AstraSec

BitCorn TSS Network Security Audit Report

November 22, 2024

Contents

1 Introduction

1.1 About Bitcorn TSS Network

1.2 Source Code

2 Overall Assessment

3 Vulnerability Summary

3.1 Overview

3.2 Security Level Reference

3.3 Methodology

3.4 Vulnerability Details

4 Appendix

4.1 About AstraSec

4.2 Disclaimer

4.3 Contact

1 Introduction

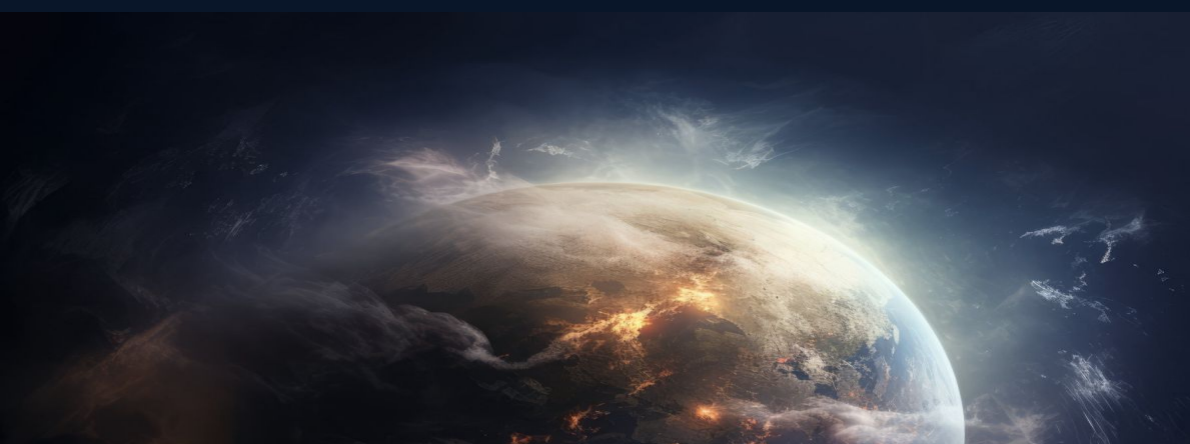
1.1 About BitCorn

BitCorn is a BTC staking protocol that allows users to stake BTC on Babylon and receive native LST, coBTC in return. Users can then deploy their coBTC in additional DeFi activities to earn amplified yields whilst the underlying collateral accumulates rewards on Babylon.

1.2 About MPC TSS

The MPC (Multi-Party Computation) TSS Network code is a Rust implementation of the Threshold Signature Scheme (TSS) for distributed key management and signing operations. It serves as the backbone of the multi-party network that manages the BTC wallet for staking, acts as the admin/owner of the ERC-20 LST, and performs unbonding transactions upon user requests.

Please note that the security of the backend project depends on multiple factors, such as code quality, Operations, Administration, and Maintenance (OAM). However, this audit focuses exclusively on the implementation of BitCorn's TSS code.



1.2 Source Code

The following source code was reviewed during the audit:

- https://github.com/kernelprotocol/MPC/tree/master/src
- CommitID: dbe9a8c

And this is the final version representing all fixes implemented for the issues identified in the audit:

- https://github.com/kernelprotocol/MPC/tree/master/src
- CommitID: 2c4d848

2 Overall Assessment

This report has been compiled to identify issues and vulnerabilities within the Bitcorn MPC TSS project. Throughout this audit, we identified a total of 9 issues spanning various severity levels. By employing auxiliary tool techniques to supplement our thorough manual code review, we have discovered the following findings.

Severity	Count	Acknowledged	Won't Do	Addressed
Critical	–	–	–	–
High	3	–	–	3
Medium	3	–	–	3
Low	3	–	1	2
Informational	-	–	–	–
Undetermined	–	–	–	–

3 Vulnerability Summary

3.1 Overview

Click on an issue to jump to it, or scroll down to see them all.

