Cyberscope

A TAC Security Company

Audit Report

XNAP Manager

December 2025

Contract	LiquidityManager.sol
Sha256	34f5092a894c34763e77786791ca202448ab918317baa3bd5784340ed024781c
Audited by	© cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope’s smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

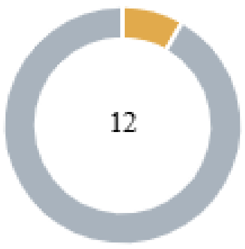
Audit Updates

Initial Audit	07 Nov 2025 https://github.com/cyberscope-io/audits/blob/main/xnap/v3/manger.pdf
Corrected Phase 2	31 Dec 2025

Source Files

Filename	SHA256
LiquidityManager.sol	34f5092a894c34763e77786791ca202448ab918317baa3bd5784340ed024781c

Findings Breakdown



●	Critical	0
●	Medium	1
●	Minor / Informative	11

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	11	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	PCB	Possible Cooldown Bypass	Unresolved
●	AME	Address Manipulation Exploit	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	IPR	ID Purpose Rotation	Unresolved
●	MC	Missing Check	Unresolved
●	PF	Pausable Functionality	Unresolved
●	PMRM	Potential Mocked Router Manipulation	Unresolved
●	RF	Redundant Functionality	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved
●	UTPD	Unverified Third Party Dependencies	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved

PCB - Possible Cooldown Bypass

Criticality	Medium
Location	LiquidityManager.sol#L98
Status	Unresolved

Description

The contract implements a time-delayed execution pattern for liquidity and recovery actions using a queued identifier and attempts to apply a cooldown period to limit repeated executions. However, the cooldown accounting is tied to the queued action identifier rather than the action type itself. As a result, a fresh identifier can be generated for each queued action and execute the same action repeatedly without ever triggering the cooldown restriction, effectively bypassing the intended rate limiting guarantees.

Shell
```
modifier onlyQueued(bytes32 id, bytes32 tag) {
    uint eta = queued[id];
    require(
        eta != 0 &&
        block.timestamp >= eta &&
        block.timestamp <= eta + GRACE,
        "not ready"
    );
    require(purpose[id] == tag, "tag");
    require(
        lastExec[id] == 0 ||
        block.timestamp >= lastExec[id] + cooldown,
        "cooldown"
    );
    lastExec[id] = block.timestamp;
    delete queued[id];
    delete purpose[id];
    emit ActionExecuted(id);
    _;
}
```

Recommendation

The team is advised to scope cooldown tracking to the action purpose instead of the queue identifier. This will enforce cooldown semantics at the action level, preserve the safety assumptions behind delayed liquidity operations, and ensure administrative actions cannot be replayed in rapid succession by cycling identifiers.

AME - Address Manipulation Exploit

Criticality	Minor / Informative
Location	LiquidityManager.sol#L213
Status	Unresolved

Description

The contract's design includes functions that accept external contract addresses as parameters without performing adequate validation or authenticity checks. This lack of verification introduces a significant security risk, as input addresses could be controlled by attackers and point to malicious contracts. Such vulnerabilities could enable attackers to exploit these functions, potentially leading to unauthorized actions or the execution of malicious code under the guise of legitimate operations.

```
Shell
(bool ok, ) = to.call{value: amt}("");
```

Recommendation

To mitigate this risk and enhance the contract's security posture, it is imperative to incorporate comprehensive validation mechanisms for any external contract addresses passed as parameters to functions. This could include checks against a whitelist of approved addresses, verification that the address implements a specific contract interface or other methods that confirm the legitimacy and integrity of the external contract. Implementing such validations helps prevent malicious exploits and ensures that only trusted contracts can interact with sensitive functions.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	LiquidityManager.sol#L121
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

Shell
```
function queue(bytes32 id, bytes32 tag) external onlyOwner
{
    require(queued[id] == 0, "exists");
    uint eta = block.timestamp + delay;
    queued[id] = eta;
    purpose[id] = tag;
    emit ActionQueued(id, tag, eta);
}

function pause() external onlyOwner {
    paused = true;
    emit Paused(msg.sender);
}

function unpause() external onlyOwner {
    paused = false;
    emit Unpaused(msg.sender);
}

function finalizeGovernance() external onlyOwner {
    governanceFinalized = true;
    paused = false;
    emit GovernanceFinalized(msg.sender);
}
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

IPR - ID Purpose Rotation

Criticality	Minor / Informative
Location	LiquidityManager.sol#L121
Status	Unresolved

Description

The contract queues actions for delayed execution using an identifier that is deleted after successful execution. However, the identifier is not permanently bound to its original action purpose and can be re-queued with a different tag once cleared. As a result, an owner can recycle the same identifier to authorize entirely different core operations, meaning the identifier behaves as a reusable queue slot or nonce rather than a unique, immutable action fingerprint.

Shell
```
function queue(bytes32 id, bytes32 tag) external onlyOwner
{
    require(queued[id] == 0, "exists");
    uint eta = block.timestamp + delay;
    queued[id] = eta;
    purpose[id] = tag;
    emit ActionQueued(id, tag, eta);
}
```

Recommendation

The team is advised to bind execution constraints to the action purpose instead of a recyclable identifier. This will prevent purpose rotation through identifier recycling.

MC - Missing Check

Criticality	Minor / Informative
Location	LiquidityManager.sol#L145
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically the contract does not ensure that the returned value from the `getPair` call is not address(0).

