Smart Contract Audit Report for Oracle and WCP Contract of CpChain

Singapore

20nd August, 2025

Content

About TheWeb3

TheWeb3 is a professional community focused on Web3 technology solutions and blockchain security research, committed to providing individuals and enterprises with end-to-end services in education, development, security auditing, and venture support.

With expertise across major blockchain ecosystems such as EVM, Solana, and Bitcoin, TheWeb3 delivers comprehensive solutions spanning Layer 1 and Layer 2 protocols, secure wallets, cross-chain bridges, zero-knowledge circuits, cryptographic applications, and web platforms.

TheWeb3 is structured around three core pillars:

- **TheWeb3 Community**: Dedicated to Web3 education and talent development, having successfully trained over 200 skilled developers and more than 60 outstanding product managers, contributing significantly to the ecosystem's talent pool.
- **TheWeb3 Security Lab**: Specializes in blockchain vulnerability research and advanced security auditing. The lab has served renowned projects like DappLink, FishCake, Parapack, and RootHash. With a team grounded in cryptography, mobile security, low-level exploitation, financial systems, and traditional infosec, TheWeb3 excels at uncovering deep, hidden vulnerabilities and pioneering novel security methodologies.
- **TheWeb3 Capital**: With over $2 million in cumulative investments, The Web3 actively incubates promising Web3 startups, providing the capital and strategic support needed to scale technical innovation into viable ventures.

By integrating robust technical expertise with a holistic ecosystem approach, TheWeb3 is becoming the go-to security and innovation partner for high-growth teams operating at the forefront of Web3.

1. Execution Order

1.1 Properties

The summary of this smart contract audit is as follows.

Client	CPChain
Title	Smart Contract Audit Report for Oracle and WCP Contract
Platform	EVM-compatible
Version	1.0
Auditors	Leo, Kaso
Reviewed by	Felix
Approved by	Seek
Classification	Public

1.2 Scope

This audit covers the following versions of code.

Source - Github	Commit Hash
https://github.com/cpchain-network/cp-chain/blob/main/packages/contracts-cpchain/src/universal/WrappedCP.sol	b8bc9be105367553fe1611a7f6aaa8aa64b13e85
https://github.com/cpchain-network/oracle-contracts	cd9c237a83a31c608958f35aed6ae9d524829ca5

1.3 Contact

For more information about this document and its contents, please contact The Web3 Security Labs

Name	Email
Felix	felix.deng129@gmail.com
Anita	anita52062@gmail.com

2. Overview

2.1 Executive Summary

During August 20, 2025, We conducted a comprehensive security evaluation of the CP Chain infrastructure. Our technical audit encompassed a thorough examination of the WCP and Oracle smart contract architecture, with particular attention to potential security vulnerabilities, architectural design considerations, and systemic weaknesses in the defensive posture of the implementation.

The assessment methodology employed a multi-layered analytical approach, evaluating both the codebase integrity and the broader security implications within the decentralized ecosystem. Our expert team deployed advanced vulnerability detection techniques to identify potential attack vectors and security concerns within the protocol's operational framework.

2.2 Results

This is a decentralized oracle network system built with Solidity smart contracts. At its core, it features an OracleManager contract that manages multiple oracle nodes (OraclePods), using BLS signatures for secure price updates. The system uses the BN254 elliptic curve for cryptographic operations and implements upgradeable patterns for future updates. Key components include operator management, price feed updates, and whitelist controls, all secured by BLS signature verification.

This audit found 6 security issues, including 1 high-risk vulnerabilities, 2 medium-risk issues, and 3 low-risk issues.

BreakDown of Finding Impacts

Impact Level	Count	Status
High	2	2 Fixed
Medium	3	3 Fixed
Low	6	6 Fixed

3. Introduction

3.1 About Oracle and WCP contracts

The Oracle Network is an advanced decentralized oracle solution designed to revolutionize price data feeds in Web3 applications. Built on a robust foundation of BLS signature technology and blockchain security, this project offers a secure, transparent, and efficient way to aggregate and verify market prices. By leveraging the power of BLS signatures, this project ensures cryptographic integrity of price updates while enabling multiple oracle nodes to operate in a trustless environment.

