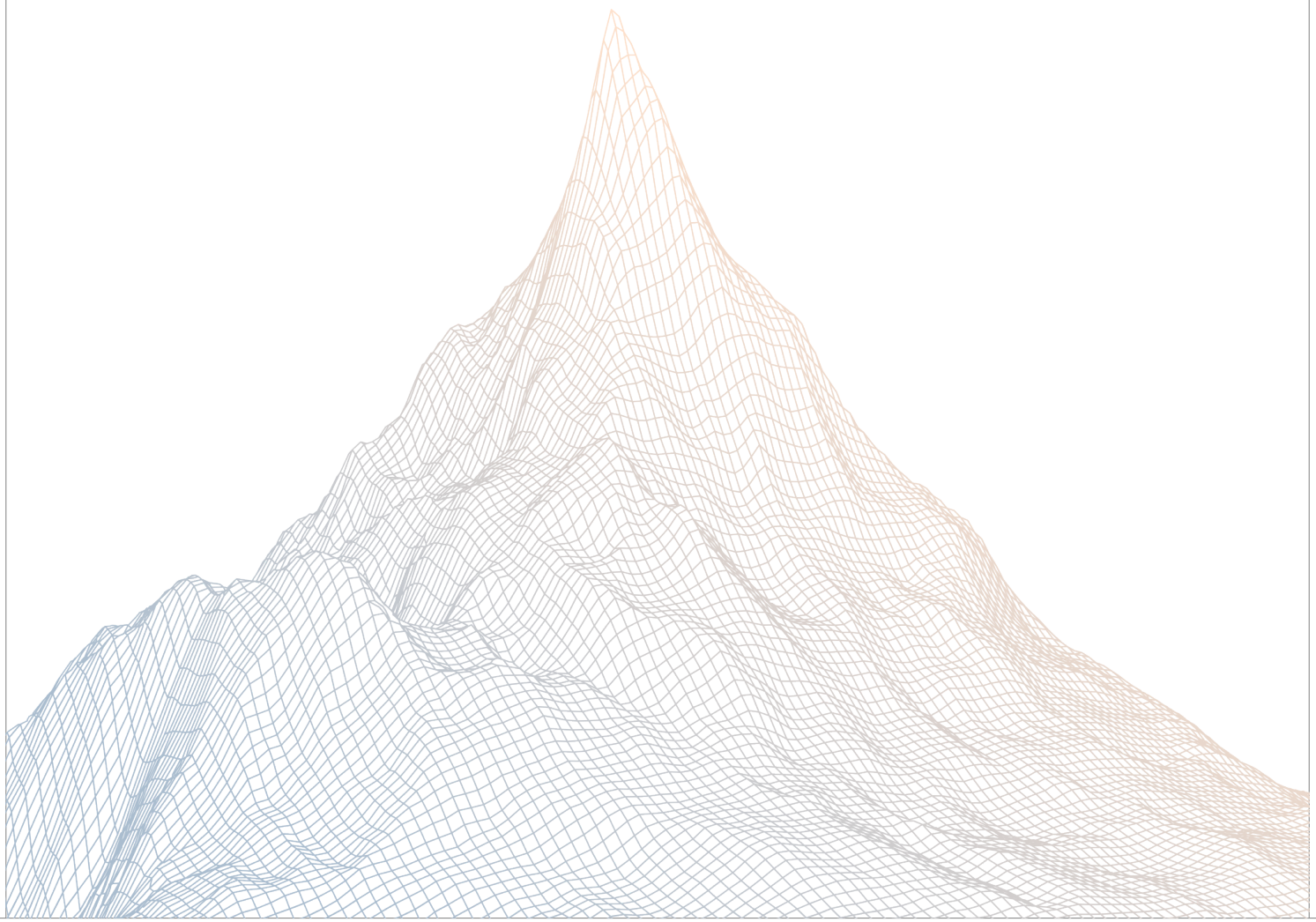
Zenith

Ammplify

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: November 28th to December 2nd, 2025

AUDITED BY: adriro
said

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Ammplify

Ammplify is an on-chain derivative-structuring service for dapps. We innovate with novel uses of traditional AMM liquidity to create and underwrite retail-oriented derivatives for partners like the prediction markets for memecoin launchpads, non-liquidating money markets, alt-yielding stablecoin vaults, and liquidation-free leveraged farming. Ammplify aims to be at the nexus of risk-on yields in DeFi, providing higher yields via diverse dapp integrations and the most sophisticated LP analysis anywhere.

