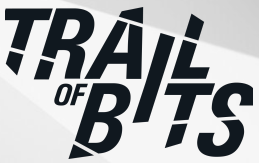

Bluefin Foundation Exchange Contracts

Security Assessment

June 23, 2023

Prepared for:
Rabeel Jawaid
Bluefin Foundation

Prepared by: **Josselin Feist and Anish Naik**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Bluefin engaged Trail of Bits to review the security of the Bluefin exchange contracts. Bluefin exchange is a decentralized exchange for perpetual swaps written in Sui Move for the Sui blockchain.

A team of two consultants conducted the review from March 20 to April 14, 2023, for a total of eight engineer-weeks of effort. Our testing efforts focused on three critical goals. Our primary goal was to assess whether the protocol correctly used Sui's unique object ownership and access control paradigm and followed Sui Move best practices. Second, we focused on the implementation of the mathematical specification to identify any rounding errors, unexpected overflows, or edge cases that could violate system properties. Finally, we focused on the data validation performed in the system to assess whether it allows any opportunities for malicious attack vectors or invalid order matching. With full access to source code and documentation, we performed manual analysis of the target.

Observations and Impact

The Move codebase has been built following the existing Solidity counterpart. As a result, it avoids several pitfalls that commonly affect new codebases, particularly bugs surrounding the logic of the system. However, the project suffers from relying only on integration tests and does not possess any Move-level unit tests. Given the young state of Move/Sui, lacking such tests significantly increases the risks of using the contracts. While the access controls rely properly on Move objects, the access control-related assumptions are undocumented, which increases the likelihood that issues will be introduced during further development. We also found that the overall arithmetic operations suffer from two systematic risks: out-of-order operations and lack of rounding considerations. Finally, some critical parts of the code (e.g., the trade functions) would benefit from more modularity, which would have eased our review of them.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Bluefin take the following steps prior to further development:

- **Remediate the findings disclosed in this report.** The findings described in Detailed Findings pose several risks for the codebase. These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.
- **Perform a thorough analysis of the arithmetic order of operations and rounding.** These two areas have systematic risks, and require further investigations

for the `isolated_liquidation` and `isolated_adl` modules (see appendix J) . Similar risks may exist in the Solidity counterpart.

- **Develop a test suite that uses Move's and Sui's built-in testing framework.** Given the nascency of the Sui and Sui Move technology stack, it would be beneficial to test the system with its native language.
- **Develop thorough inline documentation for the arithmetic operations.** This includes the following actions:
  - Clarify the roles of the variables in the arithmetic formula (e.g, `oiOpen` is described as an "open interest" but it is not an open interest in the traditional sense).
  - Add the arithmetic formulas as inline code documentation when they are defined or used.
- **Documentation and clarify the access controls expectations**. Due to Sui's complex access controls schema, particular care must be taken to document the access controls (see appendix F and appendix G).

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	4
Medium	2
Low	2
Informational	7

CATEGORY BREAKDOWN

Category	*Count*
Access Controls	4
Cryptography	4
Data Validation	4
Error Reporting	2
Patching	1

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineers were associated with this project:

Josselin Feist, Consultant
Josselin@trailofbits.com

Anish Naik, Consultant
anish.naik@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
March 17, 2023	Pre-project kickoff call
March 24, 2023	Status update meeting #1
March 31, 2023	Status update meeting #2
April 7, 2023	Status update meeting #3
April 17, 2023	Delivery of report draft
April 17, 2023	Report readout meeting
June 23, 2023	Delivery of final report

Project Goals

The engagement was scoped to provide a security assessment of the Bluefin Sui exchange contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Do the contracts follow Move/Sui best practices and correctly use the object ownership paradigm?
- Are the Sui objects and capabilities properly used for access controls?
- Does the Sui diverge from the Solidity implementation or the mathematical specification?
- Are the arithmetic operations robust?
- What are the unknowns of using Sui/Move?

Project Targets

The engagement involved a review and testing of the following target.

Exchange Contracts

Repository	https://github.com/fireflyprotocol/bluefin-exchange-contracts-sui
Version	`76e07b9`
Type	Sui Move
Platform	Sui

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

- **`trades/`**. The `trades/` folder holds three modules: `isolated_trading`, `isolated_liquidation`, and `isolated_adl`. The `isolated_trading` module is used to increase, decrease, or reverse a position on a perpetual market. The `isolated_liquidation` module is used to liquidate positions that have more losses than margin. Finally, the `isolated_adl` module is used to perform automated deleveraging (ADL) of high-risk positions. We performed a manual analysis of the three modules and investigated the following:
    - We reviewed the arithmetic operations across all modules to identify any rounding errors. This led to the discovery of four issues that highlight the inconsistency in the order of operations, as well as incorrect rounding direction of certain calculations that may lead to a loss of funds (TOB-BLUEFIN-4, TOB-BLUEFIN-5, TOB-BLUEFIN-12, TOB-BLUEFIN-13). Our arithmetic analysis focused primarily on the `isolated_trading` module; similar issues are likely to be present in the `isolated_liquidation` and `isolated_adl` modules.
    - We reviewed the arithmetic operations across all modules to ensure that they are compliant with the mathematical specification. This investigation did not lead to any findings.
    - We reviewed the `isolated_trading::Order` hashing schema. We explored ways to replay orders, or bypass the signature validation. This led to the discovery of a potential replay attack (TOB-BLUEFIN-1).
    - We reviewed whether it was possible to force a liquidation on a user position. This investigation did not lead to any findings.
    - We reviewed the order validation process across all modules to ensure that the process was compliant with the specification and did not allow an invalid/malicious order to be accepted as valid. This investigation did not lead to any findings.
    - We reviewed the various objects and access controls across these modules to identify any deviations from Sui best practices or opportunities to bypass access controls. This led to the discovery of TOB-BLUEFIN-7, which highlights the unnecessary use of Move abilities in certain objects, and TOB-BLUEFIN-8,

which highlights that the `isolated_liquidation::trade` function is callable by anyone.

- **`margin_bank`**. The `margin_bank` module holds user funds across all perpetual markets. Users are able to deposit and withdraw from the bank. Position updates and liquidations result in bank transfers to and from a perpetual market and user positions. We performed a manual analysis of the module and investigated the following:
    - We reviewed the deposit and withdrawal flows to identify any opportunities to steal funds. This investigation did not lead to any findings.
    - We reviewed the precision conversions between token balances and `BankAccount` balances to identify any opportunities to steal funds and any instances of precision loss or incorrect data validation. This investigation did not lead to any findings.
    - We reviewed the access controls to identify any opportunities for bypass. This investigation did not lead to any findings.
- **`exchange`**. The `exchange` module is the entrypoint for all user interactions. Position updates, liquidations, ADL, and creation of new perpetual markets are all done through the `exchange`. We performed a manual analysis of the module and investigated the following:
    - We reviewed each entry function's input parameters to ensure that mutable and immutable references to objects are used correctly. This investigation did not lead to any findings.
    - We reviewed each entry function to ensure that all the necessary operations are performed in-order to execute the action and transfer the funds between the parties. This investigation did not lead to any findings.
- **`perpetual`**. The `perpetual` module represents a perpetual market. The module holds the state of the perpetual market with a variety of privileged functions to change perpetual-specific parameters. Note that users do not directly interact with the `perpetual` module and instead must go through the `exchange` module. We performed a manual analysis of the module and investigated whether it was possible to perform an access control bypass attack on any of the functions. This investigation did not lead to any findings.
- **`position`**. The `position` module represents a user position. Note that the positions for a perpetual market are stored under the `perpetual::Perpetual` object. The module contains functions to compute various economic parameters

and some `friend`-controlled functions to update a user position. We performed a manual analysis of the module and investigated the following:

  - We reviewed the arithmetic calculations performed in the module to identify whether any of them deviate from the specification. This led to the discovery of TOB-BLUEFIN-10, which highlights that the margin ratio validation does not match the specification.

  - We reviewed the access controls of the module to identify whether it was possible to update another user's position. This investigation did not lead to any findings.

- **`evaluator`**. The `evaluator` module holds the `TradeChecks` object, which aids in trade verification. A `TradeChecks` object is owned by a `perpetual::Perpetual` object. Thus, each perpetual market can use its own `TradeChecks` object to validate trades and ensure that perpetual-specific parameter updates are valid. We performed a manual analysis of the module and investigated whether the verification functions comply with the mathematical specification and do not allow for invalid trades to be executed. These investigations did not lead to any findings.

- **`roles`**. The `roles` module holds the various capabilities associated with the system. The exchange admin, the owner of the `ExchangeAdminCap` capability, has the ability to update the owner of each capability (appendix F). We performed a manual analysis of the module and investigated whether there were any opportunities for access control bypassing and whether all other modules verified authorization to a specific action by using the `CapabilitiesSafe`. These investigations did not lead to any findings.

- **`guardian`**. The `guardian` module holds functions that are callable only by the Guardian, the owner of the `ExchangeGuardianCap`. The Guardian is able to (un)pause withdrawals and enable/disable trading on a perpetual market. We performed a manual analysis of the module and identified that the `guardian` module adds an unnecessary layer of complexity to the access controls of the system (TOB-BLUEFIN-11).