This project's upgradeable architecture ensures future compatibility and security improvements, while its whitelist management system maintains network integrity. The Oracle Network empowers developers to integrate reliable price feeds into their decentralized applications without intermediaries, fostering transparency and trust in the DeFi ecosystem. With features like batch price updates, secure signature verification, and operator management, this project sets a new standard for oracle reliability and security in the Web3 space.

3.2 Methodology

During security assessments, We employs a systematic security auditing process that combines automated tools and in-depth manual review. While our processes are tailored to each project's specific requirements, the core always remains a comprehensive professional review of the entire codebase.

In addition to utilizing various advanced tools and analyzers, We focuses on the following categories of security and reliability issues:

Code Foundation Vulnerabilities: Many critical security incidents throughout history stem from simple surface-level errors that could have been detected through rigorous code review. We deploy advanced tools such as fuzzing and formal verification based on project needs, while also conducting preliminary code familiarization to ensure comprehensive understanding of contract functionality.

Business Logic Flaws: The core of smart contracts lies in their business logic. We examine specifications and designs for inconsistencies, vulnerabilities, and weaknesses that may create exploitation opportunities, such as unrealistic tokenomics or dangerous arbitrage possibilities. When conditions allow, we also verify whether contract logic aligns with the intended functionality outlined in design documents.

Ecosystem Interaction Risks: Most significant vulnerabilities arise not from contract defects themselves, but from unexpected consequences of interactions with the DeFi ecosystem. We review external dependencies and interactions, identifying risk points such as flash loan attack

possibilities, oracle manipulation risks, MEV attacks, and other potential threats.

Code Quality and Best Practices: We assess the overall maturity of the codebase, looking for violations of industry standards and best practices, and provide optimization recommendations, including gas optimization, upgradability enhancements, and decentralization improvements.

For each finding, We assigns an impact rating based on its severity and likelihood of occurrence. We use the following impact levels (ordered by importance): Critical, High, Medium, Low, and Advisory.

We structures reports based on importance rather than strictly by impact rating. We consider client threat models, business requirements, and long-term goals to provide practical and implementable recommendations, not just a simple list of security issues.

Finally, we offer a series of observations that may not directly relate to security but benefit code quality and future project development. We is committed to being your security partner, not just an audit service pro.

4. Detail Findings

4.1 [H-01]: Missing Oracle Heartbeat [FIXED]

Level: High

Issue Location: src/base/OraclePod.sol

Issue Description

The Oracle system lacks a **heartbeat mechanism**, unable to ensure data timeliness. Compared to mainstream Oracles like **Chainlink** and **Redstone**, this represents a functional deficiency that affects data credibility.

Impact Analysis

This may result in inaccurate or untimely quotations obtained by third-party platforms due to untimely data updates and inability to detect them.

Fix Recommendation

Implement heartbeat mechanism and timestamp recording.

```
uint256 public updateTimestamp;
uint256 public constant maxAge = 1 days;

function fillSymbolPrice(uint256 price) external onlyOracleManager {
    uint256 oldPrice = marketPrice;
    marketPrice = price;
    updateTimestamp = block.timestamp;
    emit MarketPriceUpdated(oldPrice, price);
  }

  function isDataFresh(uint256 maxAge) external view returns (bool) {
      return block.timestamp - updateTimestamp <= maxAge;
  }
```

4.2 [H-02]: Incorrect calculation of signerApk in the function checkSignatures() [FIXED]

