CAIRO SECURITY CLAN

CADDY FINANCE

SECURITY ASSESSMENT REPORT

JULY 2025

Prepared for

CADDY FINANCE

Contents

1 About Cairo Security Clan

Cairo Security Clan is a leading force in the realm of blockchain security, dedicated to fortifying the foundations of the digital age. As pioneers in the field, we specialize in conducting meticulous smart contract security audits, ensuring the integrity and reliability of decentralized applications built on blockchain technology.

At Cairo Security Clan, we boast a multidisciplinary team of seasoned professionals proficient in blockchain security, cryptography, and software engineering. With a firm commitment to excellence, our experts delve into every aspect of the Web3 ecosystem, from foundational layer protocols to application-layer development. Our comprehensive suite of services encompasses smart contract audits, formal verification, and real-time monitoring, offering unparalleled protection against potential vulnerabilities.

Our team comprises industry veterans and scholars with extensive academic backgrounds and practical experience. Armed with advanced methodologies and cutting-edge tools, we scrutinize and analyze complex smart contracts with precision and rigor. Our track record speaks volumes, with a plethora of published research papers and citations, demonstrating our unwavering dedication to advancing the field of blockchain security.

At Cairo Security Clan, we prioritize collaboration and transparency, fostering meaningful partnerships with our clients. We believe in a customer-oriented approach, engaging stakeholders at every stage of the auditing process. By maintaining open lines of communication and soliciting client feedback, we ensure that our solutions are tailored to meet the unique needs and objectives of each project.

Beyond our core services, Cairo Security Clan is committed to driving innovation and shaping the future of blockchain technology. As active contributors to the ecosystem, we participate in the development of emerging technologies such as Starknet, leveraging our expertise to build robust infrastructure and tools. Through strategic guidance and support, we empower our partners to navigate the complexities of the blockchain landscape with confidence and clarity.

In summary, Cairo Security Clan stands at the forefront of blockchain security, blending technical prowess with a client-centric ethos to deliver unparalleled protection and peace of mind in an ever-evolving digital landscape. Join us in safeguarding the future of decentralized finance and digital assets with confidence and conviction.

2 Disclaimer

Disclaimer Limitations of this Audit:

This report is based solely on the materials and documentation provided by you to Cairo Security Clan for the specific purpose of conducting the security review outlined in the Summary of Audit and Scope. The findings presented here may not be exhaustive and may not identify all potential vulnerabilities. Cairo Security Clan provides this review and report on an ”as-is” and ”as-available” basis. You acknowledge that your use of this report, including any associated services, products, protocols, platforms, content, and materials, occurs entirely at your own risk.

Inherent Risks of Blockchain Technology:

Blockchain technology remains in its developmental stage and is inherently susceptible to unknown risks and vulnerabilities. This review is specifically focused on the smart contract code and does not extend to the compiler layer, programming language elements beyond the reviewed code, or other potential security risks outside the code itself.

Report Purpose and Reliance:

This report should not be construed as an endorsement of any specific project or team, nor does it guarantee the absolute security of the audited smart contracts. No third party should rely on this report for any purpose, including making investment or purchasing decisions.

Liability Disclaimer:

To the fullest extent permitted by law, Cairo Security Clan disclaims all liability associated with this report, its contents, and any related services and products arising from your use. This includes, but is not limited to, implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

Third-Party Products and Services:

Cairo Security Clan does not warrant, endorse, guarantee, or assume responsibility for any products or services advertised by third parties within this report, nor for any open-source or third-party software, code, libraries, materials, or information linked to, referenced by, or accessible through this report, its content, and related services and products. This includes any hyperlinked websites, websites or applications appearing on advertisements, and Cairo Security Clan will not be responsible for monitoring any transactions between you and third-party providers. It is recommended that you exercise due diligence and caution when considering any third-party products or services, just as you would with any purchase or service through any medium.

Disclaimer of Advice:

FOR THE AVOIDANCE OF DOUBT, THIS REPORT, ITS CONTENT, ACCESS, AND/OR USE, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHOULD NOT BE CONSIDERED OR RELIED UPON AS FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER PROFESSIONAL ADVICE.

3 Executive Summary

This document presents the security review performed by Cairo Security Clan on the Caddy Finance developed by Caddy Finance.

