About Blockpex

Founded in early 2021, BlockApex is a security-first blockchain consulting firm. We offer services in a wide range of areas including Audits for Smart Contracts, Blockchain Protocols, Tokenomics along with Invariant development (i.e test-suite) and Decentralized Application Penetration Testing. With a dedicated team of over 40+ experts dispersed globally, BlockApex has contributed to enhancing the security of essential software components utilized by many users worldwide, including vital systems and technologies.

BlockApex has a focus on blockchain security, maintaining an expertise hub to navigate this dynamic field. We actively contribute to security research and openly share our findings with the community. Our work is available for review at our public repository, showcasing audit reports and insights into our innovative practices.

To stay informed about BlockApex's latest developments, breakthroughs, and services, we invite you to follow us on Twitter and explore our GitHub. For direct inquiries, partnership opportunities, or to learn more about how BlockApex can assist your organization in achieving its security objectives, please visit our "Contact" page at our website, or reach out to us via email at hello@blockapex.io.

Table of Contents

About Blockpex 1
Table of Contents 2
Project Overview 3
Scope Limitation 3
Scope & Version 4
Summary of Findings 5
Detailed Findings 6
- Issue #1 : Global Cooldown Enforcement Enables Denial of Service on Withdrawals 6
- Issue #2 : Reward Tokens (ws, esAMP) Are Recoverable by Governance 8
- Issue #3 : Inaccurate Return Value in convertToAssets When Vault is Empty 10
- Issue #4 : Inaccurate Return Value in convertToShares When Vault is Empty 11
- Issue #5 : Redundant previewDeposit Function 12

Project Overview

The `YieldBearingALPVault` is an EIP-4626-style tokenized vault designed to optimize yield accrual on fsALP tokens(fee + staked ALP) within the Amped Finance ecosystem. The vault allows users to deposit native tokens (ETH), which are then converted into fsALP via the protocol's reward router. In return, users receive **yALP** — a vault share token that represents their ownership of the underlying fsALP pool.

The core innovation of the vault lies in its auto-compounding mechanism: rewards generated from staking fsALP (such as `ws` and `esAMP`) are periodically harvested and reinvested into additional fsALP, effectively increasing the value of each `yALP` token over time. This enables passive and gas-efficient yield farming for users who simply hold `yALP`.

The vault includes a cooldown mechanism to restrict withdrawals shortly after deposits and incorporates manual ERC20-style logic for transfer, approval, and share accounting. While designed for simplicity and gas efficiency, the contract exposes a set of administrative functions, such as compounding, token recovery, and governance role changes, to be managed by trusted actors.

Scope Limitation

The audit exclusively focused on the internal logic of the `YieldBearingALPVault` contract, as it was the only component in scope. No external dependencies or integrations were tested or reviewed as part of this engagement. Specifically, the following external calls present within the contract were not assessed for correctness, reliability, or security guarantees of the callee contracts:

- Line 162: `rewardRouter.mintAndStakeGlpETH`
- Line 205: `rewardRouter.unstakeAndRedeemGlpETH`
- Line 215: `rewardRouter.claim()`
- Line 222: `ws.withdraw(wsBalance)`
- Line 225: `rewardRouter.mintAndStakeGlpETH` (within compounding logic)