Level: High

Issue Location: bls/BLSApkRegistry.sol #124-149

Issue Description

The function checkSignatures() in the contract BLSApkRegistry contains a critical flaw in its calculation of the aggregate public key (signerApk). Currently, the function attempts to compute the signer's aggregate key by subtracting non-signers' public keys from currentApk. However, the implementation erroneously overwrites signerApk in each iteration of the loop, resulting in only the last non-signer's key

being subtracted. This incorrect calculation leads to faulty signature verification and may cause DoS.

```
   function checkSignatures(bytes32 msgHash, uint256 referenceBlockNumber,
VrfNoSignerAndSignature memory params) public view returns (StakeTotals memory,
bytes32) {
       require(
           referenceBlockNumber < uint32(block.number),
"BLSSignatureChecker.checkSignatures: invalid reference block"
       );
       BN254.G1Point memory signerApk = BN254.G1Point(0, 0);
       bytes32[] memory nonSignersPubkeyHashes;
       if (params.nonSignerPubKeys.length > 0) {
           nonSignersPubkeyHashes = new
bytes32[](params.nonSignerPubKeys.length);
           for (uint256 j = 0; j < params.nonSignerPubKeys.length; j++) {
               nonSignersPubkeyHashes[j] =
params.nonSignerPubKeys[j].hashG1Point();
               signerApk = currentApk.plus(params.nonSignerPubKeys[j].negate());
           }
       } else {
           signerApk = currentApk;
       }
       (bool pairingSuccessful, bool signatureIsValid) =
trySignatureAndApkVerification(msgHash, signerApk, params.apkG2, params.sigma);

       require(pairingSuccessful, "BLSSignatureChecker.checkSignatures: pairing
precompile call failed");
       require(signatureIsValid, "BLSSignatureChecker.checkSignatures: signature
is invalid");

       bytes32 signatoryRecordHash =
keccak256(abi.encodePacked(referenceBlockNumber, nonSignersPubkeyHashes));

       StakeTotals memory stakeTotals = StakeTotals({totalDappLinkStake:
params.totalDappLinkStake, totalBtcStake: params.totalBtcStake});

       return (stakeTotals, signatoryRecordHash);
   }
```

Impact Analysis

Potential DoS due to the incorrect update for the variable signerApk.

Fix Recommendation

Revise the code logic accordingly.

```
  uint256 public updateTimestamp;
  uint256 public constant maxAge = 1 days;

  function fillSymbolPrice(uint256 price) external onlyOracleManager {
      uint256 oldPrice = marketPrice;
```

```
        marketPrice = price;
        updateTimestamp = block.timestamp;
        emit MarketPriceUpdated(oldPrice, price);
    }

    function isDataFresh(uint256 maxAge) external view returns (bool) {
        return block.timestamp - updateTimestamp <= maxAge;
    }
```

4.3 [M-01]: Redundant Ownership Transfer Call [FIXED]

Level: Medium

Issue Location

src/base/OraclePod.sol#24, src/bls/BLSApkRegistry.sol#43,

src/core/OracleManager.sol#40

Issue Description

Redundant `_transferOwnership` calls in constructors cause excessive OwnershipTransferred event triggers, generating redundant data that affects monitoring system judgment.

```
_transferOwnership(_initialOwner); // Redundant call
```

Impact Analysis

May cause abnormal owner change event triggering.

Fix Recommendation

Remove redundant **_transferOwnership** calls.

4.4 [M-02]: Lack of validation for minimum valid signatures threshold [FIXED]