Caddy Finance provides a platform that enables users to earn yields on their Bitcoin holdings with multiple risk profiles (High, Medium, or Low) through a single transaction deposit. The platform operates by lending received WBTC deposits to VESU, obtaining USDC at a 50% loan-to-value ratio, which is then traded centrally through their trader wallet using derivatives strategies optimized by trading desks, with trading revenue subsequently deposited into the yield pool that distributes returns over the cycle back to users' Bitcoin vaults.

In summary, the system is highly connected to the centralized wallet. Any suspicious activity on the trader's wallet can result in the loss of the user's collateral. We suggested that the Caddy Finance team develop on-chain delta-neutral strategies for this project instead of a centralized trader wallet. This audit reviewed the vault's LP accounting, collateral operations, and interaction with external protocols; however, centralization still causes risks.

The audit was performed using

- manual analysis of the codebase,
- automated analysis tools,
- simulation of the smart contract,
- analysis of edge test cases

There are 15 points of attention, where 4 are classified as Critical, 3 as High, 3 as Medium, 3 as Low, 0 as Informational, 2 as Best Practices, and 0 as Undetermined. The issues are summarized in Fig. 1.

This document is organized as follows. Section 1 About Cairo Security Clan. Section 2 Disclaimer. Section 3 Executive Summary. Section 4 Summary of Audit. Section 5 Risk Classification. Section 6 Issues by Severity Levels. Section 7 Test Evaluation.

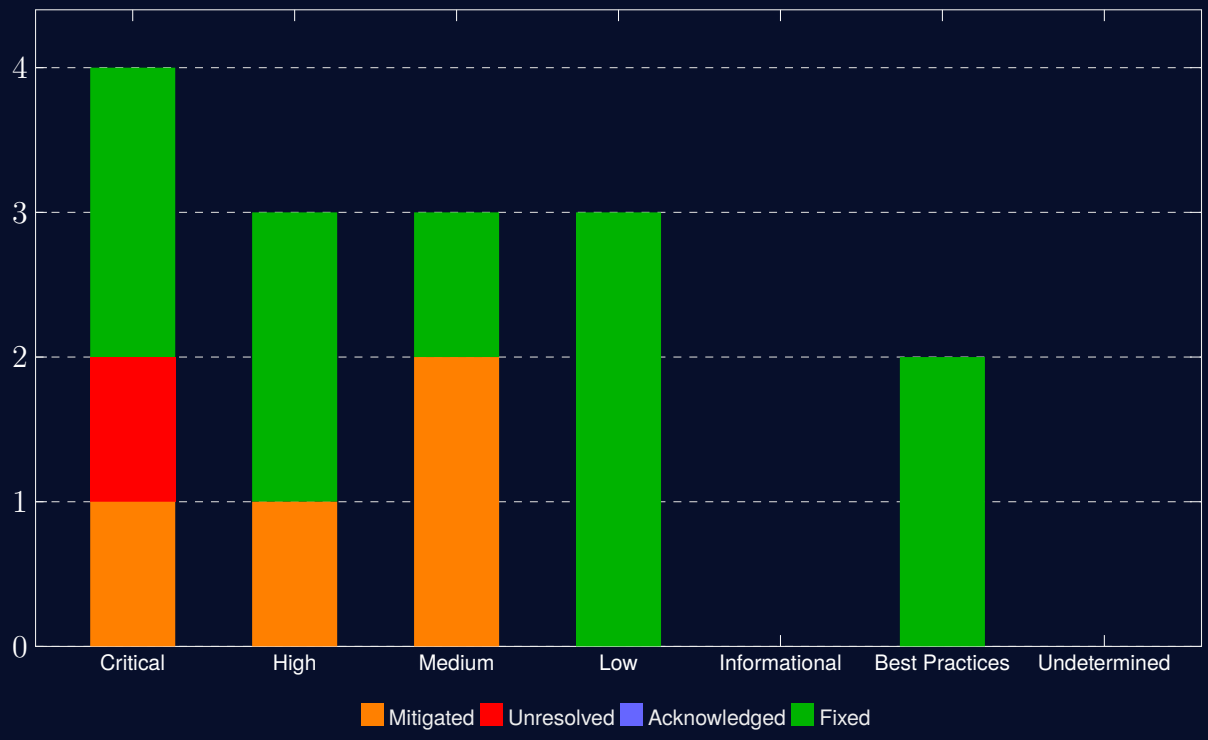

Fig 1: Distribution of issues: Critical (4), **High** (3), **Medium** (3), **Low** (3), **Informational** (0), **Best Practices** (2), **Undetermined** (0). **Distribution of status: Fixed** (10), **Acknowledged** (0), **Mitigated** (4), **Unresolved** (1).

