Cyberscope
A TAC Security Company

Audit Report
XNAP

November 2025

Network	BSC - TESTNET
Address	0x3eCECc77C859A13fc25f489E0b0991e40D803c6A
Contract	LiquidityManager
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

Explorer	https://testnet.bscscan.com/address/0x3ececc77c859a13fc25f489e0b0991e40d803c6a

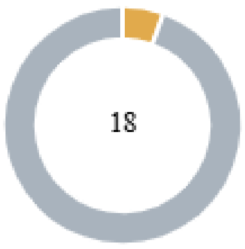
Audit Updates

Initial Audit	04 Nov 2025

Source Files

Filename	SHA256
XNAPToken.sol	b9bbdd53a3748f7e7e61c5c6271bc4ea134183407b1a62b0521fec632ac7c8e7

Findings Breakdown



●	Critical	0
●	Medium	1
●	Minor / Informative	17

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	0	0	0	0
●	Medium	1	0	0	0
●	Minor / Informative	17	0	0	0

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	UCC	Unusable Contract Code	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	CPV	Cross Pool Vulnerability	Unresolved
●	IUT	Inconsistent Unlockable Time	Unresolved
●	MEV	Miner Extraction Value	Unresolved
●	MC	Missing Check	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	ODM	Oracle Decimal Mismatch	Unresolved
●	PAIM	Possible Action ID Misuse	Unresolved
●	PDEW	Possible Double Execution Window	Unresolved
●	POSD	Potential Oracle Stale Data	Unresolved
●	PRE	Potential Reentrance Exploit	Unresolved
●	RSD	Redundant Struct Declaration	Unresolved
●	TSI	Tokens Sufficiency Insurance	Unresolved

●	UTPD	Unverified Third Party Dependencies	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

UCC - Unusable Contract Code

Criticality	Medium
Location	XNAPToken.sol#L211
Status	Unresolved

Description

The provided address contains two contracts in its code, `XNAPToken` and `LiquidityManager`. However only the `XNAPToken` contract is deployed . As a result all the existing functionality of the `LiquidityManager` is not usable. Furthermore, some of the findings of this report were downgraded due to the code being unusable. Specifically, the findings `CPV`, `MEV`, `ODM`, `PAIM`, `POSD` and `UTPD` are downgraded to informational due to the unusable nature of the code. Deploying the contract without addressing these findings will raise their severity.

```Shell
contract LiquidityManager is Ownable, ReentrancyGuard
```

Recommendation

The team is advised to deploy the `LiquidityManager` contract separately. Additionally, the team should follow the recommendations of the aforementioned findings to ensure the safety and security of the contract.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	XNAPToken.sol#L290,295,302,309,314
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 setTimelockDelay(uint256 delay_) external
onlyOwner
function queueAction(bytes32 id) external onlyOwner
function cancelAction(bytes32 id) external onlyOwner
function pause() external onlyOwner

function unpause() external onlyOwner
```

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.

CPV - Cross Pool Vulnerability

Criticality	Minor / Informative
Location	XNAPToken.sol#L437,484
Status	Unresolved

Description

The contract provides liquidity management functions, including adding and removing liquidity. If the supported token is tradable across multiple pools (e.g., different stablecoins), this creates a potential arbitrage vulnerability.

An attacker could initiate a flash loan from one pool and then add or remove liquidity in another. By swapping tokens between pools after altering liquidity, the attacker can obtain surplus tokens, repay the flash loan, and retain the excess, effectively extracting value from the contract.

Shell
```
function addLiquidityETH(uint amountTokenDesired, uint
amountTokenMin, uint amountETHMin, uint deadline, bytes32
actionId) external payable onlyQueued(actionId)
whenNotPaused nonReentrant returns (uint amountToken, uint
amountETH, uint liquidity)


function withdrawLP(address to, uint amount, bytes32
actionId) external onlyQueued(actionId) whenNotPaused
nonReentrant
```

Recommendation

The team is advised to implement proper controls over token pool configuration. If multiple pools exist, cross-pool operations may create arbitrage opportunities.

IUT - Inconsistent Unlockable Time

Criticality	Minor / Informative
Location	XNAPToken.sol#L494
Status	Unresolved

Description

The `withdrawLP` function relies on a single global `unlockTime` and only checks whether the current timestamp is greater than this value. Since `unlockTime` can be modified by the owner, it can be set to a past or near-future timestamp, allowing the owner to bypass the intended lock period and withdraw LP tokens immediately. In addition, the contract does not track lock information for each liquidity deposit. All deposits are governed by the same global `unlockTime`, which means that every LP token becomes unlocked at the same time, and any tokens deposited after the lock has already expired can be withdrawn without any delay. This design does not provide proper, deposit-specific lockup enforcement.

Shell
```
function withdrawLP(address to, uint amount, bytes32
actionId) external onlyQueued(actionId)whenNotPaused
nonReentrant {
    require(block.timestamp >= unlockTime, "LP locked");
    ...