Level: Medium

Issue Location

src/bls/BLSApkRegistry.sol #124-149

Issue Description

The function checkSignatures() lacks validation for the minimum number of required valid signatures, which poses risks of insufficient authorization. The function may accept messages signed by fewer signers than required.

```
    function checkSignatures(bytes32 msgHash, uint256 referenceBlockNumber,
VrfNoSignerAndSignature memory params) public view returns (StakeTotals memory,
bytes32) {
        require(
            referenceBlockNumber < uint32(block.number),
"BLSSignatureChecker.checkSignatures: invalid reference block"
        );
        BN254.G1Point memory signerApk = BN254.G1Point(0, 0);
        bytes32[] memory nonSignersPubkeyHashes;
        if (params.nonSignerPubKeys.length > 0) {
            nonSignersPubkeyHashes = new
bytes32[](params.nonSignerPubKeys.length);
            for (uint256 j = 0; j < params.nonSignerPubKeys.length; j++) {
                nonSignersPubkeyHashes[j] =
params.nonSignerPubKeys[j].hashG1Point();
                signerApk = currentApk.plus(params.nonSignerPubKeys[j].negate());
            }
        } else {
            signerApk = currentApk;
        }
        (bool pairingSuccessful, bool signatureIsValid) =
trySignatureAndApkVerification(msgHash, signerApk, params.apkG2, params.sigma);

        require(pairingSuccessful, "BLSSignatureChecker.checkSignatures: pairing
precompile call failed");
        require(signatureIsValid, "BLSSignatureChecker.checkSignatures: signature
is invalid");

        bytes32 signatoryRecordHash =
keccak256(abi.encodePacked(referenceBlockNumber, nonSignersPubkeyHashes));

        StakeTotals memory stakeTotals = StakeTotals({totalDappLinkStake:
params.totalDappLinkStake, totalBtcStake: params.totalBtcStake});

        return (stakeTotals, signatoryRecordHash);
    }
```

Impact Analysis

Setting the finalization period seconds for outputRoot could be approved without sufficient consensus.

Fix Recommendation

Implement checks for the minimum number of required valid signatures.

4.5 [M-03]: oracleManager management failure [FIXED]

Level: Medium

Issue Location: `src/base/OraclePod.sol`

Issue Description

The **oracleManager** address in the OraclePod contract cannot be changed after initialization. If the OracleManager contract needs to be upgraded or migrated to a new address, all existing OraclePods will permanently point to the old, possibly deprecated address, rendering them useless as they will no longer receive price updates.

Impact Analysis

The owner cannot effectively manage the oracleManager. If the oracleManager is stolen, unexpected situations will occur.

Fix Recommendation:

Add a function setOracleManager in OraclePod.sol that can only be called by the owner to allow the oracleManager address to be updated when necessary.

```
  event OracleManagerUpdated(address indexed previousManager, address
indexed newManager);

  function updateOracleManager(address newManager) external onlyOwner {
   oracleManager = newManager;
   emit OracleManagerUpdated(msg.sender, newManager);
}
```

4.6 [L-01]: SPDX-License-Identifier Error [FIXED]

Level: Low

Issue Location

src/universal/WrappedCP.sol#1, src/core/token/WCP.sol#1

Issue Description

SPDX-License-Identifier is inconsistent. Since the contract will be connected to multiple third-party dapps, it is recommended to adopt the more flexible MIT protocol with fewer constraints on the three parties.

Impact Analysis

Inconsistent certificates make it difficult to distinguish commercial products and affect the professionalism of the project. In addition, GPL-3.0 does not apply to protocols like WETH.

```
// SPDX-License-Identifier: GPL-3.0
// SPDX-License-Identifier: MIT
```

Fix Recommendation

Use MIT protocol uniformly.

4.7 [L-02]: Improper use of version numbers [FIXED]

Level: Low

Issue Location

src/core/token/WCP.sol#222, src/bls/BLSApkRegistry.sol#34

Issue Description

Here, 0.0.1 is used as the version number, while 0.0.x is usually used as a patch rather than a formal production version. Irregular usage of versions may cause difficulties in later version management and lose the practical meaning of this parameter.