4 Summary of Audit

Audit Type	Security Review
Cairo Version	2.11.4
Final Report	08/10/2025
Repository	caddyfinance/Starknet-Contracts
Initial Commit Hash	bf993dcb55338608b566c0c0b94ecd7219b97b1d
Documentation	Not Available
Test Suite Assessment	Low
Centralization Level	Highly Centralized

4.1 Scoped Files

	Contracts
1	/src/lib.cairo
2	/src/bitcoin_vault.cairo
3	/src/interfaces.cairo
4	/src/utils.cairo
5	/src/yield_pool.cairo
6	/src/interfaces/IVesu.cairo
7	/src/interfaces/bitcoin_vault_interface.cairo
8	/src/interfaces/lendcomp.cairo
9	/src/interfaces/lent_debt_token_interface.cairo
10	/src/interfaces/oracle.cairo
11	/src/interfaces/yield_pool_interface.cairo
12	/src/utils/CustomLPToken.cairo
13	/src/utils/ERC20Helper.cairo
14	/src/utils/constants.cairo
15	/src/utils/math.cairo
16	/src/utils/pow.cairo
17	/src/utils/safe_decimal_math.cairo
18	/src/utils/types.cairo

4.2 Issues

	Findings	Severity	Update
1	The `end_cycle(...)` function may fail if trader has negative P&L	Critical	Mitigated
2	Emergency withdrawal can fail due to insufficient funds	Critical	Fixed
3	Transfer functions balance effects are miscalculated	Critical	Fixed
4	Centralized trader wallet	Critical	Unresolved
5	Withdraw collateral calculation miscalculates collateral	High	Fixed
6	Emergency withdrawal values can be changed by owner	High	Fixed
7	Processing cycle transitions can reach max state update limit	High	Mitigated
8	`deposit_yield(...)` is not validating `cycle_id`	Medium	Mitigated
9	Emergency withdrawal wait period has no upper limit	Medium	Fixed
10	Ownership can be transferred to non deployed contract	Medium	Mitigated
11	No validation for `new_duration` for cycles	Low	Fixed
12	Emergency wallets can be identical addresses	Low	Fixed
13	Emergency withdrawal logic has no cancel mechanism	Low	Fixed
14	Duplicated start cycle logic	Best Practices	Fixed
15	Initiating emergency is not pausing contract	Best Practices	Fixed

5 Risk Classification

The risk rating methodology used by Cairo Security Clan follows the principles established by the CVSS risk rating methodology. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely an attacker will uncover and exploit the finding. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to Motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Likelihood		
		High	**Medium**	**Low**
Impact	**High**	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Info/Best Practices

To address issues that do not fit a High/Medium/Low severity, Cairo Security Clan also uses three more finding severities: **Informational**, **Best Practices** and **Gas**

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to formally pass to the client;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Gas** findings are used when some piece of code uses more gas than it should be or have some functions that can be removed to save gas.