- **`price_oracle`**. The `price_oracle` module represents an on-chain oracle. Currently, the owner of the `PriceOracleOperatorCap` is responsible for updating the on-chain price for a given asset. We performed a manual analysis of the module and investigated the following:

  - We reviewed the arithmetic operation to calculate the price difference between updates to ensure that there are no edge cases that could violate system properties. This investigation did not lead to any issues.

    - We reviewed the access controls to ensure that there were no opportunities for price manipulation. This investigation did not lead to any issues.

- **Arithmetic modules**. There are three modules that contain primarily arithmetic operations: `library`, `margin_math`, and `signed_number`. The `library` module contains basic helper functions for basic arithmetic operations. The `margin_math` module contains helper functions related to margin calculations. Finally, the `signed_number` module holds arithmetic functions for signed numbers. We performed a manual analysis of the module and investigated whether any of the arithmetic operations had incorrect rounding, unexpected overflows, or violated any system properties. These investigations did not lead to any issues.

- **`error`**. The `error` module holds all the custom errors used for the protocol. We performed a manual analysis of the module and investigated whether there is any overlap in custom error codes. This investigation did not lead to any issues.

- **`test_usdc`**. The `test_usdc` module holds a mock implementation of the USDC token since the Sui blockchain does not yet support it. We performed a manual review of this module and investigated whether the module used the `witness` pattern correctly. This investigation did not lead to any findings.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **Missing features**. Some critical features, such as applying the funding rate and checking for trade expiration, have not been implemented yet. Thus, these features were not reviewed during the audit.

- **Bugs in Sui and Move**. We were unable to review any issues that may stem from any underlying, latent bugs in either Sui or Move. Due to the nascency of the technology, we recommend that the Bluefin team keep up to date with the latest developments in the ecosystem.

- **Arithmetics in `isolated_liquidation` and `isolated_adl`**. We were unable to obtain complete coverage of the arithmetic operations in `isolated_liquidation` and `isolated_adl`. The issues found in the `isolated_trading` module (out-of-order operations and lack of rounding considerations) are likely present in these modules.

- **`test`**. The `test` module holds a few functions that perform public key recovery and order hashing. Since this module does not interact with the core protocol, it was considered out of scope for this audit.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The implementation of critical arithmetic operations fails to adhere to the expected best practices. First, the arithmetic operations suffer from two systematic risks: lack of consistency in the operation order, and lack of consideration for the rounding direction. These risks may lead to a loss of funds for the protocol. Second, the trade functions are dense in complexity and computation. This lack of modularity prevents their testing and makes their review more difficult. Third, while documentation was provided through a Notion page, there is a lack of inline code documentation. Finally, some of the variable naming would also benefit from clarification; for example, `oiOpen` is described as an "open interest," but it is not an open interest in the traditional sense).	**Weak**
Auditing	Bluefin provided an incident response plan and monitoring strategy for the Solidity counterpart, but does not currently have the equivalent for the Sui implementation. This is in part due to the young status of the ecosystem.	**Further Investigation Required**
Authentication / Access Controls	Bluefin relies on Sui objects for most of its access controls. It splits the controls over multiple actors, reducing the impact of partial compromise. However, access controls-based Sui objects and modifiers (i.e., `friend`) require documentation to ensure coherence; this documentation was lacking (see appendix F and appendix G).	**Moderate**
Complexity Management	The scope of most functions is clear, and the modules provide sufficient logic boundaries and abstraction.	**Moderate**

	However some functions (e.g., `isolated_trading::trade`) should be refactored to be made more modular. This would simplify the review process and testing.	
Decentralization	The Bluefin protocol heavily relies on an off-chain orderbook to match orders between makers and takers. Thus, the offchain system poses a single point of failure. If the orderbook service crashes or gets compromised, the on-chain system is no longer usable.	**Weak**
Documentation	The public documentation covers the high-level information about the protocol, and the provided Notion documentation covers the arithmetic formulas. However, there is a lack of inline documentation to match the formulas to their implementation and explain the various functions and the purpose of their input parameters. Similarly, the access controls of the Sui objects/modifiers is not documented. Finally, the system would benefit from architectural diagrams to explain how the different modules interact.	**Moderate**
Front-Running Resistance	Front-running risks are limited, as most operations outside of liquidation require privileges. Outside of market orders, the price of orders is properly bound to limit the impact of price manipulation. However, additional documentation should be provided to highlight risks related to the Oracle's price update and relevant MEV risks.	**Satisfactory**
Low-Level Manipulation	There is no assembly or low-level manipulation in the codebase.	**Not Applicable**
Testing and Verification	The system relies uniquely on integration tests that were ported from the Solidity counterpart. However, the codebase lacks Sui unit tests. The protocol should leverage the test_scenario module to obtain satisfactory unit test coverage. Another area that should be explored is the usage of the Move's built-in prover to validate critical system properties.	**Moderate**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Order hashing schema is vulnerable to replay attacks	Cryptography	High
2	Unclear usage of token decimal precision	Data Validation	Informational
3	Order type (maker/taker) is not enforced	Cryptography	Medium
4	Inconsistent order of operations when opening or increasing a position	Cryptography	High
5	Fees in apply_isolated_margin has an incorrect rounding direction	Cryptography	Low
6	Error handling deviates from Sui best practices	Error Reporting	Informational
7	Unnecessary use of Move abilities	Error Reporting	Informational
8	The liquidation module's trade function is callable by any module	Access Controls	Low
9	The create_position function is lacking access controls	Access Controls	Informational
10	Margin ratio validation deviates from the mathematical specification	Data Validation	Informational
11	Overcomplicated access control mechanism for the Guardian	Access Controls	Informational
12	Inconsistent order of operations when flipping positions	Data Validation	High

13	Incorrect rounding in the profit and loss computation allows to withdraw more assets than expected	Data Validation	**Medium**
14	Improper market order design	Access Controls	**High**
15	Sui lacks security maturity	Patching	**Informational**

Detailed Findings

1. Order hashing schema is vulnerable to replay attacks	
Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BLUEFIN-1
Target: `trades/isolated_trading.move`	

Description

The hashing schema used to sign the orders is insufficient to protect against replay attacks.

Every order's hash is signed by the order's maker:

```
fun verify_order_signature(subAccounts: &SubAccounts, maker:address,
hash:vector<u8>, signature: vector<u8>, isTaker:u64):address{

    let publicKey = ecdsa_k1::ecrecover(&signature, &hash);

    let publicAddress = library::get_public_address(publicKey);

    assert!(maker == publicAddress || roles::is_sub_account(subAccounts, maker,
publicAddress), error::order_has_invalid_signature(isTaker));

    return publicAddress
}
```

Figure 1.1: `trades/isolated_trading.move#L445-L454`

The hashing schema of the order contains the following elements:

```
fun get_hash(order:Order): vector<u8>{

    /*
    serializedOrder
     [0,15]     => price            (128 bits = 16 bytes)
     [16,31]    => quantity         (128 bits = 16 bytes)
     [32,47]    => leverage         (128 bits = 16 bytes)
     [48,63]    => expiration       (128 bits = 16 bytes)
     [64,79]    => salt             (128 bits = 16 bytes)
     [80,99]    => maker      (160 bits = 20 bytes)
     [100,119]  => market      (160 bits = 20 bytes)
```

```
    [120,120]  => reduceOnly       (1 byte)
    [121,121]  => isBuy            (1 byte)
  */
```

Figure 1.2: `trades/isolated_trading.move#L353-L366`

This hashing schema lacks protections against signatures replay, in particular:

- If the contract is redeployed, the same signature can be used on multiple contracts.
- There is no domain-specific information, allowing users to re-use signatures that were generated outside of Bluefin purposes.

Exploit Scenario

Eve creates an airdrop that requires a signature. The signature's size collides with Bluefin's hashing schema. Alice signs the airdop's data. Eve re-uses the signature to force a trade on Alice's behalf.

Recommendations

Short term, add the contract's address to the signature schema and pad the data to be signed with a domain-specific text (e.g., `Bluefin:`).

Long term, consider implementing a solution similar to EIP-712.

References

- https://github.com/ethereum/solidity-underhanded-contest/blob/master/2022/submissions_2022/submission10_SantiagoPalladino/SPOILER.md

2. Unclear usage of token decimal precision	
Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BLUEFIN-2
Target: `margin_bank.move`	

Description

The margin bank has unclear and undocumented manipulation on the amount's decimals.

`deposit_to_bank` expects the `amount` to be deposited to be in a six-decimal unit:

```
/*
 * @notice Deposits collateral token from caller's address
 * to provided account address in the bank
 * @dev amount is expected to be in 6 decimal units as
 * the collateral token is USDC
 */
entry fun deposit_to_bank(bank: &mut Bank, destination: address, amount: u64, coin:
&mut Coin<TUSDC>, ctx: &mut TxContext) {
```

Figure 2.1: `margin_bank.move#L99-L105`

`withdraw_from_bank` expects the same behavior from its `amount` parameter, but the decimal information is not documented:

```
/**
 * @notice Performs a withdrawal of margin tokens from the the bank to a provided
address
 */
entry fun withdraw_from_bank(bank: &mut Bank, destination: address, amount: u128,
ctx: &mut TxContext)
```

Figure 2.2: `margin_bank.move#L150-L153`

As a result, a user can call `withdraw_from_bank` with an incorrect `amount` value.

