contracts/LoansCore.vy#L309-310
- contracts/LoansCore.vy#L317
- contracts/LoansCore.vy#L330
- contracts/LoansCore.vy#L358-359
- contracts/LoansCore.vy#L368-369
- contracts/LoansCore.vy#L376-378
- contracts/LoansCore.vy#L381
- contracts/LoansCore.vy#L388-389
- contracts/LoansCore.vy#L396-397
- contracts/LoansCore.vy#L404-405
- contracts/LoansCore.vy#L412
- contracts/LoansCore.vy#L420
- contracts/LoansCore.vy#L428
- contracts/LoansCore.vy#L436

LendingPoolCore.vy:

- contracts/LendingPoolCore.vy#L104
- contracts/LendingPoolCore.vy#L115
- LendingPoolCore.vy#L155-157
- contracts/LendingPoolCore.vy#L189-190
- contracts/LendingPoolCore.vy#L207-209

LendingPoolPeripheral.vy:

- contracts/LendingPoolPeripheral.vy#L150
- contracts/LendingPoolPeripheral.vy#L159
- contracts/LendingPoolPeripheral.vy#L168
- contracts/LendingPoolPeripheral.vy#L177-178
- contracts/LendingPoolPeripheral.vy#L193
- contracts/LendingPoolPeripheral.vy#L201
- contracts/LendingPoolPeripheral.vy#L213-214
- contracts/LendingPoolPeripheral.vy#L223
- contracts/LendingPoolPeripheral.vy#L225
- contracts/LendingPoolPeripheral.vy#L232
- contracts/LendingPoolPeripheral.vy#L244
- contracts/LendingPoolPeripheral.vy#L246
- contracts/LendingPoolPeripheral.vy#L247
- contracts/LendingPoolPeripheral.vy#L291-295
- contracts/LendingPoolPeripheral.vy#L312

- contracts/LendingPoolPeripheral.vy#L314

Loans.vy:

- contracts/Loans.vy#L166
- contracts/Loans.vy#L175
- contracts/Loans.vy#L184
- contracts/Loans.vy#L193-194
- contracts/Loans.vy#L205-206
- contracts/Loans.vy#L215-216
- contracts/Loans.vy#L225-226
- contracts/Loans.vy#L235-236
- contracts/Loans.vy#L245
- contracts/Loans.vy#L254-255
- contracts/Loans.vy#L264-265
- contracts/Loans.vy#L304-305
- contracts/Loans.vy#L307-314
- contracts/Loans.vy#L340-347
- contracts/Loans.vy#L362-363
- contracts/Loans.vy#L382-384
- contracts/Loans.vy#L390-392
- contracts/Loans.vy#L423-425
- contracts/Loans.vy#L451

Execution Cost

The `receiveFunds` method of the `LendingPoolPeripheral` contract executes an `allowance` with the objective of knowing if the user has enough tokens before carrying out the `transferFrom`.

However, the `transferFrom` method already makes this verification internally, so the check through allowance is unnecessary, eliminating this call would mean saving GAS in the execution of the function.

```
@external
def receiveFunds(_borrower: address, _amount: uint256, _rewardsAmount: uint256) -> bool:
    # _amount and _rewardsAmount should be passed in wei

    assert msg.sender == self.loansContract, "The sender address is not the loans contract address"
    assert self._fundsAreAllowed(_borrower, self, _amount + _rewardsAmount), "Insufficient funds allowed to be transfered"
    assert _amount + _rewardsAmount > 0, "The sent value should be higher than 0"

    rewardsProtocol: uint256 = _rewardsAmount * self.protocolFeesShare / 10000
    rewardsPool: uint256 = _rewardsAmount - rewardsProtocol

    if not IERC20Token(self.erc20TokenContract).transferFrom(_borrower, self, _amount + _rewardsAmount):
        raise "Error transferring funds from borrower"

    if not IERC20Token(self.erc20TokenContract).transfer(self.protocolWallet, rewardsProtocol):
        raise "Error transferring funds to protocol wallet"
```

```
@view
@internal
def _fundsAreAllowed(_owner: address, _spender: address, _amount: uint256) -> bool:
    amountAllowed: uint256 = IERC20Token(self.erc20TokenContract).allowance(_owner, _spender)
    return _amount <= amountAllowed
```

This behavior has been observed in:

- contracts/LendingPoolPeripheral.vy#L313
- contracts/LendingPoolPeripheral.vy#L319

It is possible and advised to treat a boolean variable without having to directly compare it with `== true`. Despite having compiler optimizations turned on, the action of eliminating that condition will save gas.

```
@external
def addCollateralToWhitelist(_address: address) -> bool:
    assert msg.sender == self.owner, "Only the contract owner can add collateral addresses to the whitelist"
    assert _address.is_contract == True, "The _address sent does not have a contract deployed"
```

This behavior has been observed in:

- contracts/Loans.vy#L194

In the `_maxFundsInvestable` and `_poolHasFundsToInvest` methods, a double casting is performed, to obtain the value in `uint256`, it is recommended to study the possibility of changing the type in order to eliminate the double casting.