6 Issues by Severity Levels

6.1 Critical

6.1.1 The `end_cycle(...)` function may fail if trader has negative P&L

File(s): /src/bitcoin_vault.cairo

Description: Protocol centralized math relies on trader wallets P&L. Cycles can be finalized by calling the `end_cycle(...)` function. Ending the cycle means paying the borrowed USDCS to the lender. Generally, the strategy is depositing all WBTCs to Vesu and borrowing USDC with a 50% LTV ratio. So, as an example, if a user deposits 100 USDC worth of WBTC. 50 USDC will be borrowed, then the protocol deducts `PROTOCOL_FEE` (which is set as `200` in bps and equals 2%) from the borrowed amount and sends the remaining USDCs to the trader's wallet. Then, the protocol optimistically assumes the centralized trader wallet will not be compromised and returns the profits to the vault contract. However, ending the cycle means paying all debt. So, in case the trader's wallet has a negative P&L, it is even lower than the platform fee. Vault won't be able to cover the debt cost, and end_cycle will fail because `_repay_all_debt(...)` function tries to pay all debt, but vault doesn't have enough USDC to cover that. These causes can not be end.

```
fn _repay_all_debt(ref self: ContractState) {
    // Create Vesu struct for this contract
    let vesu_settings = VesuStruct {
        singleton: IStonDispatcher { contract_address: self.vesu_singleton.read() },
        pool_id: self.vesu_pool_id.read(),
        debt: self.usdc.read(),
        col: self.wbtc.read(),
        oracle: self.vesu_oracle.read(),
    };
    // Validate Vesu settings
    vesu_settings.assert_valid();
    let this = get_contract_address();
    let current_debt = vesu_settings.borrow_amount(self.usdc.read(), this);
    if current_debt > 0 {
        // Repay all debt to Vesu
        let repaid_amount = vesu_settings.repay(self.usdc.read(), current_debt);
        // Emit event for debt repayment
        self
            .emit(
                Borrowed {
                    user: this, amount: repaid_amount, cycle_id: self.current_cycle.read(),
                },
            );
    }
}
```

Recommendation(s): Consider creating delta-neutral and on-chain trading strategies.

Status: Mitigated

Update from the client: Fixed in commit f22f213

6.1.2 Emergency withdrawal can fail due to insufficient funds

File(s): /src/bitcoin_vault.cairo

Description: Emergency withdrawals fail if the trader's wallet is compromised and funds are stuck, or if the trader made a negative P&L that can't cover debt.

```
fn emergency_withdraw(ref self: ContractState, token_to_withdraw: ContractAddress) {
      // ...
      // @audit-issue Can fail if not enough USDC, causes colleterals to stuck.
      self._repay_all_debt();
      // ...
 }
```

Recommendation(s): Ensure funds are transferred to vault if emergency withdrawal started. With current design it can still be problematic, because trader wallet is trusted entity. Highly recommend to create another on-chain strategy for fund management.

Status: Fixed

Update from the client: Fixed in commit d4fb62f

6.1.3 Transfer functions balance effects are miscalculated

File(s): /src/utils/CustomLPToken.cairo

Description: CustomLPToken contract is an implementation for LP balances and accounting. Instead of using existing ERC20 components, a custom implementation was used here. Transfer functions (`transfer` & `transfer_from`) are not calculation new balances correctly.

```
fn transfer(ref self: ContractState, recipient: ContractAddress, amount: u256) -> bool {
    // @audit-issue Self-transfer increases balance.
    // ...
    let sender_prev_balance = self.balances.entry(sender).read();
    let recipient_prev_balance = self.balances.entry(recipient).read();
    assert(sender_prev_balance >= amount, 'Insufficient amount');
    self.balances.entry(sender).write(sender_prev_balance - amount);
    self.balances.entry(recipient).write(recipient_prev_balance + amount);
    // ...
}
fn transfer_from(
    ref self: ContractState,
    sender: ContractAddress,
    recipient: ContractAddress,
    amount: u256,
) -> bool {
    // @audit-issue Critical. Self-transfer increases balance.
    // ...
    let sender_balance = self.balances.entry(sender).read();
    let recipient_balance = self.balances.entry(recipient).read();
    assert(amount <= spender_allowance, 'amount exceeds allowance');
    assert(amount <= sender_balance, 'amount exceeds balance');
    self.allowances.entry((sender, spender)).write(spender_allowance - amount);
    self.balances.entry(sender).write(sender_balance - amount);
    self.balances.entry(recipient).write(recipient_balance + amount);
    // ...
}
```

`recipient_balance` and `sender_balance` are read from storage at the same time. Then new balances are written to storage. However, if this transfer occurs in the same accounts context (`sender == recipient`), then the actual balance will be higher.

Recommendation(s): Consider using existing trusted components like OpenZeppelins.

Status: Fixed

Update from the client: Fixed in commit eddf637

6.1.4 Centralized trader wallet

File(s): *

Description: The system is designed to generate trader wallets' P&L. Those profits are made by indirect use of collateral. The system is designed as follows;

- User deposits WBTC
- Vault deposits WBTC to Vesu and borrows USDC for 50% of the collateral's USD value.
- Vault sends borrowed USDCs to the trader's wallet. Which is the centralized account.

Then, the trader wallet makes trades or uses any other strategies to generate P&L using those collaterals. However, the platform relies on the trader's wallet security. That wallet is not a smart contract, and strategies are not included in the scope. In cases like a compromised trader wallet, wrong trades, etc., all user funds will be at risk.