}
```

Recommendation

The team should ensure that each liquidity addition has its own release time. Additionally, 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.

MEV - Miner Extraction Value

Criticality	Minor / Informative
Location	XNAPToken.sol#L437,484
Status	Unresolved

Description

The contract's liquidity management functions, addLiquidityETH and withdrawLP, are susceptible to Miner Extractable Value (MEV). A miner or bot monitoring the mempool can order transactions within a block to extract value from users. This is commonly known as a "sandwich attack."

An attacker can precede the contract's call to `addLiquidityETH` or `withdrawLP` with a swap operation that significantly skews the current reserves of the pool. The contract's transaction will then execute, updating the pool's liquidity at this altered ratio. The attacker can then immediately swap back in a subsequent transaction, capturing a profit. This profit is derived directly from the contract's balance.

Shell
```
function addLiquidityETH(uint amountTokenDesired, uint
amountTokenMin, uint amountETHMin, uint deadline, bytes32
actionId) external payable onlyQueued(actionId)
whenNotPaused nonReentrant returns (uint amountToken, uint
amountETH, uint liquidity)


function withdrawLP(address to, uint amount, bytes32
actionId) external onlyQueued(actionId) whenNotPaused
nonReentrant
```

Recommendation

The team is advised to implement strict slippage protection on all functions that are sensitive to pool ratios. Setting the minimum `amountTokenMin` and `amountETHMin` to proper value will enhance the overall security and resistance against Miner Extracted Value.

MC - Missing Check

Criticality	Minor / Informative
Location	XNAPToken.sol#L321
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 team does not ensure that `feed` on the `setPriceFeed` function is not the `address(0)`

Shell
```
function setPriceFeed(address feed) external
onlyQueued(keccak256(abi.encodePacked("setPriceFeed",
feed))) {
    priceFeed = feed;

}
```

Additionally, in `addLiquidityETH` checks are missing to ensure that the amounts and the deadline are reasonable values.

Shell
```
function addLiquidityETH(uint amountTokenDesired, uint
amountTokenMin, uint amountETHMin, uint deadline, bytes32
actionId)
```

Recommendation

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

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	XNAPToken.sol#L290,320,324,348
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

Shell
```
function setTimelockDelay(uint256 delay_) external
onlyOwner
function setPriceFeed(address feed) external onlyQueued
function setMaxPriceDeviationBP(uint256 bp) external
onlyQueued

function approveToken(uint amount) external
onlyQueued(keccak256(abi.encodePacked("approveToken",
amount)))
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

ODM - Oracle Decimal Mismatch

Criticality	Minor / Informative
Location	XNAPToken.sol#L385,399
Status	Unresolved

Description

The contract relies on data retrieved from an external Oracle to make critical calculations. However, the contract does not include a verification step to align the decimal precision of the retrieved data with the precision expected by the contract's internal calculations. This mismatch in decimal precision can introduce substantial errors in calculations involving decimal values.

Specifically, the contract does not ensure that oracle price has the same decimal precision as the `pairPrice1e18` . This will result in huge differences between the `oraclePrice` and `pairPrice1e18` . Any function that uses `_checkOracleAndPairPrice` will revert because of the decimal inconsistency.

Shell
```
function getOraclePrice() public view returns (uint256
price, uint256 updatedAt, bool ok) {
    if (priceFeed == address(0)) return (0, 0, false);
    try AggregatorV3Interface(priceFeed).latestRoundData()
returns (uint80, int256 answer, uint256, uint256 uAt,
uint80) {
        if (answer <= 0) return (0, 0, false);
        price = uint256(answer);
        updatedAt = uAt;
        return (price, updatedAt, true);
    } catch {
        return (0, 0, false);
    }
}

function pricesWithinTolerance(uint256 pairPrice1e18,
uint256 oraclePrice, uint256 bpTolerance) internal pure
returns (bool) {
    if (oraclePrice == 0) return true;
    uint256 diff = pairPrice1e18 > oraclePrice ?
pairPrice1e18 - oraclePrice : oraclePrice - pairPrice1e18;
    return (diff * 10000) / oraclePrice <= bpTolerance;
}


function _checkOracleAndPairPrice() internal view
```

Recommendation

The team is advised to retrieve the decimals precision from the Oracle API in order to proceed with the appropriate adjustments to the internal decimals representation.

