plainshift

March 17, 2025

Ekubo Protocol (EVM Deployment)

Comprehensive Security Assessment

contact@plainshift.io
plainshift.io

Table of Contents

Disclaimer

This security assessment represents a time-boxed security review using tooling and manual review methodologies. Our findings reflect our comprehensive evaluation of the materials provided in-scope and are specific to the commit hash referenced in this report.

The scope of this security assessment is strictly limited to the code explicitly specified in the report. External dependencies, integrated third-party services, libraries, and any other code components not explicitly listed in the scope have not been reviewed and are excluded from this assessment.

Any modifications to the reviewed codebase, including but not limited to smart contract upgrades, protocol changes, or external dependency updates will require a new security assessment, as they may introduce considerations not covered in the current review.

In no event shall Plainshift's aggregate liability for all claims, whether in contract or any other theory of liability, exceed the Services Fee paid for this assessment. The client agrees to hold Plainshift harmless against any and all claims for loss, liability, damages, judgments and/or civil charges arising out of exploitation of security vulnerabilities in the contracts reviewed.

By accepting this report, you acknowledge that deployment and implementation decisions rest solely with the client. Any reliance upon the information in this report is at your own discretion and risk. This disclaimer is governed by and construed in accordance with the laws specified in the engagement agreement between Plainshift and the client.

1. Introduction

Plainshift is a full-stack security firm built on the “shift left” security philosophy. We often work with teams early in the product development process to bring security to a greater organizational range than just smart contracts. From the web app, to fuzzing/formal verification, to a team’s operational security, full-stack security can only be achieved by first understanding there is no “scope” to fully protect the users that trust you.

We’re here to meaningfully revolutionize how teams approach security and guide them towards a holistic approach rather than the single sided approach so prevalent today.

Learn more about us at https://plainshift.io.

1.1. Executive Summary

Plainshift was tasked with reviewing Ekubo’s Solidity core contracts, along with the Positions and Router periphery contracts and the Oracle extension, from February 24th, 2025, to March 17th, 2025. We ultimately found and confirmed 1 high severity, 3 low severity and 6 informational issues, many of which were accompanied with PoCs demonstrating their impact.

Based on our assessment, the in-scope codebase is secure, and during our audit, it was evident that many potential issues had already been considered and mitigated. Additional documentation on Tick and SqrtRatio conversions would improve readability and assist in understanding the underlying mechanics of the system.

1.2. Project Timeline

Date	Phase
February 24th, 2025	Kickoff
March 17th, 2025	Audit End
March 17th, 2025	Delivery

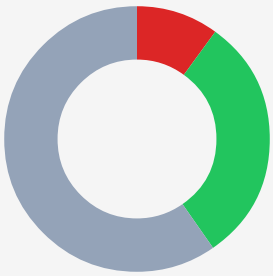
1.3. Scope

Repositories	https://github.com/EkuboProtocol/solidity
Version	`d26b02201060f13946813b055de5e84280e2d3ba`
Contracts	`src/**/*.sol`
Type	Solidity
Platform	Ethereum

1.4. Overview of Findings

Our comprehensive review yielded 1 high, 4 low and 6 informational issues.

Severity/Impact Level	Count
High	1
Medium	0
Low	3
Informational	6



2. Findings

2.1. [H1] Reentrancy in `pay()` function can drain `core`

Target	FlashAccountant.sol#L143-L203
Severity	High
Category	Vulnerability

2.1.1. Description

The `pay()` function allows users to transfer tokens into the `core` contract to increase their balance in the system. It works by taking a snapshot of the contract's token balance before and after passing control-flow to the user. The expectation is that the user will manually transfer tokens into the contract during this callback, and the function credits them with the increase in the contract's balance.

However, since the `core` contract has no reentrancy guards, this logic breaks if `pay()` is called in a nested manner. If a user makes multiple nested calls to `pay()` but only transfers tokens in the final callback, each nested call would detect an increase in token balance and count the same tokens multiple times. This allows the user to repeatedly inflate their balance and withdraw more than they deposited, ultimately draining the contract.