Recommendation(s):

Status: Unresolved

Update from the client:

6.2 High

6.2.1 Withdraw collateral calculation miscalculates collateral

File(s): /src/bitcoin_vault.cairo

Description: Collateral deposited to Vesu will be withdrawn while the cycle is ending with the `end_cycle(...)` function. However, it withdraws only the initial collateral. In case actual collateral is different than `cycle_total_collateral`, some collaterals can remain in Vesu, or if any liquidation happens, this function reverts directly.

```
fn _withdraw_cycle_collateral_from_vesu(ref self: ContractState, cycle_id: u64) {
    // Create Vesu struct for this contract
    let vesu_settings = VesuStruct {
        singleton: IStonDispatcher { contract_address: self.vesu_singleton.read() },
        pool_id: self.vesu_pool_id.read(),
        debt: self.usdc.read(),
        col: self.wbtc.read(),
        oracle: self.vesu_oracle.read(),
    };
    // Validate Vesu settings
    vesu_settings.assert_valid();
    let this = get_contract_address();
    let total_wbtc_collateral_for_cycle = self.cycle_total_collateral.read(cycle_id);
    if total_wbtc_collateral_for_cycle > 0 {
        // Withdraw all WBTC collateral from Vesu
        // @audit-issue Tries to withdraw total collateral deposits.
        let withdrawn_amount = vesu_settings
            .withdraw(self.wbtc.read(), total_wbtc_collateral_for_cycle);
        // Emit event for collateral withdrawal
        self.emit(CollateralReturned { amount: withdrawn_amount, cycle_id });
    }
}
```

Recommendation(s): Consider liquidation and interest generated during cycle.

Status: Fixed

Update from the client: Fixed in commit f6e140c

6.2.2 Emergency withdrawal values can be changed by owner

File(s): /src/bitcoin_vault.cairo

Description: An emergency withdrawal request has a time delay. Once an emergency is requested, withdrawal can happen after some delay. However, the contract owner can change that delay at any time, and it will take effect immediately. This allows the owner to immediately withdraw without waiting for any delay.

```
fn set_emergency_withdrawal_request_time(ref self: ContractState, request_time: u64) {
    self.ownable.assert_only_owner();
    self.emergency_withdrawal_request_time.write(request_time);
}
fn set_emergency_withdrawal_wait_period(ref self: ContractState, new_wait_period: u64) {
    self.ownable.assert_only_owner();
    self.emergency_withdrawal_wait_period.write(new_wait_period);
}
```

Recommendation(s): Consider adding a minimum limit before setting emergency delay.

Status: Fixed

Update from the client: Fixed in commit f92ab4c

6.2.3 Processing cycle transitions can reach max state update limit

File(s): /src/bitcoin_vault.cairo

Description: The `_process_cycle_transitions(...)` function is called at the end of `end_cycle(...)`, which transits not withdrawn balances to the next cycle. That function has a loop through all participants from the last cycle. However, Starknet has a 2000 storage change limit per transaction. On this loop, just in the `_process_cycle_transitions(...)` function, a minimum of 6 storage changes happen. That will be increased with another function called inside the loop. That function can easily exceed that limit, which causes cycles to be unable to be finalized.

```
        fn _process_cycle_transitions(ref self: ContractState) {
            // ...
            // @audit-issue Loop through all participants
            for i in 0..participants_count {
                // @audit-issue Multiple storage changes on that function
                let (borrowed_amount, platform_fee) = self._deposit_and_borrow(user_share);
                let net_investment_for_rollover = borrowed_amount - platform_fee;
                // @audit-issue Three more storage changes.
                self
                    .cycle_total_collateral
                    .write(new_cycle, self.cycle_total_collateral.read(new_cycle) + user_share);
                self
                    .cycle_initial_debt
                    .write(new_cycle, self.cycle_initial_debt.read(new_cycle) + borrowed_amount);
                self
                    .cycle_total_net_investment
                    .write(
                        new_cycle,
                        self.cycle_total_net_investment.read(new_cycle)
                            + net_investment_for_rollover,
                    );
                // @audit-issue One more storage changes.
                self.user_cycle_collateral.write((user_address, new_cycle), user_share);
                // For rollovers, collateral is active for the full duration of the new cycle.
                let cycle_duration = self.cycle_duration.read();
                let weighted_collateral = user_share * cycle_duration.into();
                // @audit-issue Two more storage changes.
                self
                    .user_cycle_weighted_collateral
                    .write((user_address, new_cycle), weighted_collateral);
                let total_weighted_collateral = self
                    .cycle_total_weighted_collateral
                    .read(new_cycle);
                self
                    .cycle_total_weighted_collateral
                    .write(new_cycle, total_weighted_collateral + weighted_collateral);
                // ...
            }
        }
```

Recommendation(s): Optimize design of cycle transition.