PAIM - Possible Action ID Misuse

Criticality	Minor / Informative
Location	XNAPToken.sol#L446,490,511,519
Status	Unresolved

Description

The contract is using the modifier onlyQueued to ensure a transaction is accepted by the owner. The `onlyQueued` modifier accepts an `id` as input and checks whether or not the id is a queued action. However, there are several functions in the contract that accept `actionId` as input from users. This allows users to pass any `actionId` and as long as it is queued it will be accepted. For example the users can use the id from any setter function to manage liquidity or transfer funds.

Shell
```
function addLiquidityETH(uint amountTokenDesired, uint
amountTokenMin, uint amountETHMin, uint deadline, bytes32
actionId) external payable onlyQueued(actionId)
whenNotPaused nonReentrant returns (uint amountToken, uint
amountETH, uint liquidity)

function withdrawLP(address to, uint amount, bytes32
actionId) external onlyQueued(actionId) whenNotPaused
nonReentrant

function recoverERC20(address erc20, uint amount, address
to, bytes32 actionId) external onlyQueued(actionId)
whenNotPaused nonReentrant


function recoverETH(uint amount, address payable to,

bytes32 actionId) external onlyQueued(actionId)

whenNotPaused nonReentrant
```

Recommendation

The team should enforce strict restrictions to the `actionId` to ensure that each id can be used only in their appropriate function.

PDEW - Possible Double Execution Window

Criticality	Minor / Informative
Location	XNAPToken.sol#L272
Status	Unresolved

Description

The contract implements the modifier onlyQueued that checks if an action should be executed. This partially depends on the timestamp of the function call and the time it was queued. If an action is already executed, a cooldown must pass in order to be used again. However, if the cooldown window is small, it is possible that a queued action can be executed multiple times in the same `GRACE_PERIOD` , as long as the owner enables it.

Shell
```
modifier onlyQueued(bytes32 id) {
        uint256 eta = queuedActions[id];
        require(eta != 0, "not queued");
        require(block.timestamp >= eta, "timelock: not
ready");
        require(block.timestamp <= eta + GRACE_PERIOD,
"timelock: expired");
        uint256 last = lastExecuted[id];
        require(last == 0 || block.timestamp >= last +
actionCooldown, "action in cooldown");

        ...
```

Recommendation

The team should review the finding and evaluate whether it describes a desired architecture. If the description mentioned is not desirable, the team should enforce restrictions to ensure that `actionCooldown` is never less than the `GRACE_PERIOD` .

POSD - Potential Oracle Stale Data

Criticality	Minor / Informative
Location	XNAPToken.sol#L419
Status	Unresolved

Description

The contract relies on retrieving price data from an oracle. However, it lacks proper checks to ensure the data is not stale. The absence of these checks can result in outdated price data being trusted, potentially leading to significant financial inaccuracies.

Specifically, during the `_checkOracleAndPairPrice` function the contract checks if the pair price calculation and oracle price calculation returned `true`. If not the timestamp checks are bypassed.

```
Shell
if (okPair && okOracle) {
    require(updatedAt != 0 && block.timestamp - updatedAt
<= oracleStaleThreshold, "oracle stale");
    require(pricesWithinTolerance(pairPrice1e18,
oraclePrice, maxPriceDeviationBP), "price divergence");

}
```

Recommendation

To mitigate the risk of using stale data, it is recommended to implement checks on the round and period values returned by the oracle's data retrieval function. The value indicating the most recent round or version of the data should confirm that the data is current. Additionally, the time at which the data was last updated should be checked against the current interval to ensure the data is fresh. For example, consider defining a threshold value, where if the difference between the current period and the data's last update period exceeds this threshold, the data should be considered stale and discarded, raising an appropriate error.

For contracts deployed on Layer-2 solutions, an additional check should be added to verify the sequencer's uptime. This involves integrating a boolean check to confirm the sequencer is operational before utilizing oracle data. This ensures that during sequencer downtimes, any transactions relying on oracle data are reverted, preventing the use of outdated and potentially harmful data.

By incorporating these checks, the smart contract can ensure the reliability and accuracy of the price data it uses, safeguarding against potential financial discrepancies and enhancing overall security.