2.1.2. Impact

All ERC20 tokens in the `core` contract can be drained by an attacker inflating their balance. Additionally, any ETH in the contract that is not reserved for protocol or LP fees can be stolen by inflating the balance of the paired ERC20 token and swapping with an artificially large internal balance.

2.1.3. Recommendation

Consider mitigating this issue with a reentrancy guard in the `pay()` function. This would prevent the vulnerability since `pay()` is the only function that relies on an increase in token balance. If a user reenters any other function during the `pay()` callback, it wouldn't introduce an exploit because no other function expects a balance increase.

However, note that some functions do result in a *decrease* in token balance, such as `withdraw()`. If a user calls `withdraw()` within the `pay()` callback, the withdrawal will be accounted for within `withdraw()` itself, and the reduced balance will also be observed in `pay()`. This effectively causes the withdrawal to be counted twice. This isn't an issue because it only negatively impacts the user, so users should avoid reentering these functions during the `pay()` callback.

2.1.4. Remediation

Fixed in commit `f094689` by adding a reentrancy guard in the `pay()` function.

2.2. [L1] `lock()` callback uses wrong sizes

Target	BaseLocker.sol#L63-L67, FlashAccountant.sol#L80-L81
Severity	Low
Category	Spec deviation

2.2.1. Description

The `lock()` function calls the `locked()` callback on the caller, giving them control-flow to interact with the `core` Ekubo contract. The `lock()` function also allows extra calldata to be passed in, which is then forwarded to the callback. This is implemented in the `FlashAccountant` contract as follows:

FlashAccountant.sol

```
function lock() external {
    assembly ("memory-safe") {
        // ...

        let free := mload(0x40)
        // Prepare call to locked(uint256) → selector 0xb45a3c0e
        mstore(free, shl(224, 0xb45a3c0e))
        mstore(add(free, 4), id) // ID argument

        calldatacopy(add(free, 36), 4, sub(calldatasize(), 4))
        let success := call(gas(), caller(), 0, free, calldatasize(), 0, 0)

        // ...
    }
}
```

However, notice that the `call()` only forwards `calldatasize()` bytes, which is incorrect. The `locked()` callback expects 32 additional bytes of calldata (specifically, the `uint256 id` argument), meaning this call is missing 32 bytes.

On the other hand, in the `BaseLocker` contract (a periphery contract that uses the `locked()` callback), the opposite mistake occurs and 32 extra bytes are forwarded. The code mistakenly uses `add(len, 36)`, even though the only additional data being appended is the 4-byte function selector:

BaseLocker.sol

```
function lock(bytes memory data) internal returns (bytes memory result) {
    // ...
    assembly ("memory-safe") {
        result := mload(0x40)
        // Selector of lock()
        mstore(result, shl(224, 0xf83d08ba))

        let len := mload(data)
        mcopy(add(result, 4), add(data, 32), len)

        if iszero(call(gas(), target, 0, result, add(len, 36), 0, 0)) {
            returndatacopy(0, 0, returndatasize())
            revert(0, returndatasize())
        }

        // ...
    }
}
```

2.2.2. Impact

These two errors cancel each other out in the codebase, because `BaseLocker` is the only contract that calls `lock()`. The extra 32 bytes forwarded by the `BaseLocker` are effectively ignored by the `FlashAccountant`, which mistakenly forwards 32 bytes too few back.

However, this issue could cause unexpected behavior for any third-party code that interacts with `lock()` and receives less data back than expected. Additionally, any code that does not forward extra calldata at all would fail to work correctly, as the `uint256 id` argument would be missing from the callback. While these issues would likely be caught during testing, they could still lead to unexpected behavior in external integrations.

2.2.3. Recommendation

Update the `FlashAccountant` contract to forward `calldatasize() + 32` bytes to `locked()` instead of `calldatasize()`, and update the `BaseLocker` contract to forward `add(len, 4)` bytes to `lock()` instead of `add(len, 36)`.