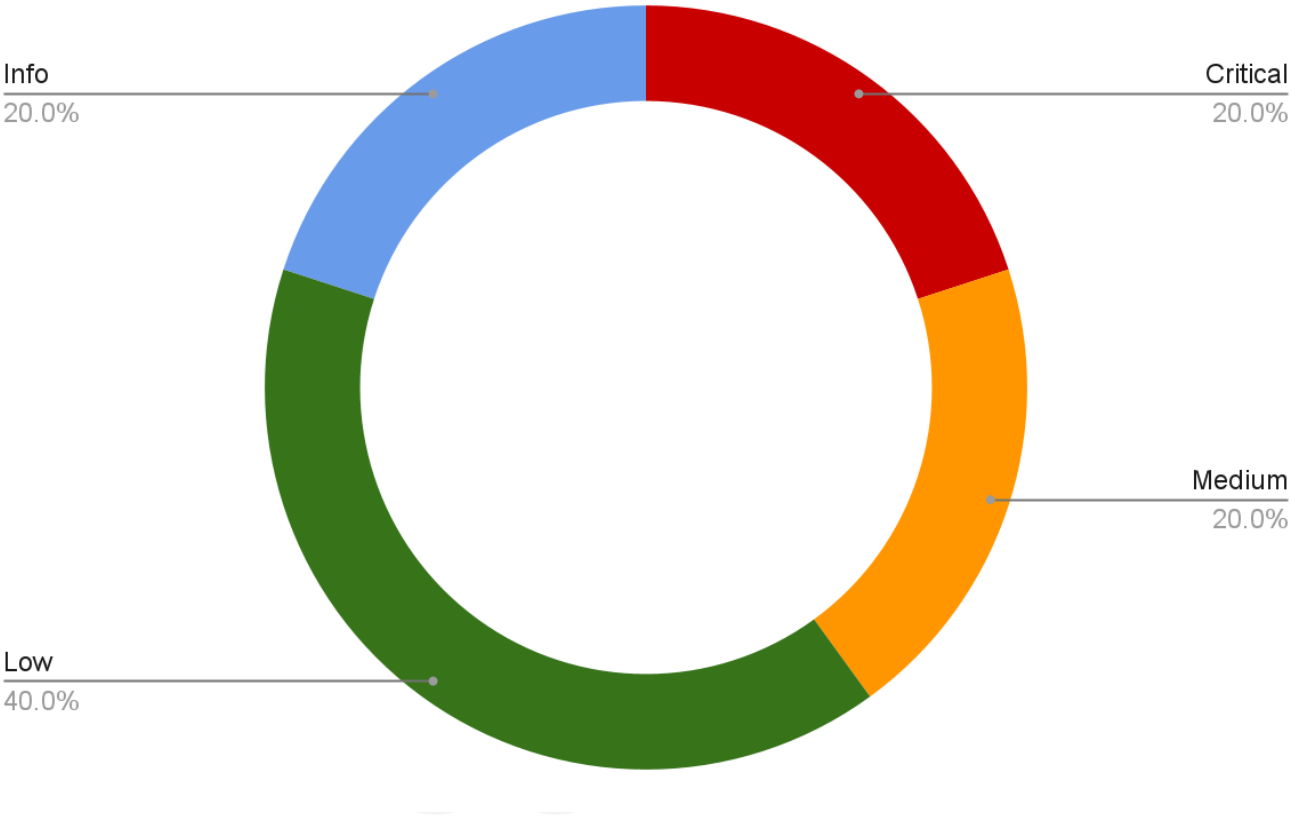
These functions rely on the correct and secure implementation of external contracts such as the `rewardRouter`, `IWETH`(Wrapped Sonic), and associated staking logic. As such, any issues stemming from those external components fall outside the scope of this audit.

Scope & Version

Item	Miscellaneous
Project Name	AMPED
Scope	Repository • https://github.com/amped-finance/amped-smart-contracts/commit/5be7538429ca30c51806b105aa24ce85c5ee7b4e Contract: • contracts/staking/YieldBearingALPVault.sol
Review Type	Comprehensive White-box Code Review
Initiation Date	Wednesday, 4th June, 2025
Delivery Date	Sunday, 8th June, 2025
Version	v1.0.0
Phase	Phase #1

Summary of Findings

During our assessment, we identified one Critical-Severity, one medium-severity issue, 2 low-severity issues and one informational level issues.



Detailed Findings

Issue #1 : Global Cooldown Enforcement Enables Denial of Service on Withdrawals

Severity: Critical

File Location -

- contracts/staking/YieldBearingALPVault.sol

Description

The vault enforces a global cooldown period for withdrawals using a **single** `lastDeposit` timestamp shared across all users:

```
uint256 public lastDeposit;

function withdrawalsAvailableAt() public view returns (uint256) {
    if (lastDeposit == 0) return 0;
    uint256 cooldown = glpManager.cooldownDuration();
    return lastDeposit.add(cooldown);
}
```

This timestamp is updated on every deposit, regardless of which user performs the deposit:

```
lastDeposit = block.timestamp;
```

Withdrawals are blocked until the cooldown has passed:

```
require(block.timestamp >= withdrawalsAvailableAt(), "YieldBearingALP: cooldown active");
```

Impact:

Any user, including a malicious attacker, can continuously reset the cooldown by calling the `depositS()` function with tiny amounts of ETH. This effectively prevents all other

users from withdrawing their funds indefinitely, creating a global Denial of Service (DoS) attack vector against the vault.

This vulnerability is particularly severe because:

- The attacker incurs negligible cost (e.g., 1 wei per deposit)
- The vault holds real user funds that may become inaccessible
- The attack can persist without needing on-chain consensus or privileged roles

Recommendation

Refactor the cooldown logic to use per-user cooldown tracking:

```
mapping(address => uint256) public lastDeposit;
```

Update it only for the depositor:

```
lastDeposit[msg.sender] = block.timestamp;
```

Modify the withdrawal condition accordingly:

```
require(block.timestamp >= lastDeposit[msg.sender] + cooldown, "YieldBearingALP: cooldown active");
```

This ensures cooldowns are enforced fairly and independently, preventing users from interfering with one another's withdrawal eligibility.