PRE - Potential Reentrance Exploit

Criticality	Minor / Informative
Location	XNAPToken.sol#L272
Status	Unresolved

Description

The `onlyQueued` modifier alters the contract's state after the function call terminates. This contradicts best security practices and may introduce vulnerabilities as state is updated after external interactions. Such re-entrant architectures could be used by a malicious user to perform unauthorized actions.

Shell
```
modifier onlyQueued(bytes32 id) {
    ...
    _;
    lastExecuted[id] = block.timestamp;
    delete queuedActions[id];
    emit ActionExecuted(id, block.timestamp);

}
```

Recommendation

The team is advised to follow the checks-effects-interactions pattern to ensure that the contract cannot be exploited and unauthorized actions cannot be used.

RSD - Redundant Struct Declaration

Criticality	Minor / Informative
Location	XNAPToken.sol#L429
Status	Unresolved

Description

The contract has a `struct` called `AddParams`. It uses this `struct` in the `AddLiquidityEth` to store the caller's input. However the function could just use the input directly to proceed in the rest of the functionality, therefore the `struct` and its use is redundant.

Shell
```
struct AddParams {
    uint amountTokenDesired;
    uint amountTokenMin;
    uint amountETHMin;
    uint deadline;

}
```

Recommendation

The team should remove redundancies in order to make the code more optimal and readable.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	XNAPToken.sol#L453
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

Specifically, in the function `addLiquidityETH` the contract expects that its token balance is sufficient in order to proceed to rest of the functionality but the tokens need to be sent externally.

Shell
```
function addLiquidityETH(uint amountTokenDesired, uint
amountTokenMin, uint amountETHMin, uint deadline, bytes32
actionId) external payable onlyQueued(actionId)
whenNotPaused nonReentrant returns (uint amountToken, uint
amountETH, uint liquidity){
    ...
    uint tokenBal =
IERC20(token).balanceOf(address(this));
    require(tokenBal >= amountTokenDesired, "insufficient
token balance");
    ...

}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

UTPD - Unverified Third Party Dependencies

Criticality	Minor / Informative
Location	XNAPToken.sol#L387
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
```
try AggregatorV3Interface(priceFeed).latestRoundData()
returns (uint80, int256 answer, uint256, uint256 uAt,
uint80)
```

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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	XNAPToken.sol#L15,217,341
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
```
function WETH() external view returns (address);
address public immutable WETH

uint256 _unlockTime
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	XNAPToken.sol#L321
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

Shell
```
priceFeed = feed
```

Recommendation

By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	XNAPToken.sol#L6
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

