FYEO

Security Code Review of Censo Vault iOS

Censo

December 2023
Version 1.0

Presented by:
FYEO Inc.
PO Box 147044
Lakewood CO 80214
United States

Security Level
Public

Table of Contents

EXECUTIVE SUMMARY

Overview

Censo engaged FYEO Inc. to perform a Security Code Review of Censo Vault iOS.

The assessment was conducted remotely by the FYEO Security Team. Testing took place on November 07 - December 05, 2023, and focused on the following objectives:

- To provide the customer with an assessment of their overall security posture and any risks that were discovered within the environment during the engagement.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the results of our tests.

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the FYEO Security Team took to identify and validate each issue, as well as any applicable recommendations for remediation.

Key Findings

The following issues have been identified during the testing period. These should be prioritized for remediation to reduce the risk they pose:

- FYEO-CENSO-IOS-01 – Sha2 function used to derive encryption key from key material.
- FYEO-CENSO-IOS-02 – Base58 and Base32 implementations not covered in tests
- FYEO-CENSO-IOS-03 – Custom implementation of TOTP generator
- FYEO-CENSO-IOS-04 – Custom implementation of Shamir's Secret Sharing Scheme

Based on our review process, we conclude that the reviewed code implements the documented functionality.

Scope and Rules of Engagement

The FYEO Review Team performed a Security Code Review of Censo Vault iOS. The following table documents the targets in scope for the engagement. No additional systems or resources were in scope for this assessment.

The source code was supplied through a private repository at https://github.com/Censo-Inc/vault-ios with the commit hash 7c825ff8b3996fa873eaaeeb1f32f678ef2f61c2.

Files included in the code review

```
vault-ios/
├── Approver/
│   ├── Helpers/
│   │   └── AppSpecificError.swift
│   ├── Views/
│   │   ├── Onboarding/
│   │   │   ├── AcceptInvitation.swift
│   │   │   ├── Onboarding.swift
│   │   │   ├── SubmitVerification.swift
│   │   │   └── VerificationCodeEntry.swift
│   │   ├── PinInputField/
│   │   │   ├── PinInput.swift
│   │   │   └── PinInputField.swift
│   │   ├── Recovery/
│   │   │   ├── OwnerVerification.swift
│   │   │   ├── RecoveryApproval.swift
│   │   │   ├── RecoveryApprovalComplete.swift
│   │   │   └── StartRecoveryApproval.swift
│   │   ├── ApproverHome.swift
│   │   ├── ApproverHomeInvitation.swift
│   │   ├── ApproverRouting.swift
│   │   ├── ContentView.swift
│   │   └── Login.swift
│   ├── API.swift
│   ├── ApproverApp.swift
│   └── ParticipantId+PrivateKeyStorage.swift
└── Censo/
    ├── API/
    │   ├── API.swift
    │   ├── APIProvider.swift
    │   ├── AuthPlugin.swift
    │   ├── ErrorResponsePlugin.swift
    │   ├── Models.swift
    │   ├── MoyaProvider+Decodable.swift
    │   ├── MoyaProvider+Session.swift
    │   └── OwnerState.swift
    ├── Helpers/
    │   ├── AppSpecificError.swift
    │   ├── BIP39.swift
    │   ├── Base32.swift
    │   ├── Base58.swift
    │   ├── Data+BigInt.swift
    │   ├── Data+Hex.swift
```

Files included in the code review

```
│   ├── RandomUtils.swift
│   ├── String+Hex.swift
│   ├── SymmetricEncryption.swift
│   ├── TotpUtils.swift
│   └── UIColor+Hex.swift
├── KeyManagement/
│   ├── DeviceKey.swift
│   ├── EncryptionKey+Owner.swift
│   ├── EncryptionKey.swift
│   ├── SecureEnclaveKey.swift
│   ├── SecureEnclaveWrapper.swift
│   └── SigningKey.swift
├── Models/
│   ├── DecodedPhrase.swift
│   ├── Error.swift
│   ├── InvitationId.swift
│   ├── Phrase.swift
│   ├── Session.swift
│   ├── UnlockedDuration.swift
│   ├── ValueWrappers.swift
│   └── Vault.swift
├── Recovery/
│   ├── BigInt+Recovery.swift
│   └── SecretSharer.swift
├── Views/
│   ├── ButtonStyles/
│   │   ├── BorderedButtonStyle.swift
│   │   ├── ButtonStyleTint.swift
│   │   ├── FilledButtonStyle.swift
│   │   └── RoundedButtonStyle.swift
│   ├── Censo/
│   │   ├── Access/
│   │   │   ├── AccessApproval.swift
│   │   │   ├── AccessApproved.swift
│   │   │   ├── AccessExpirationCountdown.swift
│   │   │   ├── AccessIntro.swift
│   │   │   ├── AccessPhrases.swift
│   │   │   ├── ChooseAccessApprover.swift
│   │   │   ├── EnterAccessVerificationCode.swift
│   │   │   ├── ShowPhrase.swift
│   │   │   └── ShowPhraseList.swift
│   │   ├── ApproversSetup/
│   │   │   ├── ActivateApprover.swift
```