This behavior has been observed in:

- contracts/LendingPoolPeripheral.vy#L86-87
- contracts/LendingPoolPeripheral.vy#L97-98

It has been found that it is possible to optimize the `addCollateralToWhitelist` method of the `Loans` contract in order to avoid going through the entire `whitelistedCollateralsAddresses` array. It is recommended to use the `whitelistedCollaterals` map with the objective of veryfing if it was not previously added to the whitelist (`false`), and thus avoid going through the entire array.

```
@external
def addCollateralToWhitelist(_address: address) -> bool:
    assert msg.sender == self.owner, "Only the contract owner can add collateral addresses to the whitelist"
    assert _address.is_contract == True, "The _address sent does not have a contract deployed"

    self.whitelistedCollaterals[_address] = True
    if _address not in self.whitelistedCollateralsAddresses:
        self.whitelistedCollateralsAddresses.append(_address)

    return True
```

This behavior has been observed in:

- contracts/Loans.vy#L197

Unnecessary returns

The existence of unnecessary returns has been detected. The value established by the user in the call is being returned, reading it from storage instead of memory, it is recommended to directly eliminate it or return the value of the received argument avoiding the query to storage.

This behavior has been observed in:

- contracts/LoansCore.vy#L134

- contracts/LoansCore.vy#L144
- contracts/LendingPoolPeripheral.vy#L145
- contracts/LendingPoolPeripheral.vy#L172
- contracts/LendingPoolPeripheral.vy#L196
- contracts/LendingPoolPeripheral.vy#L208
- contracts/LendingPoolPeripheral.vy#L218
- contracts/Loans.vy#L170
- contracts/Loans.vy#L179
- contracts/Loans.vy#L188
- contracts/Loans.vy#L220
- contracts/Loans.vy#L230
- contracts/Loans.vy#L240
- contracts/Loans.vy#L249
- contracts/Loans.vy#L259
- contracts/Loans.vy#L270

Additionally, certain methods of the `LendingPoolCore` and `Loans` contracts always return `True`. If a method always returns a constant value, consider removing the return from that method to save gas.

This behavior has been observed in:

- contracts/LendingPoolCore.vy#L148
- contracts/LendingPoolCore.vy#L182
- contracts/LendingPoolCore.vy#L200
- contracts/LendingPoolCore.vy#L215
- contracts/Loans.vy#L200
- contracts/Loans.vy#L210

Fixes Review

This issue has been addressed in the following pull request. However, only the changes referring to the remmediations of the following issue have been reviewed:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/50

Unsecured Ownership Transfer

Identifier	Category	Risk	State
ZH-16	Bad Coding Practices	**Informative**	**Fixed**

The modification process of an owner is a delicate process, since the governance of our contract and therefore of the project may be at risk, for this reason it is recommended to adjust the owner's modification logic, to a logic that allows to verify that the new owner is in fact valid and does exist.

Following, we can see a standard implementation of the owner's modification where a new owner is proposed first, the owner accepts the proposal and, in this way, we make sure that there are no errors when writing the address of the new owner.

```
function proposeOwner(address _proposedOwner) public onlyOwner
{
        require(msg.sender != _proposedOwner, ERROR_CALLER_ALREADY_OWNER);
        proposedOwner = _proposedOwner;
}


function claimOwnership() public
{
        require(msg.sender == proposedOwner, ERROR_NOT_PROPOSED_OWNER);
        emit OwnershipTransferred(_owner, proposedOwner);
        _owner = proposedOwner;
        proposedOwner = address(0);
}
```

Note that the example is written in solidity
Aditionally, in the affected code, the address of the new `owner` is not checked to be anything other than `address(0)`, which allows the owner to resign.