Status: Mitigated

Update from the client: Mitigated in commit 77ee3b3

6.3 Medium

6.3.1 `deposit_yield(...)` is not validating `cycle_id`

File(s): /src/yield_pool.cairo

Description: The `deposit_yield(...)` function handles rewards from the trader wallet. However, there is no check that `cycle_id` is a valid and current `cycle_id`, so rewards can be mistakenly deposited into the wrong cycles.

```
fn deposit_yield(ref self: ContractState, cycle_id: u64, amount: u256) {
    self.pausable.assert_not_paused();
    assert!(get_caller_address() == self.trader_wallet.read(), "Unauthorized");
    let vault_addr = self.vault.read();
    assert!(vault_addr != Zero::zero(), "Vault address not set");
    // @audit-issue [Medium] Validate cycle_id is a valid cycle.
    // Transfer USDC from trader wallet
    IERC20Dispatcher { contract_address: self.usdc.read() }
        .transfer_from(self.trader_wallet.read(), get_contract_address(), amount);
    // Update cycle yield
    self.cycle_yield.write(cycle_id, self.cycle_yield.read(cycle_id) + amount);
    self.emit(YieldDeposited { cycle_id, amount });
}
```

Recommendation(s): Consider ensuring `cycle_id` is a valid cycle.

Status: Mitigated

Update from the client: Mitigated in commit 83d082f

6.3.2 Emergency withdrawal wait period has no upper limit

File(s): /src/yield_pool.cairo

Description: Emergency withdrawals can be done after specific time passes. That delay value can be set via `set_emergency_withdrawal_wait_period(...)`. There is no check for `new_wait_period` values upper limit. It can be set to U64 max value which delay the withdrawals to too long.

```
fn set_emergency_withdrawal_wait_period(ref self: ContractState, new_wait_period: u64) {
    // @audit-issue new_wait_period can be very high to block withdrawals.
    self.ownable.assert_only_owner();
    self.emergency_withdrawal_wait_period.write(new_wait_period);
}
```

Recommendation(s): Consider adding a upper limit for `new_wait_period`.

Status: Fixed

Update from the client: Fixed in commit 3328336

6.3.3 Ownership can be transferred to non deployed contract

File(s): /src/utils/CustomLPToken.cairo

Description: Ownership of the CustomLPToken can be transferred via `set_owner(...)` function, but that function doesnt checks new owner is zero address or a valid deployed account address.

```
fn set_owner(ref self: ContractState, new_owner: ContractAddress) {
    let caller = get_caller_address();
    assert!(caller == self.owner.read(), "Only owner can transfer ownership");
    // @audit-issue Ownership can be transferred to non-deployed contract or zero address

    let previous_owner = self.owner.read();
    self.owner.write(new_owner);
    self.emit(OwnershipTransferred { previous_owner, new_owner });
}
```

Recommendation(s): Consider checking the `new_owner` contract address points to non-zero class hash. This ensures the `new_owner` is deployed contract.

Status: Mitigated

Update from the client: Mitigated in commit 73bddc7

6.4 Low

6.4.1 No validation for `new_duration` for cycles

File(s): /src/bitcoin_vault.cairo

Description: Cycle durations can be set at constructor or via `set_cycle_duration(...)` function. Set function can be called by only owner or manager. However, that function doesnt checks for `new_duration` value is too low or too high.

```
fn set_cycle_duration(ref self: ContractState, new_duration: u64) {
    // @audit-issue No validation for new_duration
    self._assert_cycle_manager_or_owner();
    // Allow setting next cycle duration even during active cycles
    self.next_cycle_duration.write(new_duration);
}
```

Recommendation(s): Consider adding a lower and upper cap for `new_duration` parameter.