Moreover, when a user deposits USDC to the `margin_bank` module, the deposited amount is multiplied by 1,000, and the depositor's `BankAccount` balance is debited by that value (figure 2.3). The reverse is true for withdrawals.

```
// convert 6 decimal unit amount to 9 decimals
amount = amount * 1000;
```

Figure 2.3: `margin_bank.move#L131-132`

There are two concerns with this approach:

1. Using a hard-coded value of 1,000 is error-prone. In case of a typo or a deviation in the required precision for a `BankAccount`, failing to change all lines where 1,000 is used may lead to a loss of funds or undefined behavior.
2. The rationale for the precision conversion is undocumented.

Exploit Scenario

The Bluefin Foundation team decides to increase the precision of a `BankAccount` by a magnitude of 1,000. This change is done correctly in the `margin_bank::deposit_to_bank` function but not in the `margin_bank::withdraw_from_bank` function. Because of this deviation, Eve is able to drain all the funds from the bank.

Recommendations

Short term, document the decimals expectation for all the functions. Create a `const` value that holds the magnitude difference between a USDC coin balance and a `BankAccount` balance. Additionally, document the rationale behind the precision difference.

Long term, create helper functions that can convert to and from arbitrary decimal precisions. This will allow for greater flexibility as the codebase continues to mature.

3. Order type (maker/taker) is not enforced

Status: **Resolved**	
Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BLUEFIN-3
Target: `trades/isolated_trading.move`	

Description

The hashing schema used to sign the order does not make a distinction between a maker and taker order, which force maker orders to pay a taker fee (or vice versa).

Every order is hashed, and the signature of the hash must be provided by the order's creator:

```
// // get order hashes
let makerHash = get_hash(data.makerOrder);
let takerHash = get_hash(data.takerOrder);
```

Figure 3.1: `trades/isolated_trading.move#L156-L158`

The hashing schema of the order contains the following elements:

```
fun get_hash(order:Order): vector<u8>{

    /*
    serializedOrder
     [0,15]     => price            (128 bits = 16 bytes)
     [16,31]    => quantity         (128 bits = 16 bytes)
     [32,47]    => leverage         (128 bits = 16 bytes)
     [48,63]    => expiration       (128 bits = 16 bytes)
     [64,79]    => salt             (128 bits = 16 bytes)
     [80,99]    => maker     (160 bits = 20 bytes)
     [100,119]  => market     (160 bits = 20 bytes)
     [120,120]  => reduceOnly       (1 byte)
     [121,121]  => isBuy            (1 byte)
    */
```

Figure 3.2: `trades/isolated_trading.move#L353-L366`

The hash does not include any indication that the order is a maker or a taker. A maker's signature can be used for a taker order (and vice versa).

Furthermore, taker and maker orders have a different fees:

```
let makerFee = perpetual::makerFee(perp);
let takerFee = perpetual::takerFee(perp);
```

Figure 3.3: `trades/isolated_trading.move#L145-L146`

As a result, a malicious settlement operator can force the maker orders to pay a taker fee (and vice versa).

Exploit Scenario

Eve creates a perpetual that takes a 0% fee on maker orders and a 20% on taker orders. Bob creates a maker order. Eve uses Bob's order as a taker, thereby forcing Bob to pay the 20% fee.

Recommendations

Short term, add the order type (maker/taker) in the order's hash.

Long term, document the order's related variables and the hashing schema, and ensure that all the order related variables are signed.

4. Inconsistent order of operations when opening or increasing a position	
Status: **Resolved**	
Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BLUEFIN-4
Target: `trades/isolated_trading.move`	

Description

Inconsistencies in the order of operations of computing the margin and transferring the assets cause their values to differ.

When opening a position, or increasing a position size, `fundsFlow` is the amount of assets to be transferred from the user, and `margin` the amount of collateral credited in the position:

- `fundsFlow = quantity * ((price * mro) + feePerUnit)`
- `margin = (quantity * price ) * mro)`

```
marginPerUnit = library::base_mul(fill.price, mro);
fundsFlow = signed_number::from(library::base_mul(fill.quantity, marginPerUnit +
feePerUnit), true);

[..]

position::set_margin(balance, margin +
library::base_mul(library::base_mul(fill.quantity, fill.price), mro));
```

Figure 4.1: `trades/isolated_trading.move#L513-L518`

If the `feePerUnit` is zero, these equation can be reduced to the following:

- `fundsFlow = quantity * (price * mro)`
- `margin = (quantity * price ) * mro)`

Here, both `fundsFlow` and `margin` should be equal, meaning that the user's position will be increased by the exact amount of assets transferred. The multiplications are rounding down:

```
/**
 * @dev Multiplication by a base value with the result rounded down
 */
public fun base_mul(value : u128, baseValue: u128) : u128 {
    return (value * baseValue) / BASE_UINT
}
```

Figure 4.2: `library.move#L24-L29`

Because the order of operations differs between (`quantity * (price * mro)` and `(quantity * price ) * mro`)), the accumulated loss of precision can cause the operations to result in different values, and `fundsFlow =! margin`.

An attacker can abuse this divergence to make an order for which `margin` is greater than `fundsFlow`, allowing the attacker to be credited for more tokens than they sent.

Appendix C contains test cases triggering this issue.

Exploit Scenario

Bob creates a perpetual that takes a 0% fee for makers. Eve is able to create orders that generate 0.01% more margin than the assets deposited. Eve generates thousands of trades over time, and her position has a significantly greater margin than the assets she sent.

Recommendations

Short term, use the same order of operation to compute `fundsFlow` and `margin`.

Long term, we recommend the following steps:

- Add code documentation for the arithmetic formulas.
- Avoid computing the same formulas multiple times.
- Refactor the arithmetic operations to make them modular and ease their testing (see appendix J).
- Investigate fuzzing opportunities in Sui.

5. Fees in apply_isolated_margin has an incorrect rounding direction

Status: **Unresolved**	
Severity: **Low**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BLUEFIN-5
Target: `trades/isolated_trading.move`	

Description

The rounding direction of the fees operation in `apply_isolated_margin` benefits the user instead of the protocol.

All multiplications and division operations that compute the trade's fee are rounding down in `apply_isolated_margin` (including `base_mul` and `base_div`):

```
fundsFlow = signed_number::from(library::base_mul(fill.quantity, marginPerUnit +
feePerUnit), true);


signed_number::mul_uint(
    signed_number::add_uint(
        signed_number::negate(pnlPerUnit),
        feePerUnit),
    fill.quantity),

[..]

signed_number::sub_uint(
    signed_number::mul_uint(
        signed_number::add_uint(
            signed_number::negate(pnlPerUnit),
            closingFeePerUnit),
        qPos),
    margin),
library::base_mul(
    newQPos,
    library::base_mul(
        fill.price,
        mro)
    + feePerUnit)

[..]
```

```
feePerUnit = library::base_div(
    library::base_mul(qPos, closingFeePerUnit) +
    library::base_mul(newQPos,feePerUnit),
    fill.quantity
);

[..]

fee: library::base_mul(feePerUnit, fill.quantity)
```

Figure 4.1: `trades/isolated_trading.move#L513-L518`

As a result, the rounding benefits the user and not the system. In particular, if the quantity is low, the fee can round toward zero.

Appendix D contains a test case to trigger this issue.

Exploit Scenario

Eve creates thousands of orders for a quantity of 1 unit. The fees round down and are equal to zero. As a result, Eve made thousands of trades without paying a fee to the operator.

Recommendations

Short term, round up for all fee-related operations.

Long term:

- Add code documentation for the arithmetic formulas.
- Avoid computing multiple times the same formulas.
- Refactor the arithmetic operations to make them modular and ease their testing (see appendix J).
- Investigate fuzzing opportunities in Sui.

6. Error handling deviates from Sui best practices

Status: **Unresolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-BLUEFIN-6
Target: `error.move`	

Description

The error handling of the protocol uses functions to return specific error codes instead of defining error codes using constants; this is a deviation from Sui's best practices.

Based on a review of a variety of standard Sui modules (e.g., `coin.move`), the best practice for custom errors is to define them as follows:

```
/// A type passed to create_supply is not a one-time witness.
const EBadWitness: u64 = 0;
/// Invalid arguments are passed to a function.
const EInvalidArg: u64 = 1;
/// Trying to split a coin more times than its balance allows.
const ENotEnough: u64 = 2;
```

Figure 6.1: `coin.move#L19-L24`

Note that each custom error is defined as a `const u64` and starts with a capital letter E.