Impact Analysis

Non-standard version numbers may lead to difficulties in version number management and errors in upgrades later.

```
string public constant version = "0.0.1";

constructor() EIP712("BLSApkRegistry", "v0.0.1") {
  _disableInitializers();
}
```

Fix Recommendation

The deployment version is set to version 1.0.0, which represents the official version.

```
string public constant version = "1.0.0";

constructor() EIP712("BLSApkRegistry", "v1.0.0") {
  _disableInitializers();
}
```

4.8 [L-03]: Missing Price Update Events [FIXED]

Level: Low

Issue Location: **src/base/OraclePod.sol#29**

Issue Description

The price update function **does not emit events**, preventing off-chain monitoring systems from timely detection of price changes, affecting system observability.

```
function fillSymbolPrice(uint256 price) external onlyOracleManager {
  uint256 previousPrice = marketPrice;
  marketPrice = price;
```

```
    // Missing event emission
  }
```

Impact Analysis

If there is no relevant event configuration, it will be very difficult to capture price updates and the price cannot be effectively monitored, which will be a potential risk point for the oracle.

Fix Recommendation

Modify the code and add the `MarketPriceChanged` event.

```
  event MarketPriceChanged(uint256 previousPrice, uint256 newPrice,
uint256 timestamp);

  function fillSymbolPrice(uint256 price) external onlyOracleManager {
    uint256 previousPrice = marketPrice;
    marketPrice = price;
    emit MarketPriceChanged(previousPrice, price, block.timestamp);
  }
```

4.9 [L-04]: Potential duplicate node registration leading to incorrect currentApk calculation [FIXED]

Level: Low

Issue Location: src/bls/BLSApkRegistry.sol #52-58

Issue Description

The function registerOperator() in the contract BLSApkRegistry lacks protection against duplicate registrations of the same node (operator address). Each registration adds the operator's public key to the current aggregated public key (i.e., currentApk) without checking whether the operator was previously registered. This could lead to incorrect currentApk calculation, due to the same public key being counted multiple times in currentApk, distorting the true aggregate value.

```
  function registerOperator(address operator) public onlyVrfManager {
      (BN254.G1Point memory pubkey,) = getRegisteredPubkey(operator);

      _processApkUpdate(pubkey);

      emit OperatorAdded(operator, operatorToPubkeyHash[operator]);
  }
```

Impact Analysis

This could cause miscalculations in BLS signature verification that rely on accurate currentApk values.

Fix Recommendation

Implement a mapping to enforce one-time registration.

4.10 [L-05]: Lack of checks for the registration status of nonSignerPubkeys [FIXED]

Level: Low

Issue Location: src/bls/BLSApkRegistry.sol #130-138

Issue Description

The current implementation assumes that all nodes corresponding to nonSignerPubkeys have been properly registered in the contract BLSApkRegistry. However, there is no validation to confirm this registration status. If a node operator fails to register via registerOperator() but is still included in nonSignerPubkeys, it will lead to incorrect signerApk calculations. This occurs because the currentApk does not actually contain the unregistered nonSignerPubkeys components. This oversight could lead to incorrect signerApk calculation, which will finally cause DoS.

```
        if (params.nonSignerPubKeys.length > 0) {
            nonSignersPubkeyHashes = new
bytes32[](params.nonSignerPubKeys.length);
            for (uint256 j = 0; j < params.nonSignerPubKeys.length; j++) {
                nonSignersPubkeyHashes[j] =
params.nonSignerPubKeys[j].hashG1Point();
                signerApk =
currentApk.plus(params.nonSignerPubKeys[j].negate());
            }
        } else {
            signerApk = currentApk;
        }
```

Impact Analysis

This oversight could lead to incorrect signerApk calculation, which will finally cause DoS.

Fix Recommendation

Revise the logic accordingly.

4.11 [L-06]: Lack of checks for the parameter pubkeyRegistrationMessageHash [FIXED]

Level: Low

Issue Location: src/bls/BLSApkRegistry.sol #71-126

Issue Description

The function registerBLSPublicKey() does not check whether the msg.sender is the operator. A malicious node1 can generate valid parameters and register it for another node2, then operatorToPubkeyHash[node2] will be set to a public key by node1, thus preventing node2 from registering its own public key.