~~H-1~~ Incomplete Authentication Implementation

~~H-2~~ Lack of API Function Classification

~~H-3~~ Improved Nonce Usage in aes_encrypt()

~~M-1~~ Lack of User Input Validation

~~M-2~~ Lack of User Input Validation (Cont)

~~M-3~~ Improved Validation of User Input

~~L-1~~ Separation of Signers on Different Machines

L-2 Known Vulnerability in curve25519-dalek Library

~~L-3~~ Removal of Dependency on Rust Nightly Release

3.2 Security Level Reference

In web3 smart contract audits, vulnerabilities are typically classified into different severity levels based on the potential impact they can have on the security and functionality of the contract. Here are the definitions for critical-severity, high-severity, medium-severity, and low-severity vulnerabilities:

Severity	Acknowledged
C-X (Critical)	A severe security flaw with immediate and significant negative consequences. It poses high risks, such as unauthorized access, financial losses, or complete disruption of functionality. Requires immediate attention and remediation.
H-X (High)	Significant security issues that can lead to substantial risks. Although not as severe as critical vulnerabilities, they can still result in unauthorized access, manipulation of contract state, or financial losses. Prompt remediation is necessary.
M-X (Medium)	Moderately impactful security weaknesses that require attention and re-mediation. They may lead to limited unauthorized access, minor financial losses, or potential disruptions to functionality.
L-X (Low)	Minor security issues with limited impact. While they may not pose significant risks, it is still recommended to address them to maintain a robust and secure smart contract.
I-X (Informational)	Warnings and things to keep in mind when operating the protocol. No immediate action required.
U-X (Undetermined)	Identified security flaw requiring further investigation. Severity and impact need to be determined. Additional assessment and analysis are necessary.

3.3 Methodology

In this code audit, code review were conducted simultaneously by different security experts. Below is a detailed breakdown of the areas we assessed.

3.3.1 Static Code Review

Manual Review:

- Inspect unsafe blocks to ensure their necessity and correct implementation.
- Verify proper error handling using Result and Option types.
- Ensure APIs enforce appropriate authentication and authorization mechanisms.

Automated Tools:

- Use security tools to scan for vulnerable dependencies.

3.3.2 Security Assessment

Memory Safety:

- Identify improper use of unsafe code and potential memory leaks.
- Ensure correct ownership and lifetimes to prevent dangling references.

Concurrency Safety:

- Verify async/await code to ensure non-blocking behavior and avoid deadlocks.
- Ensure shared data is handled safely to prevent race conditions.

Cryptographic Practices:

- Use trusted cryptographic libraries for secure operations.
- Verify key management practices and secure handling of sensitive data.

API Authentication:

- Validate JWT token usage, including expiration checks and claim validation.
- Ensure proper input sanitization to prevent SQL/NoSQL injections.

3.3.3 Business Logic and API Validation

- Validate inputs to ensure they follow expected formats (e.g., UUIDs).
- Enforce payload size limits to prevent denial-of-service (DoS) attacks.
- Ensure appropriate HTTP status codes are used for error handling.
- Threshold Signature Scheme (TSS):
    - Verify that signer nodes are distributed across independent machines.
    - Ensure multiple signer tasks are not redundantly executed.

3.3.4 Performance and Optimization Review

- Identify blocking I/O operations and handle them appropriately with asynchronous tasks.
- Optimize memory usage to prevent leaks and excessive consumption.
- Implement caching mechanisms where appropriate to minimize redundant computations.

3.3.5 Penetration Testing

- Conduct penetration testing to identify potential vulnerabilities and security gaps.
- Simulate attacks, including injection, privilege escalation, and DoS, to evaluate system resilience.
- After addressing the findings, reassess the effectiveness of authentication, access control, and input validation mechanisms.

3.4 Vulnerability Details

3.4.1 [H-1] Incomplete Authentication Implementation

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
auth.rs	Security	High	High	Addressed

In auth.rs, it integrates JWT-based authentication into a Rocket web server, with functionality for creating, validating, and extracting authentication tokens (JWTs).

However, upon reviewing its implementation, we notice the from_request() method does not validate the token or extract the user's identity (e.g., user_id). It simply checks for the presence of an Authorization header (line 94). This leaves the authentication incomplete, allowing anyone with a token to pass through without validation.

MPC - auth.rs

```
88  #[rocket::async_trait]
89  impl<'r> FromRequest<'r> for AuthenticatedUser {
90      ...
91      async fn from_request(request: &'r Request<'_>) -> Outcome<Self, Self::Error> {
92          // For now, we'll just check if an "Authorization" header is present
93          // In a real application, you'd verify the token here
94          match request.headers().get_one("Authorization") {
95              Some(_) => Outcome::Success(AuthenticatedUser {}),
96              None => Outcome::Error((Status::Unauthorized, ())),
97          }
98      }
99  }
```

Remediation Add token and user validation logic in this function.

3.4.2 [H-2] Lack of API Function Classification

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
manager.rs	Security	High	High	Addressed

The manager plays a central role in coordinating and managing the different services and tasks related to a Multi-Party Computation Threshold Signature Scheme (MPC TSS) using Rocket as a web framework. Rocket is used to expose APIs for managing and coordinating the TSS.