2.2.4. Remediation

Fixed as recommended in commit `2ef7dd3`.

2.3. [L2] Some assembly blocks violate the `memory-safe` rules

Target	FlashAccountant.sol, BaseLocker.sol
Severity	Low
Category	Vulnerability

2.3.1. Description

There are some assembly blocks in the Ekubo codebase that are marked as `memory-safe` but do not strictly follow Solidity's memory safety rules. These rules specify:

> *"A memory-safe assembly block may only access the following memory ranges:"*
>
> - *Memory allocated by yourself.*
> - *Memory allocated by Solidity, e.g. memory within the bounds of a memory array you reference.*
> - *The scratch space between memory offset 0 and 64.*
> - *Temporary memory that is located after the value of the free memory pointer at the beginning of the assembly block, i.e. memory that is "allocated" at the free memory pointer without updating the free memory pointer.*

However, these rules are violated in the `FlashAccountant` and `BaseLocker` contracts, where the following code snippet appears:

FlashAccountant.sol

```
assembly ("memory-safe") {
    // ...
    if iszero(call(gas(), target, 0, result, add(len, 36), 0, 0)) {
        returndatacopy(0, 0, returndatasize())
        revert(0, returndatasize())
    }
    // ...
}
```

This code can potentially overwrite memory slots such as the free memory pointer (`0x40`) and the zero slot (`0x60`), and it does not follow the memory safety rules.

2.3.2. Impact

Since the `memory-safe` annotation tells the compiler which optimizations are safe to perform, an assembly block that is incorrectly marked `memory-safe` is technically undefined behavior. However, it is unlikely that the above code snippet would have caused unexpected behavior since it immediately reverts. Still, it is recommended to follow the rules in these scenarios:

> *"Since this is mainly about the optimizer, these restrictions still need to be followed, even if the assembly block reverts or terminates."*

2.3.3. Recommendation

Consider updating the code to follow the `memory-safe` rules. This can be done by placing return data at the location specified by the free memory pointer instead of starting at memory location zero.

2.3.4. Remediation

Fixed in commit `768ec00` and commit `4e42e35` . Also note that commit `88dc965` introduced a change to the `ExposedStorage` contract that made its assembly blocks no longer `memory-safe` , and this instance has been acknowledged since the risk in this contract is very low and fuzz testing was added in commit `8a6fc15` .

2.4. [L3] Double counting vulnerability on hybrid native/ERC20 chains

Target	FlashAccountant.sol#L145-L220
Severity	Low
Category	Vulnerability

2.4.1. Description

The `pay()` function is designed to transfer `ERC20` tokens from the user to the `core` contract using a balance snapshot, while reentrancy protection is in place to prevent duplicate balance updates. However, on chains with hybrid `native/ERC20` token behavior (such as zkSync Era, Celo, or Polygon), the same token may exist both as a native asset and an ERC20.

An attacker can call `pay()` with an ERC20 token representing the native asset. During the execution of `pay()`, the function invokes `payCallback()`, which the attacker controls. In this callback, the attacker subsequently calls `updatePosition()` to deposit native tokens, because the native token is also represented as an ERC20, the `pay()` function ends up registering both the native token deposit (via `updatePosition()`) and the ERC20 balance change, effectively duplicating the deposit.

2.4.2. Impact

This vulnerability could allow an attacker to drain the core contract of its native/ERC20 tokens. The duplicated deposit means the contract's internal accounting will reflect a much higher token balance than is actually held, enabling the attacker to withdraw more than they deposited.

2.4.3. Recommendation

We recommend that the `pay()` function should not process hybrid native/ERC20 tokens on chains where such dual representations exist (e.g., zkSync Era, Celo, Polygon). Instead, implement explicit checks to detect if the token being processed is hybrid.

2.4.4. Remediation

Acknowledged. After a discussion with the Ekubo team, no changes are necessary for now since they are not planning to deploy on any chain with this behavior. If Ekubo decides to deploy on one of these chains in the future, the code could be updated later on to mitigate this behavior.