2.2 Scope

The engagement involved a review of the following targets:

Target	NadBets
Repository	https://github.com/itos-finance/NadBets
Commit Hash	`aa2f2091693c6384ded179353a17558298108d18`
Files	`src/*.sol`

2.3 Audit Timeline

November 28, 2025	Audit start
December 2, 2025	Audit end
December 5, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	2
Medium Risk	1
Low Risk	2
Informational	3
Total Issues	**8**

3

Findings Summary

ID	Description	Status
H-1	Reentrancy while settling balances using RFTLib	Resolved
H-2	Overflow leads to zero tax calculation	Resolved
M-1	FlashLender.forgive doesn't release currentOutstanding	Resolved
L-1	verifySignature check have no domain separation	Resolved
L-2	locked flag ineffective due to assignment before revert	Resolved
I-1	TakerVault's ERC-4626 maxWithdraw/maxRedeem over-estimates withdrawable assets	Resolved
I-2	Missing check for wMon in pool tokens	Resolved
I-3	borrow does not ensure flash loan repayment	Acknowledged

4

Findings

4.1 High Risk

A total of 2 high risk findings were identified.

[H-1] Reentrancy while settling balances using RFTLib

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- TakerVault.sol
- MonCollector.sol

Description:

The RFTLib library is used in different contracts to settle balances. The implementation of `RFTLib.settle()` performs a callback when the payer follows a certain implementation, doing the following:

1. Read token balances
2. Execute the callback
3. Read balances again
4. Calculate the difference between 3. and 1. to check if the required amount was transferred.

This can be dangerous since a reentrancy during step 2 could transfer the required funds using another functionality and pass the checks. While `RFTLib.settle()` has a reentrancy guard, this doesn't prevent cross-function reentrancy.

A particular case is the TakerVault contract. The `repay()` function uses `RFTLib.settle()`, in which a malicious actor could re-enter to execute `deposit()` to pass the balance checks and clear the debt without actually repaying anything.

Recommendations:

Apply reentrancy guards in contracts that use the `RFTLib.settle()` functionality. When possible, use simpler patterns such as `transferFrom()` to request funds without handing

control to an untrusted third-party.

Ammplify: Resolved with `@2ee28be220 ...`

Zenith: Verified. `safeTransferFrom` is now used to request funds.

[H-2] Overflow leads to zero tax calculation

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- Bettor.sol

Description:

When a bet is closed, the tax is calculated using the following implementation:

```
uint256 tax = mulX128(payoutSplitX32 << 96, earnings, false);
```

The idea here is to upscale `payoutSplitX32` by 2^96 to align it with 2^128, the dividend in `mulX128()`.

However, since `payoutSplitX32` is of type `uint32`, it will overflow to zero when doing `payoutSplitX32 << 96`, as the operation is performed in the 32-bit domain.

Recommendations:

Convert `payoutSplitX32` to `uint256` before upscaling the value.

```
uint256 tax = mulX128(uint256(payoutSplitX32) << 96, earnings, false);
```

Ammplify: Resolved with `@0f1b8f54cd ...`

Zenith: Verified.

4.2 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] `FlashLender.forgive` doesn't release `currentOutstanding`

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- FlashLender.sol#L80-L87

Description:

`FlashLender.forgive` only reduces the per-borrower `outstanding` mapping and does not decrease the global `currentOutstanding`. As a result, even after the owner writes off a loan, `currentOutstanding` remains unchanged, causing every future borrow call to fail the `verifyIntegrity` check and potentially fail the borrow limit check (`currentOutstanding + add > borrowLimit`). This permanently bricks the flash pool despite the owner explicitly clearing the debt.

```
function forgive(address recipient, uint256 amount) external {
    require(msg.sender == owner, FlashUnauthorized());
    if (outstanding[recipient] < amount) {
        outstanding[recipient] = 0;
    } else {
        outstanding[recipient] -= amount;
    }
}
```

Recommendations:

Decrease `currentOutstanding` inside `forgive`.

Ammplify: Resolved with `@578dbf64c1 ...`

Zenith: Verified.

4.3 Low Risk

A total of 2 low risk findings were identified.