Files included in the code review

```
│   │   │   ├── ApproversSetup.swift
│   │   │   ├── EnterApproverNickname.swift
│   │   │   ├── GetLiveWithApprover.swift
│   │   │   ├── ProposeToAddAlternateApprover.swift
│   │   │   ├── RenameApprover.swift
│   │   │   ├── SavedAndSharded.swift
│   │   │   └── SetupApprover.swift
│   │   ├── Home/
│   │   │   ├── ApproversView.swift
│   │   │   ├── CensoHomeScreen.swift
│   │   │   ├── HomeView.swift
│   │   │   ├── PhrasesView.swift
│   │   │   └── SettingsView.swift
│   │   ├── SecretsManagement/
│   │   │   ├── AdditionalPhrase.swift
│   │   │   ├── PastePhrase.swift
│   │   │   ├── PhraseSaveSuccess.swift
│   │   │   ├── SaveSeedPhrase.swift
│   │   │   ├── SecretsListView.swift
│   │   │   ├── SeedEntry.swift
│   │   │   ├── SeedVerification.swift
│   │   │   ├── Word.swift
│   │   │   ├── WordEntry.swift
│   │   │   └── WordList.swift
│   │   ├── BiometryGatedScreen.swift
│   │   └── LockScreen.swift
│   ├── Facetec/
│   │   ├── FacetecAuth.swift
│   │   ├── FacetecError.swift
│   │   ├── FacetecUIKitWrapper.swift
│   │   └── IdentityVerification.swift
│   ├── Onboarding/
│   │   ├── FirstPhrase.swift
│   │   ├── InitialApproversSetup.swift
│   │   ├── InitialPlanSetup.swift
│   │   └── Welcome.swift
│   ├── Shared/
│   │   ├── AppleSignIn.swift
│   │   ├── ApproverPill.swift
│   │   ├── LoginBottomLinks.swift
│   │   └── OperationCompletedView.swift
│   ├── TextFieldStyles/
│   │   └── RoundedTextFieldStyle.swift
```

Files included in the code review
│ ├── Authentication.swift │ ├── BackButton.swift │ ├── CloudCheck.swift │ ├── ContentView.swift │ ├── Login.swift │ ├── Owner.swift │ ├── RetryView.swift │ ├── RotatingTotpPinView.swift │ ├── SignIn.swift │ └── TermsOfUse.swift ├── CensoApp.swift ├── CensoTheme.swift ├── Configuration.swift ├── Keychain.swift ├── RemoteResult.swift └── UserCredentials.swift

Table 1: Scope

TECHNICAL ANALYSES AND FINDINGS

During the Security Code Review of Censo Vault iOS, we discovered:

- 1 finding with MEDIUM severity rating.
- 2 findings with LOW severity rating.
- 1 finding with INFORMATIONAL severity rating.

The following chart displays the findings by severity.