Shell
```
pragma solidity ^0.8.24;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

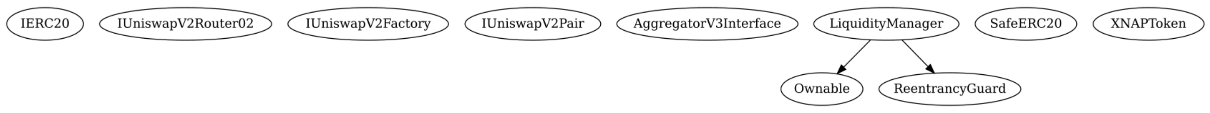
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	approve	External	✓	-
	transferFrom	External	✓	-
	transfer	External	✓	-
	balanceOf	External		-
	decimals	External		-
IUniswapV2Router02	Interface			
	factory	External		-
	WETH	External		-
	addLiquidityETH	External	Payable	-
IUniswapV2Factory	Interface			
	getPair	External		-
IUniswapV2Pair	Interface			
	balanceOf	External		-
	transfer	External	✓	-
	getReserves	External		-
	token0	External		-

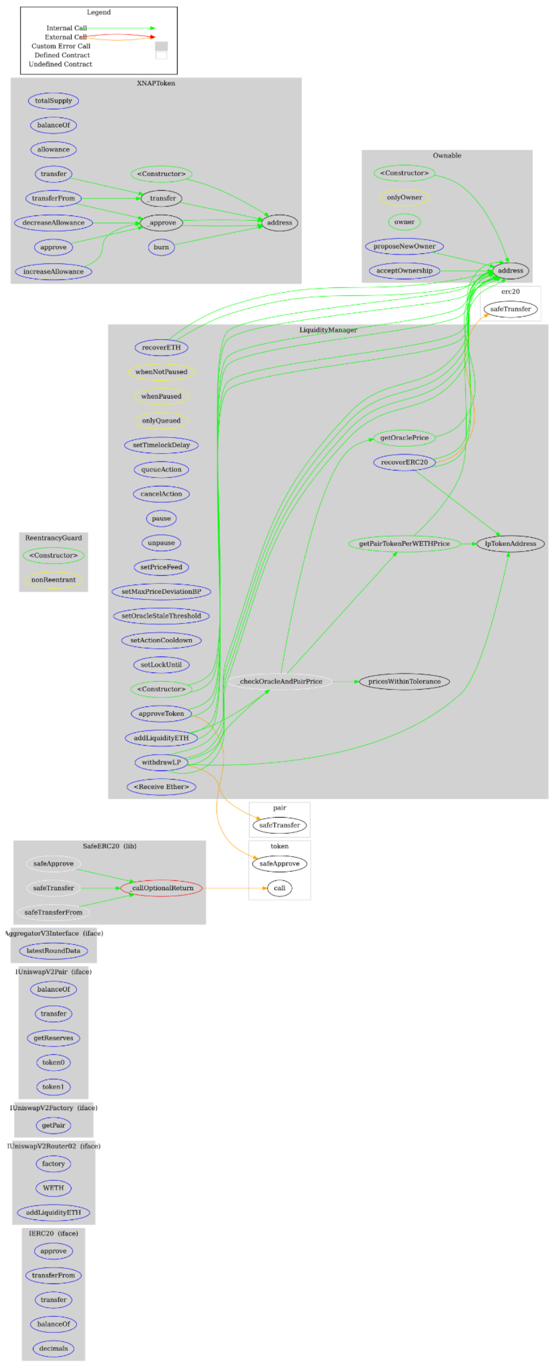
	token1	External		-
AggregatorV3Interface	Interface			
	latestRoundData	External		-
Ownable	Implementation			
		Public	✓	-
	owner	Public		-
	proposeNewOwner	External	✓	onlyOwner
	acceptOwnership	External	✓	-
ReentrancyGuard	Implementation			
		Public	✓	-
SafeERC20	Library			
	_callOptionalReturn	Private	✓	
	safeTransfer	Internal	✓	
	safeTransferFrom	Internal	✓	
	safeApprove	Internal	✓	
LiquidityManager	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	setTimelockDelay	External	✓	onlyOwner

	queueAction	External	✓	onlyOwner
	cancelAction	External	✓	onlyOwner
	pause	External	✓	onlyOwner
	unpause	External	✓	onlyOwner whenPaused
	setPriceFeed	External	✓	onlyQueued
	setMaxPriceDeviationBP	External	✓	onlyQueued
	setOracleStaleThreshold	External	✓	onlyQueued
	setActionCooldown	External	✓	onlyQueued
	setLockUntil	External	✓	onlyQueued
	approveToken	External	✓	onlyQueued
	lpTokenAddress	Public		-
	getPairTokenPerWETHPrice	Public		-
	getOraclePrice	Public		-
	pricesWithinTolerance	Internal		
	_checkOracleAndPairPrice	Internal		
	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
XNAPToken
totalSupply
balanceOf
allowance
transfer
transferFrom
decreaseAllowance
approve
increaseAllowance
<Constructor>
transfer
approve
burn
address
Ownable
<Constructor>
onlyOwner
owner
proposeNewOwner
acceptOwnership
address
erc20
safeTransfer
LiquidityManager
recoverETH
whenNotPaused
whenPaused
onlyQueued
setTimelockDelay
queueAction
cancelAction
pause
unpause
setPriceFeed
setMaxPriceDeviationBP
setOracleStaleThreshold
setActionCooldown
setLockUntil
<Constructor>
approveToken
addLiquidityETH
withdrawLP
<Receive Ether>
getOraclePrice
recoverERC20
getPairTokenPerWETHPrice
lpTokenAddress
checkOracleAndPairPrice
pricesWithinTolerance
ReentrancyGuard
<Constructor>
nonReentrant
pair
safeTransfer
token
safeApprove
call
SafeERC20 (lib)
safeApprove
safeTransfer
safeTransferFrom
callOptionalReturn
AggregatorV3Interface (iface)
latestRoundData
IUniswapV2Pair (iface)
balanceOf
transfer
getReserves
token0
token1
IUniswapV2Factory (iface)
getPair
IUniswapV2Router02 (iface)
factory
WETH
addLiquidityETH
IERC20 (iface)
approve
transferFrom
transfer
balanceOf
decimals

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