[L-1] `verifySignature` check have no domain separation

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- Bettor.sol#L336-L350

Description:

The `verifySignature` omits both `address(this)` and `block.chainid`, so a signature produced for any `Bettor` deployment / network can be replayed on every other deployment that reuses the same signer.

Anyone who ever receives a valid signature on, e.g., a testnet `Bettor` can reuse it on mainnet to open the same bet without the signer's approval.

Recommendations:

Move to an EIP-712 domain separator or at least append `block.chainid` and `address(this)` to the hashed payload.

Ammplify: Resolved with `@8f25f29188 ...`

Zenith: Verified. The signature now includes the chain ID and the address of the Bettor instance.

[L-2] `locked` flag ineffective due to assignment before revert

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- FlashLender.sol#L106-L107

Description:

The locked flag is supposed to freeze the lender after invariant violation, but it is assigned before revert `IntegrityFailed()`. Because reverting rolls back all state, `locked` never `true` on-chain, every violation just reverts without changing storage. As a result, the lock mechanism is ineffective.

```
function verifyIntegrity(uint256 add) internal {
    if (locked) {
        revert IntegrityFailed();
    }
    if (currentOutstanding + add > borrowLimit) {
        revert InsufficientFlashBalance(add, borrowLimit
    - currentOutstanding);
    }

    if (currentOutstanding ≠ transactionOutstanding) {
        locked = true;
        revert IntegrityFailed();
    }
}
```

Recommendations:

Set `locked = true` and preserve it, then return early from the `borrow` operation instead of reverting.

Ammplify: Resolved with @2be7051e39 ...

Zenith: Verified.

4.4 Informational

A total of 3 informational findings were identified.

[I-1] `TakerVault`'s ERC-4626 `maxWithdraw/maxRedeem` overestimates withdrawable assets

`SEVERITY`: Informational	`IMPACT`: Informational
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- TakerVault.sol

Description:

`TakerVault`'s-ERC-4626's default `maxWithdraw` and preview helpers rely on `totalAssets`, depositors are told they can withdraw their share of `balance + outstanding`, even though the loaned tokens aren't sitting in the vault. calling `maxWithdraw/maxRedeem` will return value that could lead in revert when provided to `withdraw/redeem` operation.

Recommendations:

Override `maxWithdraw`, `maxRedeem` and consider the `outstanding`.

Ammplify: Resolved with `@4115fd0e89 ...`

Zenith: Verified.

[I-2] Missing check for `wMon` in pool tokens

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Bettor.sol#L375-L380

Description:

`setupParams` assumes one token in every pool is `wMon`, but it never validate it. It simply marks whichever token is not `wMon` as the bet asset, even if neither side is MON.

```
function setupParams(BetParams memory betInput)
    internal view returns (Params memory params) {
        // ...
    {
        address token0 = IUniswapV3Pool(betInput.poolAddr).token0();
        address token1 = IUniswapV3Pool(betInput.poolAddr).token1();
        params.is0 = token0 != address(wMon);
        params.token = params.is0 ? token0 : token1;
    }
    // ...
}
```

Recommendations:

Explicitly validate pool tokens, e.g. `require(token0 == address(wMon) || token1 == address(wMon))`.

Ammplify: Resolved with @d083871814 ...

Zenith: Verified.

[I-3] `borrow` does not ensure flash loan repayment

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- FlashLender.sol#L42-L53

Description:

`borrow` never enforces repayment within the same transaction, so any whitelisted borrower can call `borrow`, skip repay, and keep the transferred token. Nothing reverts the original borrow after the transfer if not repaid in the same transaction, it only compares bookkeeping variables before the transfer.

```
function borrow(address recipient, uint256 amount) external {
    require(borrowers[msg.sender], FlashUnauthorized());
    uint256 balance = token.balanceOf(address(this));
    require(balance >= amount, InsufficientFlashBalance(amount, balance));

    verifyIntegrity(amount);
    outstanding[msg.sender] += amount;
    currentOutstanding += amount;
    transactionOutstanding += amount;

    token.safeTransfer(recipient, amount);
}
```

Recommendations:

Require repayment inside the `borrow` logic (e.g., execute the borrower logic via a callback and assert `transactionOutstanding == 0` afterward).

Ammplify: Acknowledged. For now we want to keep it this way in case a flash repayment isn't exact. Then we can notice and unlock.