```
Shell
function lpToken() public view returns (address) {
    return IPancakeFactory(factory).getPair(token, WETH);
}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

PF - Pausable Functionality

Criticality	Minor / Informative
Location	LiquidityManager.sol#L129
Status	Unresolved

Description

The contract includes a pausing mechanism that allows the owner to halt core liquidity addition and asset recovery functionality. However, the pause control is fully retained by the owner without structural constraints or permanent unpause guarantees. As a result, the owner can interrupt essential operations, including liquidity provisioning, which could conflict with assumptions of continuous protocol availability, especially after governance is declared finalized. This creates a centralization pressure point where availability depends entirely on a single privileged role.

Shell
```
function pause() external onlyOwner {
    paused = true;
    emit Paused(msg.sender);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

PMRM - Potential Mocked Router Manipulation

Criticality	Minor / Informative
Location	LiquidityManager.sol#L79
Status	Unresolved

Description

The contract includes a method that allows the owner to modify the router address This introduces a security threat. The owner could set the router address to any contract that implements the router's interface, potentially containing malicious code. In the event of a transaction triggering the swap functionality with such a malicious contract as the router, the transaction may be manipulated.

Shell
```
router = _router;
```

Recommendation

The team should carefully manage the private keys of the owner’s account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

RF - Redundant Functionality

Criticality	Minor / Informative
Location	LiquidityManager.sol#L93,139
Status	Unresolved

Description

The contract introduces a `governanceFinalized` boolean. This boolean is used in the modifier `onlyOwnerGov` and its purpose is to restrict certain functions when it is false. The owner can change the boolean to true with the use of `finalizeGovernance`. However the modifier is not used in any of the contract's functionality, therefore the modifier, boolean and function that uses it are redundant.

Shell
```
modifier onlyOwnerGov() {
    require(!governanceFinalized, "governance finalized");
    _;
}
function finalizeGovernance() external onlyOwner {
    governanceFinalized = true;
    paused = false;
    emit GovernanceFinalized(msg.sender);
}
```

Recommendation

The team is advised to remove any unusable functionality to enhance code optimization and readability.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	LiquidityManager.sol#L151
Status	Unresolved

Description

The contract does not pull or transfer tokens from users at the moment they are required. Instead, it assumes that all necessary tokens are already pre-deposited and held by the contract before any operation that depends on them is executed. Although this model allows external administration to manage token supply with greater flexibility, it creates a strong reliance on administrator intervention, introducing a central point of dependency. This can result in operational failures, delayed execution, and increased centralization risk, ultimately weakening trust assumptions in the system.

More specifically, in the case of `addLiquidityETH` , the function does not transfer tokens from the user. Instead, it expects the contract itself to already hold the tokens that will be paired with ETH.

Shell
```
function addLiquidityETH(
    uint amtT,
    uint amtTmin,
    uint amtEmin,
    uint deadline,
    bytes32 id
)
    external
    payable
    onlyQueued(id, keccak256("addLiquidityETH"))
    whenNotPaused
    nonReentrant
{
    IERC20(token).forceApprove(router, 0);
    IERC20(token).forceApprove(router, amtT);

    IPancakeRouter02(router).addLiquidityETH{value:
msg.value}(
        token,
        amtT,
        amtTmin,
        amtEmin,
        address(this),
        deadline
    );

    IERC20(token).forceApprove(router, 0);
    emit LiquidityAdded(amtT, msg.value);
}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to send the tokens from the user to the contract when the functions that need them are called.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	LiquidityManager.sol#L78
Status	Unresolved

Description

The contract uses an external contract in order to determine the transaction's flow. The external contract is untrusted. As a result, it may produce security issues and harm the transactions.