However, this is not the case for the custom errors defined in the `error` module:

```
module bluefin_foundation::error {

    // Setter Errors
    public fun min_price_greater_than_zero() : u64 {
        return 1
    }

    public fun min_price_less_than_max_price() : u64 {
        return 2
    }

    public fun max_price_greater_than_min_price() : u64 {
        return 9
    }
```

Figure 6.2: `error.move#L1-L14`

Recommendations

Short term, create `const u64` values for each custom error and update the contracts to use them accordingly.

Long term, ensure that the protocol follows Sui Move best practices and document any deviations.

7. Unnecessary use of Move abilities	
Status: **Resolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Error Reporting	Finding ID: TOB-BLUEFIN-7
Target: `bluefin_foundation`	

Description

A number of structs within the protocol have unnecessary Move abilities.

For example, the `TradeResponse` struct has the `store` ability. Based on its usage in the protocol, this ability is not needed.

```
struct TradeResponse has copy, store, drop {
    makerFundsFlow: Number,
    takerFundsFlow: Number,
    fee: u128
}
```

Figure 7.1: `trades/isolated_trading.move#L99-103`

The `store` ability should be added to values that need to be stored inside Sui global storage. However, `TradeResponse` is not stored at top-level storage or within any struct; thus, the ability is not needed.

A similar argument can be made for the following structs in the `isolated_trading` module:

- `OrderStatus` does not need the `drop` ability.
- `IMResponse` does not need the `store` ability.

Note that the breadth of this issue was not checked across the entire codebase, and additional structures may have the same issue.

Recommendations

Short term, review the Move abilities provided to each struct in the protocol and identify which ones are unnecessary.

Long term, ensure that the abilities provided to each struct are reviewed periodically since an ability may no longer be necessary as the codebase evolves.

8. The liquidation module's trade function is callable by any module

Status: **Resolved**	
Severity: **Low**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BLUEFIN-8
Target: `trades/isolated_liquidation.move`	

Description

The `isolated_liquidation::trade` function should be callable only by the `exchange` module. However, since the function is missing the `friend` modifier, it is callable by any module, increasing the likelihood of mistakes when code is updated.

```
public fun trade(sender: address, perp: &mut Perpetual, data:TradeData):
TradeResponse{
```

Figure 8.1: `trades/isolated_liquidation.move#L84`

The ability to call the `trade` function directly would allow an attacker to manipulate a liquidatable user's position without actually performing the liquidation. Any honest liquidation can be front-run and cause the liquidation attempt to revert. This would make the position no longer liquidatable and also increase the exposure of the protocol to losses.

It is important to note that, since another module cannot craft a malicious `TradeData` object, this bypass is currently not exploitable. If, however, the `isolated_liquidation::pack_trade_data` function is updated to no longer have the `friend` modifier, the likelihood of an access control bypass significantly increases.

Exploit Scenario

The Bluefin team decides to make the `isolated_liquidation::pack_trade_data` function callable by any module to improve the composability/accessibility of the platform. However, this allows a malicious module to create their own `TradeData` object and call `isolated_liquidation::trade` to update a liquidatable user's position, causing any subsequent honest liquidation attempts to revert.

Recommendations

Short term, add the `friend` modifier to the `isolated_liquidation::trade` function.

Long term, extend the Sui testing suite to ensure that functions that require authorization are only callable by select modules/actors. Document the access controls expectations (see appendix H and appendix I).

9. The `create_position` function is lacking access controls

Status: **Resolved**	
Severity: **Informational**	Difficulty: **High**
Type: Access Controls	Finding ID: TOB-BLUEFIN-9
Target: `position.move`	

Description

The `position::create_position` function, which adds a new position to a given positions table (figure 9.1), has fragile access controls:

```
public fun create_position(perpID:ID, positions: &mut Table<address, UserPosition>,
addr: address){
```

Figure 9.1: `position.move#L170`

This function has no access controls, allowing any module to call it. One of its parameters is a table containing `UserPosition` objects. The perpetual's positions are accessible only through the `perpetual::positions` function, which has the `friend` modifier:

```
public (friend) fun positions(perp:&mut Perpetual):&mut Table<address,UserPosition>{
    return &mut perp.positions
}
```

Figure 9.2: `perpetual.move#L191-193`

As a result, while the function is callable by any module due to its lack of the `friend` modifier, the restriction on `UserPosition` limits its usage.

This situation highlights an important tradeoff for friend modules on non-state changing or getter functions, such as `perpetual::positions`. Adding the `friend` modifier decreases the composability of the system, whereas removing it softens the access controls of the system. Additionally, if a getter function returns a mutable reference to an object, an attacker may be able to manipulate the underlying data.

Recommendations

Short term, add the `friend` modifier to the `position::create_position` function.

Long term, add the `friend` modifier to all non-state changing functions with considerations for any risks to the composability of the system. Document the access controls expectations (see appendix H and appendix I).

10. Margin ratio validation deviates from the mathematical specification

Status: **Resolved**	
Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-BLUEFIN-10
Target: `position.move`	

Description

The margin ratio (MR) validation deviates from the mathematical specification, which may lead to undefined behavior or introduce economic risks.

When a user wishes to increase, reverse, or reduce their position, the `isolated_trading::trade` function will call the `position::verify_collat_checks` function to ensure that the position's MR is sufficient for the trade that the user wishes to perform. If the MR is too low, the transaction reverts and the trade is not completed on-chain.

When evaluating the MR, it is important to check if it is *less than or equal to* the Maintenance Margin Ratio (MMR). If this is the case, the user's position size can only decrease or stay the same. If the position size has increased, the transaction should revert (figure 10.1).

```
// Case III: For MR <= MMR require qPos to go down or stay the same
assert!(
    signed_number::gte_uint(currentMarginRatio, mmr)
    ||
    (
        initialPosition.qPos >= currentPosition.qPos
        &&
        initialPosition.isPosPositive == currentPosition.isPosPositive
    ),
    error::mr_less_than_imr_position_can_only_reduce(isTaker)
);
```

Figure 10.1: `position.move#L216-L226`

However, note that the check in figure 10.1 asserts that the MR must be *greater than or equal to* the MMR. This deviates from the mathematical specification and implies that a user may be able to increase their position size while having a MR that is equal to the MMR.

Note that bypassing this check is not currently possible because the previous MR checks prevent any violations to critical economic properties.

Recommendations

Short term, update the assertion to `assert!(signed_number::gt_uint(currentMarginRatio, mmr)`.

Long term, ensure that all mathematical operations comply with the specification.

11. Overcomplicated access control mechanism for the Guardian

Status: **Resolved**	
Severity: **Informational**	Difficulty: **Low**
Type: Access Controls	Finding ID: TOB-BLUEFIN-11
Target: `guardian.move`	

Description

The access control design for Guardian-related privileges is overcomplicated, which makes it more difficult to maintain and more error-prone.

The owner of the `ExchangeGuardianCap` capability, the Guardian, is responsible for (un)pausing withdrawals (`guardian::set_withdrawal_status`) and enabling/disabling trading on a given perpetual market (`guardian::set_trading_permit`).

```
module bluefin_foundation::guardian {
    use bluefin_foundation::roles::{CapabilitiesSafe, ExchangeGuardianCap};
    use bluefin_foundation::margin_bank::{Self, Bank};
    use bluefin_foundation::roles;
    use bluefin_foundation::perpetual::{Self, Perpetual};

    entry fun set_withdrawal_status(safe: &CapabilitiesSafe, guardian:
&ExchangeGuardianCap, bank: &mut Bank, isWithdrawalAllowed: bool) {
        roles::check_guardian_validity(safe, guardian);
        margin_bank::set_withdrawal_status(bank,isWithdrawalAllowed);
    }

    entry fun set_trading_permit(safe: &CapabilitiesSafe, guardian:
&ExchangeGuardianCap, perp: &mut Perpetual, isTradingPermitted: bool) {
        roles::check_guardian_validity(safe, guardian);
        perpetual::set_trading_permit(perp, isTradingPermitted);
    }
}
```

Figure 11.1: `position.move#L1-L16`

The `guardian::set_withdrawal_status` function will validate that the caller of the function is in fact the Guardian via `roles::check_guardian_validity` and will then call `margin_bank::set_withdrawal_status`. The `margin_bank::set_withdrawal_status` must be protected with an additional `friend` modifier so that only the `guardian` module can call into it (figure 11.2).

```
public (friend) fun set_withdrawal_status(bank: &mut Bank, isWithdrawalAllowed:
bool) {
    // setting the withdrawal allowed flag
    bank.isWithdrawalAllowed = isWithdrawalAllowed;

    emit(WithdrawalStatusUpdate{status: isWithdrawalAllowed});
}
```

Figure 11.2: margin_bank#L88-93

However, this access control design is overcomplicated since the `guardian` module is unnecessary. Allowing the Guardian to directly call into the `margin_bank::set_withdrawal_status` function removes the need to use the `friend` modifier and removes one module-to-module transaction.

A similar reasoning can be applied to the `perpetual::set_trading_permit` function.

Recommendations

Short term, remove the `guardian` module and allow the Guardian to directly call into the `margin_bank::set_withdrawal_status` and the `perpetual::set_trading_permit` functions by using `roles::check_guardian_validity`. Additionally, remove the `friend` modifier from each of these functions.