2.5. [I1] `balanceOf()` selectively failing could trick `pay()`

Target	FlashAccountant.sol#L143-L203
Severity	Informational
Category	Vulnerability

2.5.1. Description

The `pay()` function takes a snapshot of the `core` contract's token `balanceOf()` before and after giving control-flow to the caller via `payCallback()`. The first `balanceOf()` call is implemented in assembly as follows:

FlashAccountant.sol

```
let tokenBalanceBefore :=
   mul( // The arguments of `mul` are evaluated from right to left.
       mload(free),
       and( // The arguments of `and` are evaluated from right to left.
           gt(returndatasize(), 0x1f), // At least 32 bytes returned.
           staticcall(gas(), token, 0x10, 0x24, free, 0x20)
       )
   )
```

Note that if the `staticcall()` reverts or returns insufficient data, the logic defaults the return value to 0.

This introduces a potential issue: if an attacker can cause the first `balanceOf()` call to fail while also making the second `balanceOf()` call succeed (possibly by manipulating state within the `payCallback()`), the contract could incorrectly observe a positive balance difference even if no tokens were actually transferred.

2.5.2. Impact

For this to be an issue, there would need to be a commonly used token that is also used within Ekubo, where the first `balanceOf()` call could fail while the second `balanceOf()` call succeeds.

This could theoretically happen if a token has unusual `balanceOf()` logic that reverts under certain conditions until an action is taken. However, this seems unlikely.

An attacker might also attempt to forward insufficient gas to the `pay()` function so that the first `balanceOf()` call fails due to an out-of-gas error. However, per EIP-150, only 1/64 of the remaining gas would be available after the first `balanceOf()` call fails. It is unlikely that the `payCallback()` and the second `balanceOf()` call would succeed with just 1/64 of the gas that was insufficient for the first `balanceOf()` call.

After considering potential token implementations, no known example was found where this issue could occur in practice. As a result, this remains a theoretical concern that does not appear to affect most tokens and may never be exploitable within Ekubo.

2.5.3. Recommendation

Consider addressing this concern by propagating any errors observed in the `balanceOf()` call instead of defaulting to zero. Alternatively, if the risk is considered low, the code could remain as is, and tokens used within Ekubo could be monitored for strange `balanceOf()` behavior.

2.5.4. Remediation

Acknowledged.

2.6. [I2] Negating `type(int128).min` in Router can silently overflow

Target	Router.sol#L86
Severity	Informational
Category	Vulnerability

2.6.1. Description

One important edge case to consider is that the absolute value of `type(int128).min` is one larger than `type(int128).max`, meaning that `-type(int128).min` will overflow. If this overflow occurs within an `unchecked` block, it will wrap around and remain a negative number.

In the `Router` contract, this appears to be possible in the slippage check where `amountCalculated` is compared against the user-provided `calculatedAmountThreshold`. For example:

Router.sol

```
unchecked {
    // ...
    (int128 delta0, int128 delta1) = core.swap(value, poolKey, amount, isToken1, sqrtRatioLimit,
    skipAhead);

    int128 amountCalculated = isToken1 ? -delta0 : -delta1;
    if (amountCalculated < calculatedAmountThreshold) {
        revert SlippageCheckFailed(calculatedAmountThreshold, amountCalculated);
    }
    // ...
}
```

It is technically possible for `delta0` or `delta1` to equal `type(int128).min`. If this happens, `amountCalculated` will overflow when negated, leading to an incorrect value.

2.6.2. Impact

This issue does not appear to be exploitable, as it results in a slippage check that is accidentally too strict rather than too lenient. The described scenario would only occur in an exact input trade, where `amountCalculated` would end up negative instead of positive. Since the slippage check in this case is intended for ensuring that a certain positive output amount is reached, this would simply cause the transaction to revert rather than allowing an unintended trade.

Also, note that it is unlikely for `delta0` or `delta1` to equal `type(int128).min` in the first place, as this would represent an extremely large value for most tokens.