We notice the current /sign and /get_signing_result endpoints are publicly accessible and no authentication mechanism (e.g., JWT) is implemented. The other four APIs (/signup_sign, /set, /get, /update_signing_result) are intended for internal use by the signer and there is no restricted access, such as IP whitelisting in place to enhance security.

MPC - manager.rs

```
50  let rocket_future = rocket::custom(config)
51      .manage(manager_service_for_rocket)
52      .mount(
53          "/",
54          routes![
55              sign,
56              signup_sign,
57              set,
58              get,
59              get_signing_result,
60              update_signing_result
61          ],
62      )
63      .launch();
```

Remediation Implement authentication mechanism and restricted access for the APIs as mentioned.

3.4.3 [H-3] Improved Nonce Usage in aes_encrypt()

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
utils.rs	Security	High	High	Addressed

The aes_encrypt() function performs AES-256-GCM encryption. A nonce (initialization vector) is required for AES-GCM mode. While examine its logic, we notice a 12-byte array of zeros is used as the nonce (line 18). A constant nonce ([0u8; 12]) should not be used repeatedly, as it compromises security. It should be random for each encryption and is not the same as an authentication tag.

Using a static nonce (all zeros) for every encryption is insecure and should be avoided in practice. Typically, nonces should be random and unique for each encryption to ensure security.

MPC - utils.rs

```
12  pub fn aes_encrypt(key: &[u8], plaintext: &[u8]) -> AEAD {
13      let mut key_sized = [0u8; 32];
14      key_sized[..key.len()].copy_from_slice(key);
15      let aes_key = aes_gcm::Key::from_slice(&key_sized);
16      let cipher = Aes256Gcm::new(aes_key);
17
18      let nonce = Nonce::from_slice(&[0u8; 12]);
19      let ciphertext = cipher.encrypt(nonce, plaintext).unwrap();
20
21      AEAD {
22          ciphertext,
23          tag: nonce.to_vec(),
24      }
25  }
```

Remediation User random Nonce value in this function.

3.4.4 [M-1] Lack of User Input Validation

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
manager::api.rs	Security	High	Low	Addressed

The manager::api.rs defines APIs for managing and processing signing requests in the TSS (Multi-Party Computation Threshold Signature Scheme) project. The /sign endpoint receives a signing request from a client, processes it, and returns a response indicating that the signing process has started.

However, SigningRequestDTO (line 28) accepts arbitrary strings as input for the message to be signed and there is no validation for the input. Imposing a size limit on the HTTP POST body is crucial to prevent resource exhaustion and Denial-of-Service.

MPC - api.rs

```
25  #[post("/sign", format = "json", data = "<request>")]
26  pub async fn sign(
27      manager: &State<Arc<ManagerService>>,
28      request: Json<SigningRequestDTO>,
29  ) -> Result<Created<Json<SigningResponseDTO>>, Status> {
30      let message: Vec<u8> = request.message.as_bytes().to_vec();
31
32      let signing_request = SigningRequest {
33          id: uuid::Uuid::new_v4().to_string(),
34          message,
35      };
```

Remediation Suggest to address it through configuration in either of the following ways:

- **Reverse proxy** (if one is placed in front of your service)
- **Rocket web framework** configuration

3.4.5 [M-2] Lack of User Input Validation (Cont)

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
storage::mongodb.rs	Security	High	Low	Addressed

The storage::mongodb.rs implements a MongoDB-based storage system for managing signing requests and their associated results in the TSS (Multi-Party Computation Threshold Signature Scheme) project. The insert_request() method inserts a new signing request into the MongoDB collection.

We notice that there is no input validation for the size of request.message. To prevent large data from being written to MongoDB, it's essential to implement size checks before insertion.

MPC - mongodb.rs

```
21  pub async fn insert_request(&self, request: &SigningRequest) -> Result<()> {
22      let message_to_sign = MessageToSignStored {
23          request_id: request.id.clone(),
24          message: request.message.clone(),
25          status: MessageStatus::Pending,
26          signature: None,
27      };
28      self.requests.insert_one(message_to_sign, None).await?;
29      Ok(())
30  }
```

Remediation Check the size of the request.message before writing to the database.

3.4.6 [M-3] Improved Validation of User Input

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
mongodb.rs	Security	High	Low	Addressed

The get_signing_result function() is responsible for retrieving the result of a signing request by its unique request_id. To prevent **NoSQL** injection, it is essential to validate and sanitize user input. Specifically:

- Ensure the input is a valid UUID before sending it in queries to MongoDB.
- Sanitize input to remove any potentially malicious content.
- Prefer using parameterized queries or ORM frameworks to minimize injection risks.