Long term, identify all privileged operations that can be simplified by removing a layer of complexity. For example, the initialization of a new perpetual market does not need to happen via the `exchange` module. Allowing the exchange admin to directly call a protected `perpetual::initialize` function removes the need for one `friend` modifier and one module-to-module transaction.

12. Inconsistent order of operations when flipping positions

Status: **Resolved**	
Severity: **High**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BLUEFIN-12
Target: `trades/isolated_trading.move`	

Description

Inconsistencies in the order of operations of computing the margin and transferring the assets during a flipping operation cause their values to differ.

When flipping a position, `fundsFlow` is the amount of assets to be transferred from the user, and `margin` the amount of collateral credited in the position:

- `fundsFlow = (-pnlPerUnit + fee) - margin + (quantity - qPos) * (price * mro)`
- `margin = ((quantity - qPos) * price) * mro`

```
fundsFlow = signed_number::add_uint(
                signed_number::sub_uint(
                    signed_number::mul_uint(
                        signed_number::add_uint(
                            signed_number::negate(pnlPerUnit),
                            closingFeePerUnit),
                        qPos),
                    margin),
                library::base_mul(
                    newQPos,
                    library::base_mul(
                        fill.price,
                        mro)
                    + feePerUnit)
            );
```

Figure 12.1: `trades/isolated_trading.move#L578-L592`

If the price of the asset did not change, and the fee are zero, `fundsFlow` can be reduced to the following:

- `0 - margin + (quantity - qPos) * (price * mro)`

Here, both `(quantity - qPos) * (price * mro)` from `fundsFlow` and `((quantity - qPos) * price) * mro` from `margin` should be equal, meaning that the user's position will be increased by the exact amount of assets transferred. The multiplications round down:

```
/**
 * @dev Multiplication by a base value with the result rounded down
 */
public fun base_mul(value : u128, baseValue: u128) : u128 {
    return (value * baseValue) / BASE_UINT
}
```

Figure 12.2: library.move#L24-L29

Because the order of operations differs between `(quantity * (price * mro)` and `(quantity * price ) * mro)`, the accumulated loss of precision can cause the operations to result in different values.

An attacker can abuse this divergence to make an order for which `margin` is greater than `fundsFlow`, allowing the attacker to be credited for more tokens than they sent.

This issue is similar to TOB-BLUEFIN-4.

Appendix E contains test cases triggering this issue.

Exploit Scenario

Bob creates a perpetual that takes a 0% fee for makers. Eve is able to flip orders that generate 0.01% more margin than assets deposited. Eve generates thousands of trades over time, and her position has a significantly greater margin than the assets she sent.

Recommendations

Short term, use the same order of operation to compute `fundsFlow` and `margin`.

Long term, add code documentation for the arithmetic formulas. Avoid computing multiple times the same formulas. Refactor the arithmetic operations to make them modular and ease their testing (see appendix J). Investigate fuzzing opportunities in Sui.

13. Incorrect rounding in the profit and loss computation allows to withdraw more assets than expected

Status: **Unresolved**	
Severity: **Medium**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-BLUEFIN-13
Target: `trades/isolated_trading.move`	

Description

The rounding direction of the fees operation in the profit and loss computation can benefit the user instead of the protocol.

For a reduce order on a long position, `fundsFlow` (the amount of assets transferred) is defined by the following formula:

- `fundsFlow = (- pnlPerUnit + feePerUnit) * quantity - margin * quantity / qPos`

```
fundsFlow = signed_number::sub_uint(
    signed_number::mul_uint(
        signed_number::add_uint(
            signed_number::negate(pnlPerUnit),
            feePerUnit),
        fill.quantity),
    (margin * fill.quantity) / qPos);
```

Figure 13.1: `trades/isolated_trading.move#L542-L548`

If the fee is zero, the formula can be reduced to the following:

- `fundsFlow = - pnlPerUnit * quantity - margin * quantity / qPos`

Here, `pnlPerUnit` represents the profit and loss for the position:

- `pnlPerUnit = price - oiOpen / qPos`

```
public fun compute_average_entry_price(position:UserPosition): u128 {
    return if (position.oiOpen == 0) { 0 } else {
        library::base_div(position.oiOpen, position.qPos)
        }
}
```

Figure 13.2: position.move#L164-L168

```
public fun compute_pnl_per_unit(position: UserPosition, price: u128): Number{
    let pPos = compute_average_entry_price(position);

    return if (position.isPosPositive) {
            signed_number::from_subtraction(price, pPos)
            } else {
            signed_number::from_subtraction(pPos, price)
            }
```

Figure 13.3: trades/isolated_trading.move#L542-L548

If the price has decreased, the following occurs:

- `price - oiOpen / qPos` will return a negative number, and therefore
- `- pnlPerUnit * quantity` will be positive.

This will cause the amount of assets transferred to decrease. All the multiplications and divisions that apply to `pnlPerUnit` round down. As a result, this decrease will always be less than expected, and will profit the user instead of the protocol.

If the shift in the price is low, or the quantity is low, `(price - oiOpen / qPos) * quantity` will round toward zero. As a result, `fundsFlow` will be equal to the same amount if the price of the asset did not change.

The same issue happens in the case of a short position. Appendix F contains a test case to trigger this issue.

Exploit Scenario

Ten thousands of long orders are open on asset A. The price of the asset decreases by 0.1%. All of the orders are reduced, but the owners successfully withdraw the same amount that they would if the asset's price did not change.

Recommendations

Short term, properly round up or down during the profit and loss computation to allocate profit to the protocol instead of to the users.

Long term, document the expected rounding direction for every arithmetic operation. Refactor the arithmetic operations to make them modular and ease their testing (see appendix J). Investigate fuzzing opportunities in Sui.

14. Improper market order design

Status: **Undetermined**	
Severity: **High**	Difficulty: **Medium**
Type: Access Controls	Finding ID: TOB-BLUEFIN-14
Target: `trades/isolated_trading.move`, `sources/evaluator.move`	

Description

Market orders in Bluefin are not actual market orders, as their execution can bypass the current order book.

If a taker order has no price set, its price will be set by the Bluefin-controlled settlement operator:

```
// if taker order is market order
if(data.takerOrder.price == 0){
    data.takerOrder.price = data.fill.price;
};
```

Figure 14.1: `trades/isolated_trading.move#L163-L167`

This price is restricted to be a value within the market bound range:

```
/**
 * verifies if the trade price for both long and short parties confirms to market
take bound checks
 * @dev reversion implies taker order is at fault
 */
public fun verify_market_take_bound_checks(
    checks: TradeChecks,
    tradePrice: u128,
    oraclePrice: u128,
    isBuy: bool
) {
    if(isBuy){
        assert!(tradePrice <= (oraclePrice + library::base_mul(oraclePrice,
checks.mtbLong)), error::trade_price_greater_than_mtb_long());
    }
    else {
        assert!(tradePrice >= (oraclePrice - library::base_mul(oraclePrice,
checks.mtbShort)), error::trade_price_greater_than_mtb_short());
    };
```

```
}
```

Figure 14.2: `evaluator.move#L339-L355`

As a result, these orders do not need to follow the market, allowing a malicious settlement operator to execute the order at an unfair price.

Exploit Scenario

The current order book has millions of trades on a 0.1% price difference. Bob makes a maker order and expects it to be executed within a fair price of the current order book. Eve, a malicious settlement operator, executes the order with a price significantly lower than the current order book depth.

Recommendations

Short term, rename the marker order, and document explicitly that it may not be executed over the current order book.

Long term, improve the visibility regarding the centralization risks in https://learn.bluefin.io/.

15. Sui lacks security maturity	
Status: **Undetermined**	
Severity: **Informational**	Difficulty: **High**
Type: Patching	Finding ID: TOB-BLUEFIN-15
Target: All contracts	

Description

Sui is still at an early stage of development and did not receive a security review. As a result, it is unclear if its underlying protections are properly implemented

The Sui documentation mentions the following:

> *Sui is pre-release software under rapid development. It has not been audited and is not yet ready for production use.*
>
> *After we have completed internal and external audits of Sui, we will establish a responsible disclosure policy and a bug bounty program covering both protocol security and software security. In the meantime, please report security problems to security@mystenlabs.com.*

The fast pace of Sui's development is highlighted by the fact that the branch used during this audit was already not compatible with the Bluefin codebase by the end of the security review due to the changes in devnet-0.29.0.

Moreover, most of the security assumptions of Sui are undocumented (see appendix E), and Move itself continues to receive fixes for security issues (e.g., move-language/1029).

If Bluefin is one of the first systems deployed on the Sui's mainnet, it will inherit the underlying risks of Sui and Move.

Recommendations

Short term, document the risks of Sui/Move to the users before deployment.

Long term, follow Move and Sui issues discussions on GitHub. Follow the results of the Sui security review. Include the risks of compromise for Move/Sui in the incident response plan.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.
Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.

Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Unit tests for issue TOB-BLUEFIN-4

This appendix contains two test cases to trigger TOB-BLUEFIN-4 . These test cases were generated using fuzzing, and highlight different scenarios where assets can be drained from the system:

- Figure C.1: One unit can be drained.
- Figure C.2: 111815645163369119180401366O units can be drained, but the price of the asset is one unit.

```
   #[test]
    fun test_rounding_1() {

        // Value found using fuzzing
        let price = 2190885;
        let quantity = 415768982;
        let leverage = 2000000000;
        let feePerUnit = 0; // do not consider a fee

        let mro =position::compute_mro(leverage);

        // follow apply_isolated_margin, case 1 (Opening position or adding to
position size)
        let marginPerUnit = library::base_mul(price, mro);

        // Here fundsFlow and margin are supposed to be equal if feePerUnit is zero
        // However because the order of the operation are different, the rounding
does not propagated in the same way
        let fundsFlow = library::base_mul(quantity, marginPerUnit + feePerUnit); //
fundsFlow = quantity * (price *mro)  - assuming no fee
        let margin = library::base_mul(library::base_mul(quantity, price), mro); //
margin = (quantity * price) * mro

        // assert with concrete value to ease debugging
        assert!(fundsFlow == 455450, 1);
        assert!(margin == 455451, 1);

        // The following should always be true, otherwise you can receive more
margin than assets transferred, but it fails
        assert!(fundsFlow >= margin, 1);

      }
```

Figure C.1: One unit can be drained.

```
   #[test]
fun test_rounding_2() {

    // Value found using fuzzing
    let price = 1;
```

```
    let quantity = 22363129032673823836080273200782053 62;
    let leverage = 2000000000;
    let feePerUnit = 0; // do not consider a fee

    let mro =position::compute_mro(leverage);

    // follow apply_isolated_margin, case 1 (Opening position or adding to position
size)
    let marginPerUnit = library::base_mul(price, mro);

    // Here fundsFlow and margin are supposed to be equal if feePerUnit is zero
    // However because the order of the operation are different, the rounding does
not propagated in the same way
    let fundsFlow = library::base_mul(quantity, marginPerUnit + feePerUnit); //
fundsFlow = quantity * (price *mro)  - assuming no fee
    let margin = library::base_mul(library::base_mul(quantity, price), mro); //
margin = (quantity * price) * mro

    // assert with concrete value to ease debugging
    assert!(fundsFlow == 0, 1);
    assert!(margin == 111815645163369119180401 3660, 1);

    // The following should always be true, otherwise you can receive more margin
than assets transferred, but it fails
    assert!(fundsFlow >= margin, 1);

}
```

Figure C.2: 111815645163369119180401 3660 units can be drained.

D. Unit tests for issue TOB-BLUEFIN-5

This appendix contains a test case to trigger TOB-BLUEFIN-5, where fees round down incorrectly.

```
   #[test]
    fun test_rounding_3() {

        // Value found using fuzzing
        let price = 2000000000;
        let quantity = 1;
        let leverage = 2000000000;
        let feePerUnit = 100000000; // 10**8

        let mro =position::compute_mro(leverage);

        // follow apply_isolated_margin, case 1 (Opening position or adding to
position size)
        let marginPerUnit = library::base_mul(price, mro);

        // Fee round down
        let fundsFlow = library::base_mul(quantity, marginPerUnit + feePerUnit);
        let margin = library::base_mul(library::base_mul(quantity, price), mro);

        // assert with concrete value to ease debugging
        assert!(fundsFlow == 1, 1);
        assert!(margin == 1, 1);

        // Fee is non-zero, so margin<fundsFlow should be true, but it fails
        assert!(margin < fundsFlow, 1);

      }
```

Figure D.1: Test case

E. Unit tests for issue TOB-BLUEFIN-12

This appendix contains a test case to trigger TOB-BLUEFIN-12, showcasing inconsistent margin amounts.

```
    #[test]
    fun test_rounding_flip() {

        // Value found using fuzzing
        let price = 1176935311;
        let leverage = 2028594565;
        let q0 = 2;
        let q1 = 9610882290262814626862285;

        leverage = library::round_down(leverage);
        let mro =position::compute_mro(leverage);


        // simulate apply_isolated_margin case 1
        // Only track funds flow and margin increase
        // assume no fee
        let fundsFlow_pay = library::base_mul(library::base_mul(price, mro), q0);
        let margin_increase = fundsFlow_pay;

        // simulate apply_isolated_margin case 3
        // Only track funds flow and margin increase
        // assume no fee, no change in price
        let qPos = q0;

        let fundsFlow_pay2 = library::base_mul((q1 - qPos), library::base_mul(price,
mro)) - margin_increase;
        let margin_increase2 = library::base_mul(library::base_mul(q1 - qPos,
price), mro);


        assert!(fundsFlow_pay == 1, 1);
        assert!(margin_increase2 == 5655693368637429002300754, 1);
        assert!(fundsFlow_pay2 == 565569336383198785716934 6, 1);
        assert!(fundsFlow_pay + fundsFlow_pay2 == 5655693363831987857169347, 1);
        // This fail
        // margin increased by 4805441145131407 for free
        //
        assert!(margin_increase2 <= fundsFlow_pay + fundsFlow_pay2, 1);

      }
```

Figure E.1: Test case

F. Unit tests for issue TOB-BLUEFIN-13

This appendix contains a test case to trigger TOB-BLUEFIN-13, showing that incorrect rounding allows attackers to withdraw more assets than intended.

```
   #[test]
    fun test_rounding_3() {

        // Value found using fuzzing
        let price = 2000000000;
        let quantity = 1;
        let leverage = 2000000000;
        let feePerUnit = 100000000; // 10**8

        let mro =position::compute_mro(leverage);

        // follow apply_isolated_margin, case 1 (Opening position or adding to
position size)
        let marginPerUnit = library::base_mul(price, mro);

        // Fee round down
        let fundsFlow = library::base_mul(quantity, marginPerUnit + feePerUnit);
        let margin = library::base_mul(library::base_mul(quantity, price), mro);

        // assert with concrete value to ease debugging
        assert!(fundsFlow == 1, 1);
        assert!(margin == 1, 1);

        // Fee is non-zero, so margin<fundsFlow should be true, but it fails
        assert!(margin < fundsFlow, 1);

      }
```

Figure F.1: Test case

G. Move/Sui Checks

The following list includes some of the built-in checks provided by Move/Sui.

Sui has unclear documentation about its built-in protections (see sui#7604), which can cause developers to make incorrect assumptions. In particular, the documentation does not clarify if some checks are performed by the compiler or at the bytecode level. Checks done by the compiler might not be strong enough, as an attacker could craft malicious bytecode. This appendix aims to highlight checks done at the bytecode level.

- **Restrictions on the init function**. In particular:
  - The `init` function must be private and have two parameters at most.
  - The first parameter must be a `TxContent`, and the second parameter is optional and must be a one-time witness.
  - This is checked by the Sui bytecode verifier: entry_points_verifier.rs#L30-L37
- **Restrictions on functions with the entry modifier.** In particular, any function that has the entry modifier should meet the following criteria:
  - It cannot have any return value, and
  - If it has a `TxContext` parameter, it must be the last parameter.

  This is checked by the sui bytecode verifier: entry_points_verifier.rs#L39-L43
- **Restrictions on structures with the key ability**. In particular, such structures must meet the following criteria:
  - It must have a first field named `id` whose type is `sui::object::UID`. This is checked by the Sui bytecode verifier: struct_with_key_verifier.rs#L4-L7.
  - It must have an ID field that cannot be leaked. This is checked by the Sui bytecode verifier: id_leak_verifier.rs#L4-L14.
- **Restrictions on opcodes.** In particular, all global storage opcodes are forbidden. This is checked by the Sui bytecode verifier: global_storage_access_verifier.rs#L15-L16.
- **Restrictions on structure/objects access.** In particular, the following criteria apply:
  - New instances can be created only inside the module that defines them.

    - Their fields can be accessed only inside the module that defines them.
    - This is checked by Move (advanced-topics/struct.html). Our initial review indicates that it is checked at the bytecode level, but we could not confirm these assumptions.

- **Restrictions on one-time witnesses objects.** These are special structures that have the same name as the module, and they have specific properties. This is checked by the Sui bytecode verifier: one_time_witness_verifier.rs#L4-L17.
- **Restrictions on transfer functions.** All transfer functions (`transfer::transfer<T>(...)`) are restricted to their current module. This is checked by the Sui bytecode verifier: private_generics.rs#L37-L46.

H. Access Control Capabilities

The following appendix lists the various capabilities that are used within the system and highlights each capability's responsibilities.

Capability	Description
`ExchangeAdminCap`	The `ExchangeAdminCap` is the most critical user of the system. The owner of this capability is responsible for the following: 1. Update the exchange guardian 2. Update the price oracle operator 3. Update the deleveraging operator 4. Add/remove a settlement operator 5. Create/remove a perpetual market 6. Update the insurance pool address and percentage 7. Set the fee pool address 8. Update the minimum/maximum price of the perpetual market 9. Update the step/tick size of the perpetual market 10. Update the range bounds for a perpetual market 11. Update the minimum/maximum quantities for limit and market orders for a perpetual market 12. Update the maximum open interest for a given leverage for a perpetual market 13. Update the maximum price difference between oracle updates for a perpetual market
`ExchangeGuardianCap`	The owner of the `ExchangeGuardianCap` is responsible for the following: 1. Enable/disable withdrawals from the bank 2. Enable/disable trading for a perpetual market
`SettlementCap`	The owner of a `SettlementCap` is responsible for executing trades on-chain after they are matched on the off-chain orderbook.