And the function registerBLSPublicKey() uses pubkeyRegistrationMessageHash as an input parameter, but fails to check whether this hash was properly generated by the function getPubkeyRegMessageHash() based on the operator's address. A malicious node can directly use parameters signed by someone else (e.g., obtained from the historical registration information of an EigenLayer node), and it can also satisfy the BLS pairing verification.

```
    function registerBLSPublicKey(
        address operator,
        PubkeyRegistrationParams calldata params,
        BN254.G1Point calldata pubkeyRegistrationMessageHash
    ) external returns (bytes32) {
        require(
            blsRegisterWhitelist[msg.sender],
            "BLSApkRegistry.registerBLSPublicKey: this address have not permission
to register bls key"
        );

        bytes32 pubkeyHash = BN254.hashG1Point(params.pubkeyG1);

        require(pubkeyHash != ZERO_PK_HASH, "BLSApkRegistry.registerBLSPublicKey:
cannot register zero pubkey");
        require(
            operatorToPubkeyHash[operator] == bytes32(0),
            "BLSApkRegistry.registerBLSPublicKey: operator already registered
pubkey"
        );

        require(
            pubkeyHashToOperator[pubkeyHash] == address(0),
            "BLSApkRegistry.registerBLSPublicKey: public key already registered"
        );

        uint256 gamma = uint256(
            keccak256(
                abi.encodePacked(
                    params.pubkeyRegistrationSignature.X,
                    params.pubkeyRegistrationSignature.Y,
                    params.pubkeyG1.X,
                    params.pubkeyG1.Y,
                    params.pubkeyG2.X,
                    params.pubkeyG2.Y,
                    pubkeyRegistrationMessageHash.X,
                    pubkeyRegistrationMessageHash.Y
                )
            )
        ) % BN254.FR_MODULUS;
```

```
        require(
            BN254.pairing(

params.pubkeyRegistrationSignature.plus(params.pubkeyG1.scalar_mul(gamma)),
                BN254.negGeneratorG2(),

pubkeyRegistrationMessageHash.plus(BN254.generatorG1().scalar_mul(gamma)),
                params.pubkeyG2
            ),
            "BLSApkRegistry.registerBLSPublicKey: either the G1 signature is
wrong, or G1 and G2 private key do not match"
        );

        operatorToPubkey[operator] = params.pubkeyG1;
        operatorToPubkeyHash[operator] = pubkeyHash;
        pubkeyHashToOperator[pubkeyHash] = operator;

        emit NewPubkeyRegistration(operator, params.pubkeyG1, params.pubkeyG2);

        return pubkeyHash;
    }
```

Impact Analysis

This oversight creates a potential vulnerability where malicious actors could register public keys by providing a crafted pubkeyRegistrationMessageHash.

Fix Recommendation

Add the check logic accordingly.

```
  event MarketPriceChanged(uint256 previousPrice, uint256 newPrice,
uint256 timestamp);

  function fillSymbolPrice(uint256 price) external onlyOracleManager {
    uint256 previousPrice = marketPrice;
    marketPrice = price;
    emit MarketPriceChanged(previousPrice, price, block.timestamp);
  }
```

5. Conclusion

The WCP and Oracle contract is relatively complete in its functionality, but contains several security issues that need to be addressed. **We strongly recommend the following actions before mainnet deployment**:

1. **Prioritize fixing all high-risk vulnerabilities**, especially the signature replay attack and reward calculation logic flaws
2. **Address medium-risk issues** to improve contract robustness and user experience
3. **Consider low-risk issues** and optimization suggestions to enhance code quality and maintainability
4. **Conduct another security audit** after fixes to ensure all issues have been resolved
5. Consider **performing formal penetration testing** before deployment to verify contract security in a real environment

The contract design is fundamentally sound and, once the above issues are fixed, will safely meet its design objectives.

6. Disclaimers

This audit report is based on the provided code version and does not guarantee discovery of all potential issues. Smart contract security is affected by multiple factors, including but not limited to blockchain security itself, external contract interactions, deployment environment, etc. We recommend conducting multiple rounds of audits and testing before formal deployment, and continuous monitoring of contract operations.