Source Code References

- contracts/LendingPoolCore.vy#L103
- contracts/LoansCore.vy#L128
- contracts/LendingPoolPeripheral.vy#L140
- contracts/Loans.vy#L165

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/52

Lack of Event Index

Identifier	Category	Risk	State
ZH-17	Bad Coding Practices	**Informative**	**Fixed**

Event indexing of Smart contracts can be used to filter during the querying of events. This can be very useful when making dApps or in the off-chain processing of the events in our contract, as it allows filtering by specific addresses, making it much easier for developers to query the results of invocations.

Recommendations

It could be convenient to review the `Loans`, `LendingPoolPeripheral` and `LendingPoolCore` contracts to ensure that all the events have the necessary indexes for the correct functioning of the possible DApps. Addresses are usually the best argument to filter an event.

References

- https://vyper.readthedocs.io/en/stable/event-logging.html

Source Code References

- contracts/LendingPoolCore.vy#L27
- contracts/LendingPoolPeripheral.vy#L28
- contracts/LendingPoolPeripheral.vy#L33
- contracts/LendingPoolPeripheral.vy#L38
- contracts/LendingPoolPeripheral.vy#L43
- contracts/LendingPoolPeripheral.vy#L48
- contracts/Loans.vy#L52
- contracts/Loans.vy#L57
- contracts/Loans.vy#L62
- contracts/Loans.vy#L67
- contracts/Loans.vy#L73
- contracts/Loans.vy#L78
- contracts/Loans.vy#L84
- contracts/Loans.vy#L89

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/55

Emit Events on State Changes

Identifier	Category	Risk	State
ZH-18	Bad Coding Practices	**Informative**	**Fixed**

It is a good practice to emit events when there are significant changes in the states of the contract that can affect the result of its execution by the users.

The changes in any administrative method should emit events so that the potential actors monitoring the blockchain; such as DApps, automated processes and users, can be notified of these significant state changes.

Following find a list of the main methods that make significant changes and should be properly notified:

- contracts/LendingPoolPeripheral.vy#L192
- contracts/LendingPoolPeripheral.vy#L200
- contracts/LendingPoolPeripheral.vy#L212
- contracts/LendingPoolPeripheral.vy#L212
- contracts/LendingPoolPeripheral.vy#L231
- contracts/Loans.vy#L165-270
- contracts/LendingPoolCore.vy#L103

Recommendations

- Consider adding events to the changes that an owner can make to the contract.

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/56

Contracts Management Risks

Identifier	Category	Risk	State
ZH-19	Design Weaknesses	**Informative**	**Fixed**

The logic design of the **Zharta** contracts imply a few minor risks that should be reviewed and considered for their improvement. Zharta team maintains some centralized parts that imply trust in the project, **which are indeed necessary**. A few of the administrative functionalities are under the control of the project, such as: `setLoansCoreAddress`, `setLendingPoolAddress`, etc.

The fact that the administrative changes are not limited to the current state of the contract can affect the economics of both the project and the user. For example, the owner has the possibility to change the address of `loansCore` at any time, being able to prevent the user from paying his loan and receiving his collateral. This type of action should only be allowed if there are no active loans.

In order to promote decentralization, it would be advisable to improve the logic of the contract, in order to reduce the chances of rogue pool type of attacks and increase confidence in the project.

Recommendations

- Use a `TimeLock` type governance system.
- Restrict certain administrative actions to the status of the contract.
- Emit events when making changes to administrative methods, as recommended in the **"Emit Events on State Changes"** issue.

Fixes Review

This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/53

Wrong Fallback Implementation

Identifier	Category	Risk	State
ZH-20	Code Optimization	**Informative**	**Fixed**

All of the **Zharta** contracts that were analyzed, incorrectly implement the `fallback` method to try to prevent the reception of ether.

```
@external
@payable
def __default__():
    if msg.value > 0:
        send(msg.sender, msg.value)
```

The `__default__` method returns the `ethers` sent to the sender, when if you don't want to receive `ether`, it is enough not to implement said method.

Additionally, the current implementation implies that the contract responds positively to the call of any non-existing method, which may not match the needs of the project and may not revert the execution when expected.

Recommendations

- Eliminate the `fallback` method if it is not strictly necessary for the logic of the project.