`DeleveragingCap`	The owner of the `DeleveragingCap` is responsible for performing auto-deleveraging of underwater positions.
`PriceOracleOperatorCap`	The owner of the `PriceOracleOperatorCap` is responsible for updating the oracle price for a perpetual.

I. Access Control Review

This appendix lists all the access controls associated with externally accessible functions (public or entry). Sui possesses a complex built-in access controls system, which is a mix of parameter restrictions through the objects system and modifiers (public/friend/entry) of a function.

We recommend that Bluefin review this list and integrate it into their documentation. A particular care must be taken if any change is applied on the access on objects (see TOB-BLUEFIN-8 and TOB-BLUEFIN-9).

Error

Name	Access Controls	Change state	Notes
`error`'s functions	None		

Evaluator

Name	Access Controls	Change state	Notes
`evaluator::verify_X`	None		`TradeChecks` objects can be access through `perpetual::checks`
`evaluator::tickSize`	None		`TradeChecks` objects can be access through `perpetual::checks`

Exchange

`exchange::create_perpetual`	`ExchangeAdminCap`	X	
`exchanges::trade`	`CapabilitiesSafe`	X	
`exchanges::liquidate`	None	X	Allow anyone to liquidate a position
`exchanges::deleverage`	`CapabilitiesSafe`	X	

`exchanges::add_margin`	None	X	
`exchanges::remove_margin`	None	X	
`exchanges::adjust_leverage`	None	X	
`exchanges::close_position`	None	X	

Guardian

Name	Access Controls	Change state	Notes
`guardian::set_withdrawal`	`CapabilitiesSafe`	X	
`guardian::set_trading_permit`	`CapabilitiesSafe`	X	

Isolated_liquidated

Name	Access Controls	Change state	Notes
`isolated_liquidated::trade`	TradeData	X	`TradeData` is accessible only through `isolated_liquidation::pack_trade_data`, which has the `friend` modifier (TOB-BLUEFIN-8)

Library

Name	Access Controls	Change state	Notes
`library`'s functions	None		

Margin_bank

Name	Access Controls	Change state	Notes
`margin_bank::get_balance`	None		Bank is a shared object.
`margin_bank::is_withdrawal_allowed`	None		Bank is a shared object.
`margin_bank::deposit_to_bank`	None	X	
`margin_bank::withdraw_from_bank`	None	X	
`margin_bank::withdraw_all_margin_from_bank`	None	X	

Perpetual

Name	Access Controls	Change state	Notes
`perpetual`'s getters (L207-L265)	None		Perpetual is a shared object.
`perpetual`'s `setters (L272 - L431)`	ExchangeAdminCap	X	`Set_oracle_price` has different access controls
`perpetual::set_oracle_price`	CapabilitiesSafe, PriceOracleOperatorCap	X	

Position

Name	Access Controls	Change state	Notes
position's getters (L69-91)	UserPosition		
position::compute_margin_ratio	UserPosition		
position::compute_average_entry_price	UserPosition		
position::create_position	UserPosition	X	See TOB-BLUEFIN-9
position::verify_collat_checks	UserPosition		
position::compute_mro			
position::compute_pnl_per_unit	UserPosition		
position::is_undercollat	UserPosition		

Price_oracle

Name	Access Controls	Change state	Notes
price_oracle::price	None		PriceOracle can be accessed through perpetual::priceOracle

Roles

Name	Access Controls	Change state	Notes
`roles:check_X`	`CapabilitiesSafe` and `XCap` objects		
`roles::is_sub_account`	`None`		`SubAccounts` is a shared object.
`roles's setters (L137-L237)`	`ExchangeAdminCap`	X	
`roles::set_sub_account`	`None`	X	

Signed_number

Name	Access Controls	Change state	Notes
`signed_number`'s functions	`None`		Can create Number objects and apply arithmetic operations on them

J. Rounding Recommendations

Bluefin's arithmetics always round down on every operation, which can lead to the loss of precision that benefits the user instead of the system. This has caused multiple issues (TOB-BLUEFIN-4, TOB-BLUEFIN-5, TOB-BLUEFIN-12, TOB-BLUEFIN-13), and further issues are likely to be present.

The following describes how to determine the rounding direction per operation.

We recommend applying the same analysis for all arithmetic operations, and to consider rewriting some of the formulas to simplify rounding.

Determining the rounding direction: Simple rounding

To determine how to apply a rounding up or down, one can reason about every operation's outcome.

For example, when a position is open, `fundsFlow` (the amount of assets to be paid) is determined by the following formula:

$$fundsFlow = quantity * (price * \frac{1}{leverage} + feePerUnit)$$

Here, we want `fundsFlow`'s loss of precision favor the protocol; therefore, its value should tend toward a non-zero value (↗). As a result:

- $quantity * (price * \frac{1}{leverage} + feePerUnit)$ must ↗
- $price * \frac{1}{leverage}$ must ↗
- $\frac{1}{leverage}$ must ↗

Which gives:

$$fundsFlow_{\nearrow} = quantity *_{\nearrow} (price *_{\nearrow} \frac{1}{leverage}^{\nearrow} + feePerUnit)$$

Context-dependent rounding: PnL example

Some formulas are context-dependent, as their rounding may depend on their use case. This section provides an analysis of the impact of rounding in the profit and loss computation.

Rounding and negative number

In the context of signed integers, rounding can be context-dependent. For example, for a signed integer representing `fundsFlow`, the amount of assets is transferred as follows:

- If the number is positive, the user will pay assets. The rounding should tend toward a non-zero value (↗).
    - For example, + *1.56* should be rounded toward *+1.6.*
- If the number is negative, the user will receive assets. The rounding should tend toward a zero value (↘).
    - For example, - *1.56* should be rounded toward *-1.5.*

For a reduce order on a long position, `fundsFlow` is determined by the following formula:

$$fundsFlow = (-pnlPerUnit + feePerUnit) * quantity - margin * \frac{quantity}{qPos}$$

Figure J.1: `trades/isolated_trading.move#L551`

With:

$$pnlPerUnit = price - \frac{oiOpen}{qPos}$$

In the code, `fundsFlow` is bounded to be a zero or negative number, reducing the complexity of the associated rules:

```
fundsFlow = signed_number::negative_number(fundsFlow);
```

Figure J.2: `trades/isolated_trading.move#L551`

As a result, `fundsFlow` should tend toward zero (↘). There are two components, separated by the subtraction operation:

$$A = (-pnlPerUnit + feePerUnit) * quantity$$
$$B = margin * \frac{quantity}{qPos}$$
$$fundsFlow_{\searrow} = min(A - B, 0)$$

This formulation makes the rounding complex:

- If A > B, then $fundsFlow$ is zero
- If A < B, then B should round toward a zero value ↘, and then
    - $margin * \frac{quantity}{qPos}$ must ↘
    - $\frac{quantity}{qPos}$ must ↘
- If $pnlPerUnit > feePerUnit$
    - $(-pnlPerUnit + feePerUnit) * quantity$ must ↘
- If $pnlPerUnit < feePerUnit$
    - $(-pnlPerUnit + feePerUnit) * quantity$ must ↗

Here, `pnlPerUnit` can be positive or negative, further increasing the complexity of the associated rules.

Rewriting PnL to ease rounding

`fundsFlow` can be rewritten to ease the rounding and reduce the rounding rules. For example:

$$fundsFlow = (- pnlPerUnit + feePerUnit - \frac{margin}{qPos}) * quantity$$

Figure J.3: `trades/isolated_trading.move#L551`

Considering that `fundsFlow` is bounded to be a zero or negative number, this can be written as:

$$accumulator = - pnlPerUnit + feePerUnit - \frac{margin}{qPos}$$
$$negativeAcc = min(accumulator, 0)$$
$$fundsFlow = negativeAcc * quantity$$

Figure J.4: `trades/isolated_trading.move#L551`

Here, in order to have `fundsFlow` tend toward a zero value (↘), only the rounding in `pnlPerUnit` is context-dependent:

- $pnlPerUnit$ is positive (the long has a profit):
  - $pnlPerUnit$ must ↘
  - $price - \frac{oiOpen}{qPos}$ must ↘
  - $\frac{oiOpen}{qPos}$ must ↗
- $pnlPerUnit$ is negative (the long has a loss):
  - $pnlPerUnit * quantity$ must ↗
  - $price - \frac{oiOpen}{qPos}$ ↗
  - $\frac{oiOpen}{qPos}$ must ↘

Therefore, `compute_pnl_per_unit`, which is used for a long position, should be implemented as follows:

```
pPos_high = compute_average_entry_price_high(position)

if long_position
   # By checking first for price < pPos_high
```

```
# We favorite loss over profit
# In case pPos_low <= price <= pPos_high
if price < pPos_high
   # Return a loss
   return price - pPos_high
# Return a profit
return price - compute_average_entry_price_low(position)
```

Figure J.5: Pseudo code for `compute_pnl_per_unit` for reducing long position order

General rules

The following describes general rules to follow:

- **Start from the desired rounding direction of the entire formula, and analyze the inner components step by step.** Rounding analysis is easier when applied from the outer formula to its inner components.
- **Create arithmetics primitives that round up and down**, such as `base_div_down`, `base_div_up`, `base_mul_up`, and `base_mul_down`.
- **Reduce the number of multiplication and division when the underlying variables have the same sign, or a bound (e.g., min/max) can be applied.** This may reduce the accumulation of precision loss.
- **Reduce the number of variables that can be both positive and negative.** If a variable has a sign that can switch, its rounding direction may also switch.
- **Create unit tests to highlight the result of the precision loss.** Unit tests will help to validate the manual analysis.
- **Use fuzzing and differential testing to increase the confidence level of the testing.** Automated analysis, such as fuzzing and differential testing, will increase the confidence level and the security maturity of the system.