2.6.3. Recommendation

Since this does not seem to be a concern, there is no need to change the code. However, consider documenting this behavior in the code or in a known-issues list.

2.6.4. Remediation

Acknowledged.

2.7. [I3] `QuoteReturnValue(int128,int128)` error thrown in extension could be misinterpreted by the `Router`

Target	Router.sol#L312
Severity	Informational
Category	Vulnerability

2.7.1. Description

The `quote()` function in the `Router` contract performs a `swap()` call and then reverts with the `QuoteReturnValue(int128,int128)` error to undo state changes and return information to the `quote()` function:

Router.sol

```
function handleLockData(uint256, bytes memory data) internal override returns (bytes memory
result) {
    // ...
    if (callType == bytes1(0x00)) {
        // ...
    } else if (callType == bytes1(0x01) || callType == bytes1(0x02)) {
        // ...
    } else if (callType == bytes1(0x03)) {
        // ...

        (int128 delta0, int128 delta1) =
            ICore(payable(accountant)).swap(0, poolKey, amount, isToken1, sqrtRatioLimit,
            skipAhead);

        revert QuoteReturnValue(delta0, delta1);
    }
}
```

The `quote()` function checks for this specific error selector and parses it for its return value.

However, note that if the same error selector is thrown during the `swap()` call, it will bubble up to the `quote()` function just like the intended `QuoteReturnValue()` revert. Since `swap()` can pass control-flow to an extension, an extension could intentionally throw its own `QuoteReturnValue(int128,int128)` error. If this happens, the `quote()` function would misinterpret it as a legitimate quote from `swap()`, even though it could contain arbitrary values.

2.7.2. Impact

Since the `quote()` function is not used anywhere in the codebase, the risk of this issue is limited. This may only be a concern for off-chain logic that relies on `quote()` to calculate quotes.

Also, this issue depends on an extension reverting with the matching error selector, which may not be a concern if the extension is trusted by the caller of `quote()`. However, note that even a non-malicious extension might pass control-flow to another contract that triggers the revert instead.

2.7.3. Recommendation

Consider wrapping the `swap()` call in `quote()` with try-catch logic and throwing a different error if `swap()` fails. This would prevent the issue outright by ensuring that `QuoteReturnValue(int128,int128)` can only be returned from the intended location.

Alternatively, consider documenting this behavior and ensuring that any integrators are aware of it.

2.7.4. Remediation

Acknowledged.

2.8. [I4] `SlippageChecker` might observe balance changes for unrelated reasons

Target	SlippageChecker.sol
Severity	Informational
Category	Vulnerability

2.8.1. Description

The `SlippageChecker` contract works by snapshotting the user's balance into transient storage during a multicall, and then later uses that snapshot to determine how much their balance has changed and whether it meets a slippage threshold.

Since this snapshot is based solely on the user's balance (i.e. their EOA balance), external factors unrelated to Ekubo could potentially cause their balance to increase during the multicall. For example, if the user has an open order to sell an NFT for ETH, and that order is fulfilled during the multicall, the resulting ETH would be included in the balance change. This could make the slippage check more lenient than intended by considering the external transfer as part of the Ekubo-related actions.

2.8.2. Impact

This concern is only an issue if a malicious actor can gain control-flow during the multicall. If they can, they could attempt to introduce unrelated balance increases to manipulate the slippage check in their favor and sandwich any difference. However, it may be unlikely for an attacker to gain control-flow in this way.

Also note that both the `Positions` and `Router` contracts (which inherit from `SlippageChecker`), already have their own slippage values provided in function calls. As a result, the `SlippageChecker` may not be necessary.

2.8.3. Recommendation

Consider whether this potential concern about the `SlippageChecker` warrants updating the contract. Also, consider if the `SlippageChecker` is necessary at all, as the `Positions` and `Router` contracts could rely on their own slippage checks based on the arguments provided in each call.

2.8.4. Remediation

Acknowledged. After discussing with the Ekubo team, it was decided that the `SlippageChecker` may also be removed in the future.