Status: Fixed

Update from the client: Fixed in commit ec5b5f5

6.4.2 Emergency wallets can be identical addresses

File(s): /src/yield_pool.cairo

Description: Emergency logic needs three different addresses. However, these addresses can be identical at the constructor. If identical addresses are set, there is no way to process emergency withdrawal.

```
#[constructor]
fn constructor(
    ref self: ContractState,
    pragma_contract: ContractAddress,
    owner: ContractAddress,
    wbtc: ContractAddress,
    usdc: ContractAddress,
    treasury: ContractAddress,
    trader_wallet: ContractAddress,
    emergency_wallet_1: ContractAddress,
    emergency_wallet_2: ContractAddress,
    emergency_wallet_3: ContractAddress,
    cycle_manager_1: ContractAddress,
    cycle_manager_2: ContractAddress,
    vesu_singleton: ContractAddress,
    vesu_pool_id: felt252,
    vesu_oracle: ContractAddress,
    lp_token: ContractAddress,
) {
    // ...
    self.emergency_wallet_1.write(emergency_wallet_1);
    self.emergency_wallet_2.write(emergency_wallet_2);
    self.emergency_wallet_3.write(emergency_wallet_3);
    // ...
}
```

Recommendation(s): Consider checking `emergency_wallet` addresses are not identical.

Status: Fixed

Update from the client: Fixed in commit be45d74

6.4.3 Emergency withdrawal logic has no cancel mechanism

File(s): /src/yield_pool.cairo

Description: The yield pool contract has an emergency withdrawal mechanism. Any of the emergency wallets can initiate an emergency withdrawal, and once there are more than or equal to two approvals, funds can be transferred to the treasury wallet by calling the `emergency_withdraw` function. However, there is no cancel mechanism for emergency withdrawal. So once an emergency withdrawal is initiated accidentally or by a malicious actor, there is no way to cancel that withdrawal.

```
fn request_emergency_withdrawal(ref self: ContractState) {
    // Once initialized. There are no way to cancel.
    // ...
}
```

Recommendation(s): Consider adding cancel logic.

Status: Fixed

Update from the client: Fixed in commit ea39457

6.5 Best Practices

6.5.1 Duplicated start cycle logic

File(s): /src/bitcoin_vault.cairo

Description: Starting new cycle done at `start_cycle(...)` or `_start_next_cycle(...)` functions. However, these functions includes same logic. This increases maintenance burden and risk of inconsistency in future updates.

```
fn start_cycle(ref self: ContractState) {
    self._assert_cycle_manager_or_owner();
    assert!(!self.active_cycle.read(), "Previous cycle active");
    let new_cycle = self.current_cycle.read() + 1;
    self.current_cycle.write(new_cycle);
    self.cycle_start_times.write(new_cycle, get_block_timestamp());
    // Use next_cycle_duration for the new cycle
    let next_duration = self.next_cycle_duration.read();
    self.cycle_duration.write(next_duration);
    self.active_cycle.write(true);
    self.emit(CycleStarted { cycle_id: new_cycle, start_time: get_block_timestamp() });
}
```

```
fn *start*next_cycle(ref self: ContractState) {
    let new_cycle = self.current_cycle.read() + 1;
    self.current_cycle.write(new_cycle);
    self.cycle_start_times.write(new_cycle, get_block_timestamp());
    // Use next_cycle_duration for the new cycle
    let next_duration = self.next_cycle_duration.read();
    self.cycle_duration.write(next_duration);
    self.active_cycle.write(true);
    self.emit(CycleStarted { cycle_id: new_cycle, start_time: get_block_timestamp() });
}
```

Recommendation(s): Refactor to use single implementation

Status: Fixed

Update from the client: Fixed in commit 7d0f9f2

6.5.2 Initiating emergency is not pausing contract

File(s): /src/yield_pool.cairo, /src/bitcoin_vault.cairo

Description: An emergency can be initiated by calling the `request_emergency_withdrawal(...)` function on both the yield pool and the vault. This initiates an emergency state and waits for the other emergency wallets' approval to initiate the fund withdrawal process. However, during the emergency wait period;

```
let wait_period = self.emergency_withdrawal_wait_period.read();
assert!(get_block_timestamp() >= request_time + wait_period, "Wait period not over");
```

Users can still make deposits, and their funds can be stuck in the contract.