K. Code Quality Recommendations

Trail of Bits recommends the following steps to enhance code quality.

Exchange

- **Clarify the documentation of `create_perpetual` (`sources/exchange.move#L54`).** The current documentation states that *"Transfers adminship of created perpetual to admin,"* which could be interpreted as the transfer of the perpetual object's ownership. This is not possible since a `Perpetual` object is shared.

MarginBank

- **Create helper functions to convert to/from tokens amount and bank's account balances.** Using hard-coded values that are copied/pasted is error prone.

Evaluator

- **Remove the check at `evaluator.move#L273`**. The check `checks.maxPrice > checks.minPrice` is already implicitly verified by the two previous statements.

Trades

- **Follow a consistent order of verification in the `verify_trade` functions.** For example, the `isolated_adl` module checks for `user_position_size_is_zero` and `adl_all_or_nothing_constraint_can_not_be_held` one after the other (in `verify_account`), while the `isolated_liquidation` module checks for `liquidatee_above_mmr` in between. Inconsistency between the checks that are shared makes it more difficult to review them.

Position

- **Remove `if (position.oiOpen == 0) { 0 }` in `compute_average_entry_price (sources/position.move#L165)`.** The function already returns zero if `oiOpen` is zero.
    - If the intent was instead to return zero when `qPos` is zero, update the check.

L. Trade Modules Architecture Recommendations

The following recommendations will facilitate the review and testing of the trade modules and reduce the likelihood of mistakes.

- **Split apply_isolated_margin functions into multiple internal functions to make them modular.** While these functions contain the core arithmetics of the protocol, they are large functions that contain significant complexity. Additionally, this will reduce the amount of repetitive code across the trade modules.
- **Add inline documentation to describe the arithmetic formulas.** The lack of documentation and links to the Notion's documentation impeded our review and increase the likelihood of issues introduced by future updates.
- **Add Sui-level units tests**. Having dedicated functions for the arithmetic will allow developers to create arithmetic tests more easily and reduce the likelihood of issues in this area.
- **Consider splitting large operations through temporary variables.** This will increase the code's readability. For example, Figure L.1 could be split into multiple operations through temporary variables.

```
fundsFlow = signed_number::add_uint(
                signed_number::sub_uint(
                    signed_number::mul_uint(
                        signed_number::add_uint(
                            signed_number::negate(pnlPerUnit),
                            closingFeePerUnit),
                        qPos),
                    margin),
                library::base_mul(
                    newQPos,
                    library::base_mul(
                        fill.price,
                        mro)
                    + feePerUnit)
            );
```

Figure L.1: `trades/isolated_trading.move#L578-L592`

M. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not a comprehensive analysis of the system.

From June 12 to June 13, 2023, Trail of Bits reviewed the fixes and mitigations implemented by the Bluefin team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 15 issues described in this report, Bluefin has resolved 10, has not resolved three issues, and two issues' resolutions remain undetermined. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Order hashing schema is vulnerable to replay attacks	Resolved
2	Unclear usage of token decimal precision	Resolved
3	Order type (maker/taker) is not enforced	Resolved
4	Inconsistent order of operations when opening or increasing a position	Resolved
5	Fees in apply_isolated_margin has an incorrect rounding direction	Unresolved
6	Error handling deviates from Sui best practices	Unresolved
7	Unnecessary use of Move abilities	Resolved
8	The liquidation module's trade function is callable by any module	Resolved
9	The create_position function is lacking access controls	Resolved

10	Margin ratio validation deviates from the mathematical specification	**Resolved**
11	Overcomplicated access control mechanism for the Guardian	**Resolved**
12	Inconsistent order of operations when flipping positions	**Resolved**
13	Incorrect rounding in the profit and loss computation allows to withdraw more assets than expected	**Unresolved**
14	Improper market order design	**Undetermined**
15	Sui lacks security maturity	**Undetermined**

Detailed Fix Review Results

TOB-BLUEFIN-1: Order hashing schema is vulnerable to replay attacks
Resolved in PR 88. The Bluefin team has added domain-specific information to the order payload that must be signed, preventing signature re-use.

TOB-BLUEFIN-2: Unclear usage of token decimal precision
Resolved in PR 75. The Bluefin team updated the decimal conversion logic to use a helper function that converts USDC-decimal precision to Bluefin's base decimal precision for both deposits and withdrawals.

TOB-BLUEFIN-3: Order type (maker/taker) is not enforced
Resolved in PR 83. The Bluefin team added a Boolean parameter to the order object to differentiate between maker and taker orders.

TOB-BLUEFIN-4: Inconsistent order of operations when opening or increasing a position
Resolved in PR 77. The Bluefin team has updated the order of operations when opening/increasing a position to prevent the funds flow and margin calculations from deviating. The team also added inline documentation to better explain the arithmetic operations.

TOB-BLUEFIN-5: Fees in apply_isolated_margin has an incorrect rounding direction
The issue has not been resolved. The client provided the following context for this finding's fix status:

> *This is by design, we truncate the fee owed at the 10th digit deliberately. There is no way around truncation errors when working with finite precision decimals. The best we can do is always round up in favor of the protocol, but we reason that an attack would be too expensive to be viable. Considering the loss of precision at the 10th digit, ~1 billion trades would have to be made to deprive the protocol of $1 worth of fees. With the minor consequence of the design in the monetary value, we prefer to keep the trader-first design.*

TOB-BLUEFIN-6: Error handling deviates from Sui best practices
The issue has not been resolved. The client provided the following context for this finding's fix status:

> *We won't be doing anything for this issue as our approach to managing error codes in a single source/module makes it easy to manage the same errors that are being used across modules. Any future update to an error code will require changes only in a single file.*

TOB-BLUEFIN-7: Unnecessary use of Move abilities
Resolved in PR 76. The Bluefin team identified structs that had superfluous Move abilities and removed them accordingly.

TOB-BLUEFIN-8: The liquidation module's trade function is callable by any module
Resolved in PR 78. The Bluefin team updated the liquidation module's `trade` function to include the `friend` modifier, which will allow only `friend` modules to call that function.

TOB-BLUEFIN-9: The create_position function is lacking access controls
Resolved in PR 79. The Bluefin team updated the position module's `create_position` function to include the `friend` modifier.

TOB-BLUEFIN-10: Margin ratio validation deviates from the mathematical specification
Resolved in PR 80. The Bluefin team updated the margin ratio validation to match that of the mathematical specification.

TOB-BLUEFIN-11: Overcomplicated access control mechanism for the Guardian
Resolved in PR 81. The Bluefin team removed the `guardian` module and updated the `margin_bank::set_withdrawal_status` and the `perpetual::set_trading_permit` functions to ensure that they are callable only by the owner of the `ExchangeGuardianCap` capability.

TOB-BLUEFIN-12: Inconsistent order of operations when flipping positions
Resolved in PR 82. The Bluefin team has updated the order of operations when flipping a position to prevent the funds flow and margin calculations from deviating.

TOB-BLUEFIN-13: Incorrect rounding in the profit and loss computation allows to withdraw more assets than expected
The issue has not been resolved. The client provided the following context for this finding's fix status:

> *Similarly to TOB-BLUEFIN-5, there is no way to get around truncation errors. The best we can do is to round in favor of the protocol. Under normal operation, the profits and losses of the system from rounding down positive and negative user PnLs will even themselves out over many trades in the long run. We believe an attack at the vulnerability would also be too expensive to be viable. At minimum 10^6 trades would be needed to extract $1, given the unlikely accumulation of numeric imprecision.*

TOB-BLUEFIN-14: Improper market order design
Undetermined. The Bluefin team has noted that they will update their public-facing documentation to highlight the pricing mechanism and pricing bounds of the market order. However, since the updated documentation was not provided, this issue's resolution remains "Undetermined."

TOB-BLUEFIN-15: Sui lacks security maturity
Undetermined. The Bluefin team has acknowledged the risk and notes that they will document it before deployment. However, since the updated documentation was not provided, this issue's resolution remains "Undetermined."