Issue #2 : Reward Tokens (`ws`, `esAMP`) Are Recoverable by Governance

Severity: Medium

File Location -

- contracts/staking/YieldBearingALPVault.sol

Description

The `recoverToken()` function allows the `gov` address to recover any ERC-20 token except **`fsALP`**, which is the core asset backing the vault:

```
function recoverToken(address _token, uint256 _amount, address _receiver) external
onlyGov {
    require(_token != address(fsAlp), "YieldBearingALP: cannot recover fsALP");
    IERC20(_token).safeTransfer(_receiver, _amount);
}
```

However, the contract also manages:

- `ws` (Wrapped Sonic) — a reward token actively used in auto-compounding
- `esAMP` — another reward token claimed by the vault

These tokens are not protected in the `recoverToken()` logic, meaning the `gov` can freely transfer them to an external address.

Impact:

Reward tokens (`ws` and `esAMP`) held by the vault may be unintentionally or maliciously withdrawn using `recoverToken()`, bypassing the vault's reward compounding logic. This could:

- Disrupt yield accrual for `yALP` holders
- Undermine trust in the vault's internal accounting
- Allow emergency asset movement without timelock or transparency

While this may be an intentional design choice for flexibility, it also introduces governance risk if not explicitly controlled or disclosed.

Recommendation

Add explicit checks to protect core reward tokens from being recovered unintentionally:

```
require(
    _token != address(fsAlp) &&
    _token != address(ws) &&
    _token != address(esAmp),
    "YieldBearingALP: cannot recover core vault tokens"
);
```

Issue #3 : Inaccurate Return Value in convertToAssets When Vault is Empty

Severity: Low

File Location -

- contracts/staking/YieldBearingALPVault.sol

Description

The `convertToAssets` function is intended to calculate the amount of `fsALP` assets represented by a given number of `yALP` shares. However, when `totalSupply == 0`, the function currently returns the same number of assets as shares:

```
function convertToAssets(uint256 shares) public view returns (uint256) {
    uint256 supply = totalSupply;
    return supply == 0 ? shares : shares.mul(totalAssets()).div(supply);
}
```

This behavior is incorrect. If the vault has no supply and no assets, then shares should not correspond to any real asset value. Returning `shares` creates a misleading 1:1 mapping and may be misinterpreted by other contracts or UIs as meaningful, when in fact the vault is empty.

Impact:

This leads to incorrect assumptions about the value of `yALP` shares during edge cases (e.g., before the first deposit). If any other contract or frontend uses this function to preview or validate user actions, it may falsely interpret non-zero asset value where there is none.

Recommendation

Update the return condition when `totalSupply == 0` to return `0` instead of `shares`:

```
return supply == 0 ? 0 : shares.mul(totalAssets()).div(supply);
```

This change ensures that share-to-asset conversion always reflects actual vault state and avoids incorrect valuation when the vault is uninitialized.

Issue #4 : Inaccurate Return Value in convertToShares When Vault is Empty

Severity: Low

File Location -

- contracts/staking/YieldBearingALPVault.sol

Description

The `convertToShares` function is responsible for converting a given amount of `fsALP` assets into the equivalent number of `yALP` shares. However, the current implementation returns `assets` directly when `totalSupply == 0`:

```
function convertToShares(uint256 assets) public view returns (uint256) {
    uint256 supply = totalSupply;
    return supply == 0 ? assets : assets.mul(supply).div(totalAssets());
}
```

While this may appear convenient for the initial deposit, it is logically inconsistent. When the vault has no `fsALP` and no `yALP`, there is **no valid exchange rate** between assets and shares. Returning the same value (`assets`) falsely implies a 1:1 share issuance rate, which may not align with the vault's real state or intended minting logic.

Impact:

This behavior may lead to incorrect value projections in UI components or external contracts that rely on `convertToShares()` to estimate shares from asset input. It also creates inconsistencies between `convertToShares` and `convertToAssets`, especially during vault bootstrap.

Recommendation

Return `0` when `totalSupply == 0` to clearly indicate that no shares can be issued due to the absence of existing vault liquidity:

```
return supply == 0 ? 0 : assets.mul(supply).div(totalAssets());
```

Alternatively, enforce correct share estimation logic during the actual deposit to avoid encoding assumptions in view functions.

Issue #5 : Redundant `previewDeposit` Function

Severity: Info

File Location -

- contracts/staking/YieldBearingALPVault.sol

Description

The contract defines a `previewDeposit` function as follows:

```
function previewDeposit(uint256 assets) public view returns (uint256) {
    return convertToShares(assets);
}
```

However, `convertToShares` is already a `public` function that provides the same output. The `previewDeposit` function acts as a direct pass-through without adding any logic, validation, or abstraction. Since `convertToShares` already fulfills the exact same purpose and is publicly callable, this additional wrapper appears redundant.

Impact:

While this redundancy does not introduce functional issues or vulnerabilities, it adds unnecessary surface area to the contract and could increase confusion for integrators or developers reading the code. Keeping such wrappers without purpose contradicts the principle of minimalism in smart contract design and may marginally increase gas usage in deployments.

Recommendation

Remove the `previewDeposit` function entirely unless there is a specific forward-compatibility or interface compliance reason to retain it. Consumers can directly use `convertToShares(assets)` to obtain the same output with no functional difference.