Shell
```
token = _token;
```

Recommendation

The contract should use a trusted external source. A trusted source could be either a commonly recognized or an audited contract. The pointing addresses should not be able to change after the initialization.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	LiquidityManager.sol#L53
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

Shell
```
uint256 public cooldown = 14 days
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	LiquidityManager.sol#L39,40
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

Shell
```
address public immutable WETH
uint256 public immutable HARD_LOCK_UNTIL
```

Recommendation

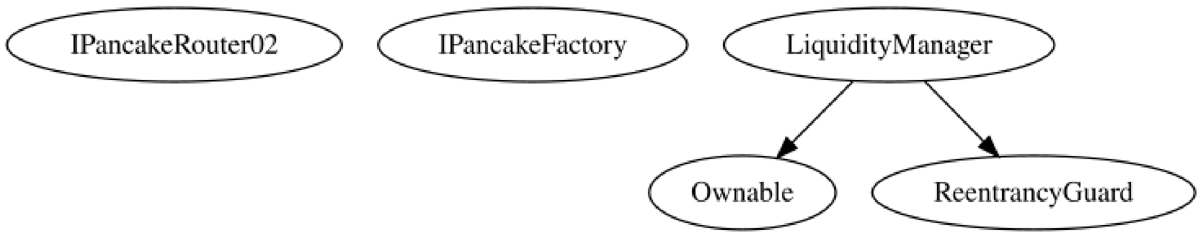
By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

Functions Analysis

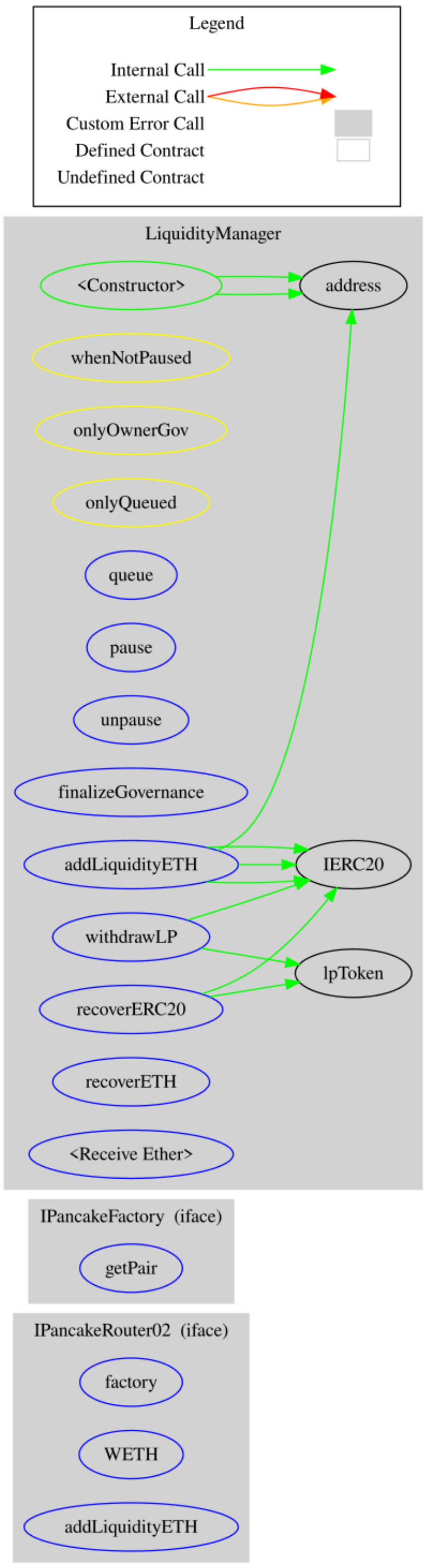
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IPancakeRouter02	Interface			
	factory	External		-
	WETH	External		-
	addLiquidityETH	External	Payable	-
IPancakeFactory	Interface			
	getPair	External		-
LiquidityManager	Implementation	Ownable, ReentrancyGuard		
		Public	✓	Ownable
	queue	External	✓	onlyOwner
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner
	finalizeGovernance	External	✓	onlyOwner
	lpToken	Public		-
	addLiquidityETH	External	Payable	onlyQueued whenNotPaused nonReentrant
	withdrawLP	External	✓	onlyQueued whenNotPaused nonReentrant

	recoverERC20	External	✓	onlyQueued whenNotPaused nonReentrant
	recoverETH	External	✓	onlyQueued whenNotPaused nonReentrant
		External	Payable	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
LiquidityManager
<Constructor>
address
whenNotPaused
onlyOwnerGov
onlyQueued
queue
pause
unpause
finalizeGovernance
addLiquidityETH
IERC20
withdrawLP
lpToken
recoverERC20
recoverETH
<Receive Ether>
IPancakeFactory (iface)
getPair
IPancakeRouter02 (iface)
factory
WETH
addLiquidityETH

Summary

XNAP Token contract implements a locker and utility mechanism. This audit investigates security issues, business logic concerns and potential improvements.

.

Disclaimer

About Cyberscope

Cyberscope is a TAC blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

A *TAC Security* Company

The Cyberscope team

cyberscope.io