This approach ensures that only properly formatted, valid data is sent to the database, reducing the chance of injection attacks.

MPC - mongodb.rs

```rust
pub async fn get_signing_result(&self, id: &str) -> Result<Option<MessageToSignStored>> {
    let filter = doc! { "request_id": id };
    if let Some(doc) = self.requests.find_one(filter, None).await? {
        Ok(Some(doc))
    } else {
        Ok(None)
    }
}
```

Remediation Apply the input validation as suggestion above.

3.4.7 [L-1] Separation of Signers on Different Machines

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
bin::siger.rs	Business Logic	High	Low	Addressed

The singer.rs started three SignerService tasks in an MPC (Multi-Party Computation) environment. Each signer service is responsible for handling cryptographic operations as part of a Threshold Signature Scheme (TSS). The bin::signer.rs need to be modified to run one signer task with only one key file.

It is intended to run on separate machines and be owned by different parties to make sure the security of the signer process. In the context of TSS (Threshold Signature Scheme), running multiple signer tasks on the same machine, as shown in the code, violates the purpose of the distributed architecture.

MPC - signer.rs

```rust
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    ...
    let signer1_task = task::spawn(async move {
        ...
    });

    let signer2_task = task::spawn(async move {
        ...
    });

    let signer3_task = task::spawn(async move {
        ...
    });

    let _ = tokio::join!(signer1_task, signer2_task, signer3_task);

    Ok(())
}
```

Remediation Make sure to run signer tasks on different machines in the production environment.

3.4.8 [L-2] Known Vulnerability in curve25519-dalek Library

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Cargo.toml	Security	Medium	Low	won't do

There is a vulnerability in the curve25519-dalek library, specifically within the Scalar29::sub (32-bit) and Scalar52::sub (64-bit) functions. These functions were found to have timing variability due to a mask value used inside a loop. The LLVM compiler inserted a branch instruction (jns on x86) that conditionally bypasses the loop when the mask value is zero. This introduced timing variability that could potentially leak private information, such as elliptic curve scalar values used in cryptographic operations, making the system vulnerable to side-channel attacks.

Please refer to: https://rustsec.org/advisories/RUSTSEC-2024-0344

MPC - Cargo.lock

```
920  [[package]]
921  name = "curve25519-dalek"
922  version = "3.2.0"
923  source = "registry+https://github.com/rust-lang/crates.io-index"
924  checksum = "0b9fdf9972b2bd6af2d913799d9ebc165ea4d2e65878e329d9c6b372c4491b61"
925  dependencies = [
926   "byteorder",
927   "digest 0.9.0",
928   "rand_core 0.5.1",
929   "subtle",
930   "zeroize",
931  ]
```

Remediation Upgrade curve25519-dalek to version that is >=4.1.3.

Response By Team curve25519-dalek is not a direct dependency of the codebase. It is an indirect dependency to curv-kzen, but the ed25519 and curve_ristretto modules aren't used since we only use secp256k1.

3.4.9 [L-3] Removal of Dependency on Rust Nightly Release

TARGET	CATEGORY	IMPACT	LIKELIHOOD	STATUS
Compile tool chain	Security	Low	Low	Addressed

Currently the project uses Rust nightly build for compiling for specific feature. It is recommended to transition to stable Rust by modifying or removing the #![feature(...)] attribute. Relying on nightly Rust can introduce instability and complicate maintenance, whereas using the stable release ensures better compatibility and long-term support.

Remediation Use Rust stable release instead of nightly release.

4 Appendix

4.1 About AstraSec

AstraSec is a blockchain security company that serves to provide high-quality auditing services for blockchain-based protocols. With a team of blockchain specialists, AstraSec maintains a strong commitment to excellence and client satisfaction. The audit team members have extensive audit experience for various famous DeFi projects. AstraSec's comprehensive approach and deep blockchain understanding make it a trusted partner for the clients.

4.2 Disclaimer

The information provided in this audit report is for reference only and does not constitute any legal, financial, or investment advice. Any views, suggestions, or conclusions in the audit report are based on the limited information and conditions obtained during the audit process and may be subject to unknown risks and uncertainties. While we make every effort to ensure the accuracy and completeness of the audit report, we are not responsible for any errors or omissions in the report.

We recommend users to carefully consider the information in the audit report based on their own independent judgment and professional advice before making any decisions. We are not responsible for the consequences of the use of the audit report, including but not limited to any losses or damages resulting from reliance on the audit report.

This audit report is for reference only and should not be considered a substitute for legal documents or contracts.

4.3 Contact

Phone	+86 156 0639 2692
Email	contact@astrasec.ai
Twitter	https://twitter.com/AstraSecAI