Recommendation(s): Consider re-designing emergency logic. Pausing the contract directly if an emergency is requested can cause DOS if any emergency wallets are compromised.

Status: Fixed

Update from the client: Fixed in commit 130fc7d

7 Compilation Evaluation

7.1 Compilation Output

```
scarb build
    Updating git repository https://github.com/astraly-labs/pragma-lib
warn: `edition` field not set in `[package]` section for package `pragma_lib`
 Downloading openzeppelin_utils v1.0.0
 Downloading openzeppelin_governance v1.0.0
 Downloading alexandria_data_structures v0.5.1
 Downloading openzeppelin_upgrades v1.0.0
 Downloading alexandria_math v0.5.1
 Downloading openzeppelin_introspection v1.0.0
 Downloading openzeppelin_access v1.0.0
 Downloading openzeppelin_merkle_tree v1.0.0
 Downloading openzeppelin_finance v1.0.0
 Downloading openzeppelin_security v1.0.0
 Downloading openzeppelin_account v1.0.0
 Downloading openzeppelin_token v1.0.0
 Downloading openzeppelin v1.0.0
 Downloading alexandria_storage v0.5.1
 Downloading snforge_std v0.43.1
 Downloading snforge_scarb_plugin v0.43.1
 Downloading openzeppelin_presets v1.0.0
   Compiling lib(caddy_finance_contracts) caddy_finance_contracts v0.1.0 (/home/runner/work/069-Caddy-Finance
    /069-Caddy-Finance/contracts/Scarb.toml)
warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/bitcoin_vault.cairo:608:17
            let total_collateral_before = self.user_collateral.read(caller);
                ^^^^^^^^^^^^^^^^^^^^^^^

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/bitcoin_vault.cairo:1503:17
            let this = get_contract_address();
                ^^^^

warn: Unused import: `caddy_finance_contracts::mocks::mock_erc20::ERC20::Zero`
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/mocks/mock_erc20.cairo:21:28
    use core::num::traits::Zero;
                          ^^^^

warn: Usage of deprecated feature `"deprecated-starknet-consts"` with no `#[feature("deprecated-starknet-consts")]
    ` attribute. Note: "Use␣`TryInto::try_into`␣in␣const␣context␣instead."
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/utils/constants.cairo:2:33
use starknet::{ContractAddress, contract_address_const};
                                ^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `caddy_finance_contracts::utils::CustomLPToken::CustomLPToken::Zero`
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/utils/CustomLPToken.cairo:37:28
    use core::num::traits::Zero;
                          ^^^^

   Compiling starknet-contract(caddy_finance_contracts) caddy_finance_contracts v0.1.0 (/home/runner/work/069-
    Caddy-Finance/069-Caddy-Finance/contracts/Scarb.toml)
warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/bitcoin_vault.cairo:608:17
            let total_collateral_before = self.user_collateral.read(caller);
                ^^^^^^^^^^^^^^^^^^^^^^^

warn[E0001]: Unused variable. Consider ignoring by prefixing with `_`.
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/bitcoin_vault.cairo:1503:17
            let this = get_contract_address();
                ^^^^

warn: Unused import: `caddy_finance_contracts::mocks::mock_erc20::ERC20::Zero`
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/mocks/mock_erc20.cairo:21:28
    use core::num::traits::Zero;
```

```
                            ^^^^

warn: Usage of deprecated feature `"deprecated-starknet-consts"` with no `#[feature("deprecated-starknet-consts")]` attribute. Note: "Use ‘TryInto::try_into‘ in const context instead."
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/utils/constants.cairo:2:33
use starknet::{ContractAddress, contract_address_const};
                                ^^^^^^^^^^^^^^^^^^^^^^^

warn: Unused import: `caddy_finance_contracts::utils::CustomLPToken::CustomLPToken::Zero`
 --> /home/runner/work/069-Caddy-Finance/069-Caddy-Finance/contracts/src/utils/CustomLPToken.cairo:37:28
    use core::num::traits::Zero;
                            ^^^^

warn: libfunc `print` is not allowed in the libfuncs list `Default libfunc list`
 --> contract: BitcoinVault
help: try compiling with the `experimental` list
 --> Scarb.toml
    [[target.starknet-contract]]
    allowed-libfuncs-list.name = "experimental"

    Finished `dev` profile target(s) in 2 minutes
```