Source Code References

- contracts/LendingPoolCore.vy#L220
- contracts/LendingPoolPeripheral.vy#L338
- contracts/Loans.vy#L470
- contracts/LoansCore.vy#L444

Fixes Review

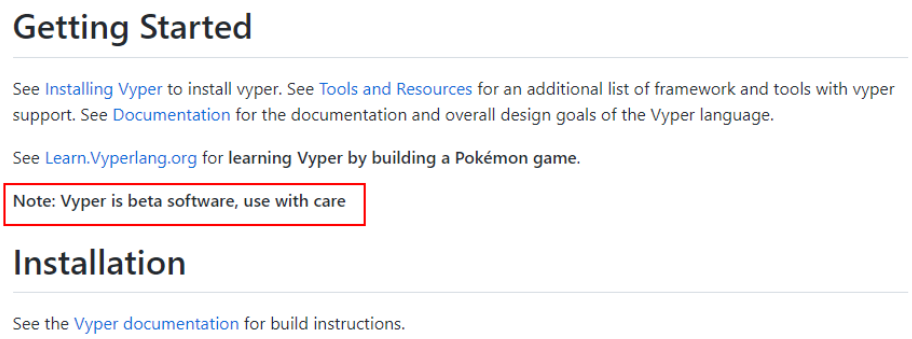
This issue has been addressed in the following pull request:

- https://gitlab.com/z106/smart-contracts-eth/-/merge_requests/54

Beta Compiler

Identifier	Category	Risk	State
ZH-21	Bad Coding Practices	**Informative**	**Assumed**

The current smart contract has been developed using a programming language which is in "beta" state for development. Even though the security is one of the main goals of the Vyper language, it is still in a development state and the presence of errors is more frequent than in other more tested languages for smart contracts such as Solidity.



Getting Started

See Installing Vyper to install vyper. See Tools and Resources for an additional list of framework and tools with vyper support. See Documentation for the documentation and overall design goals of the Vyper language.

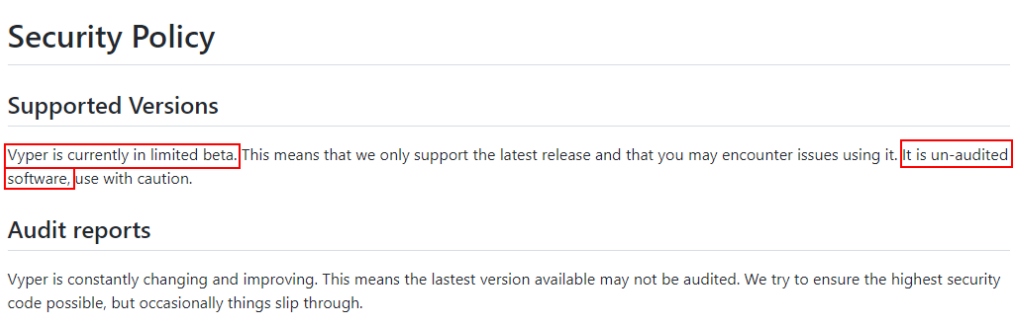
See Learn.Vyperlang.org for **learning Vyper by building a Pokémon game**.

Note: Vyper is beta software, use with care

Installation

See the Vyper documentation for build instructions.

In fact, as indicated in the official repository, Vyper is currently in beta stage, this means that they only support the latest release and that you may encounter issues using it. It is unaudited software, so it is recommended to use it with caution.



Security Policy

Supported Versions

Vyper is currently in limited beta. This means that we only support the latest release and that you may encounter issues using it. It is un-audited software, use with caution.

Audit reports

Vyper is constantly changing and improving. This means the lastest version available may not be audited. We try to ensure the highest security code possible, but occasionally things slip through.

References

- https://github.com/vyperlang/vyper/blob/master/README.md#getting-started
- https://github.com/vyperlang/vyper/blob/master/SECURITY.md#supported-versions

Annexes

Annex A – Vulnerabilities Severity

Red4Sec determines the vulnerabilities severity in the following levels of risk according to the impact level defined by CVSS v3 (Common Vulnerability Scoring System) by the National Institute of Standards and Technology (NIST):

Vulnerability Severity

The risk classification has been made on the following 5 value scale:

Severity	Description
Critical	Vulnerabilities that possess the highest impact over the systems, services and/or sensitive information. The existence of these vulnerabilities is dangerous and should be fixed as soon as possible.
High	Vulnerabilities that could compromise severely compromise the service or the information it manages even if the vulnerability requires expertise to be exploited.
Medium	Vulnerabilities that on their own can have a limited impact and/or that combined with other vulnerabilities could have a greater impact.
Low	These vulnerabilities do not suppose a real risk for the systems. Also includes vulnerabilities which are extremely hard to exploit or whose impact on the service is low.
Informative	It covers various characteristics, information or behaviours that can be considered as inappropriate, without being considered as vulnerabilities by themselves.

Invest in Security, invest in your future