2.9. [I5] Operand evaluation order dependency risks

Target	FlashAccountant.sol#L145-L218
Severity	Informational
Category	Vulnerability

2.9.1. Description

The protocol's `pay()` function, as implemented in `FlashAccountant.sol`, relies on a specific operand evaluation order (right-to-left) in inline assembly to ensure that a `staticcall()` is executed before the returned data size is checked. The relevant code snippet is as follows:

FlashAccountant.sol

```
...
            let tokenBalanceBefore :=
                mul( // The arguments of `mul` are evaluated from right to left.
                    mload(free),
@>                  and( // The arguments of `and` are evaluated from right to left.
                        gt(returndatasize(), 0x1f), // At least 32 bytes returned.
                        staticcall(gas(), token, 0x10, 0x24, free, 0x20)
                    )
                )
...
...
            let tokenBalanceAfter :=
                mul( // The arguments of `mul` are evaluated from right to left.
                    mload(0x20),
@>                  and( // The arguments of `and` are evaluated from right to left.
                        gt(returndatasize(), 0x1f), // At least 32 bytes returned.
                        staticcall(gas(), token, 0x10, 0x24, 0x20, 0x20)
                    )
                )
```

If future compiler optimizations or changes alter this evaluation order, the `staticcall()` may not execute before the `returndatasize()` check, potentially causing balances to be computed with uninitialized or stale memory data.

2.9.2. Impact

If the operand evaluation order is modified, the `staticcall()` might not be executed prior to the `returndatasize()` check. This could lead to an incorrect calculation of token balances, which is critical for the protocol's accounting and fund management.

2.9.3. Recommendation

Keep in mind that future compiler versions could change the evaluation order behavior. While Solidity 0.8.x should maintain compatibility, there is no guarantee that future compiler versions will preserve this evaluation order.

If this behavior changes in a future compiler version, refactoring the code to explicitly execute the `staticcall()` before checking `returndatasize()` would ensure correctness.

2.9.4. Remediation

Acknowledged.

2.10. [I6] ChainID dependency causes inconsistent position identification

Target	MintableNFT.sol#L27-L37
Severity	Informational
Category	Vulnerability

2.10.1. Description

The `saltToId()` public function incorporates `chainid()` into the computation of token IDs to ensure that token IDs are unique and user-friendly across different chains.

MintableNFT.sol

```
...
    function saltToId(address minter, bytes32 salt) public view returns (uint256 result) {
        assembly ("memory-safe") {
            let free := mload(0x40)
            mstore(free, minter)
            mstore(add(free, 32), salt)
            mstore(add(free, 64), chainid())
            mstore(add(free, 96), address())

            result := shr(128, keccak256(free, 128))
        }
    }
...
...
    function mint(bytes32 salt) public payable returns (uint256 id) {
        id = saltToId(msg.sender, salt);
        _mint(msg.sender, id);
    }
```

However, this approach introduces a potential inconsistency if a blockchain undergoes a hard fork that changes its chain ID, the same input parameters (minter and salt) will yield different token IDs on the forked chain compared to the original chain. Consider the following steps:

1. A user calls `mint(salt)` on the original chain where `chainid()` returns `X`.
2. The `saltToId()` function computes the token ID using `minter, salt, chainid(), address()`, resulting in a specific token ID.
3. After a hard fork, the new chain has a different chain ID `Y`.
4. The same input parameters now produce a different token ID on the forked chain.
5. Consequently, positions or data tied to the original token ID may not be recognized on the new chain, leading to management discrepancies.

2.10.2. Impact

While this behavior may not directly compromise funds, it can lead to operational inconsistencies. In a hard fork scenario, external integrations or users managing positions might encounter issues where positions created on the original chain are not recognized on the forked chain even when data positions is available in both chains. This mismatch could cause confusion, mismanagement, or errors in position tracking, potentially affecting user experience and downstream processes.

2.10.3. Recommendation

It is recommended that this design decision be clearly documented so that users and external integrators are aware of the potential for different token IDs on forked chains.

2.10.4. Remediation

Acknowledged.