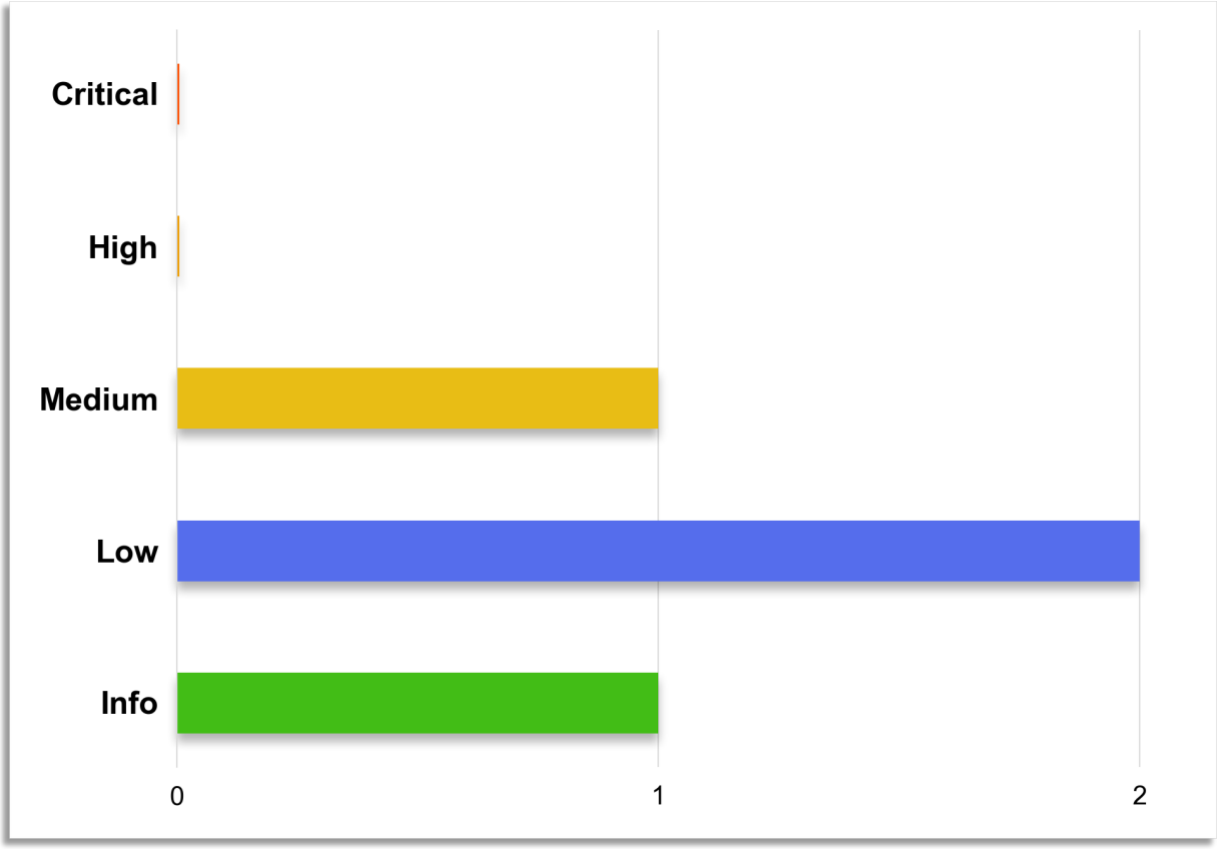


Figure 1: Findings by Severity

FINDINGS

The *Findings* section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

Finding #	Severity	Description
FYEO-CENSO-IOS-01	**Medium**	Sha2 function used to derive encryption key from key material.
FYEO-CENSO-IOS-02	**Low**	Base58 and Base32 implementations not covered in tests
FYEO-CENSO-IOS-03	**Low**	Custom implementation of TOTP generator
FYEO-CENSO-IOS-04	**Informational**	Custom implementation of Shamir's Secret Sharing Scheme

Table 2: Findings Overview

TECHNICAL ANALYSIS

The source code has been manually validated to the extent that the state of the repository allowed. The validation includes confirming that the code correctly implements the intended functionality.

CONCLUSION

Based on our review process, we conclude that the code implements the documented functionality to the extent of the reviewed code.

TECHNICAL FINDINGS

GENERAL OBSERVATIONS

This code review is part of the security audit of the Censo Seed Phrase Manager system.

The main functionality of the system is to provide the ability to securely store a seed phrase for crypto wallets. The system allows multiple participants to be responsible for storing a seed phrase. In particular, the supported schemes are:

- 1:1 - a scheme where the owner of the seed phrase acts as an approver,
- 2:2 - a scheme with the owner and one approver, and
- 2:3 - a scheme with the owner and two approvers, in which confirmation of one of the approvers is sufficient.

This audit covers the Censo and Approver iOS applications, which are the components of the Censo Seed Phrase Manager system. Censo is the main application for storing seed phrase. Approver iOS application is an application for verifiers.

Applications are developed using Swift for iOS16.4 and higher, to incorporate security improvements of the platform. However, this will limit the number of supported devices.

The audited code is complex and makes heavy use of the functionality in the hooks framework of setting foreign states of other hooks. This makes the code highly convoluted and more time-consuming to audit.

In our thorough examination of the code base, we dedicated significant attention to both logical and common errors.

The code is implementing the functionality as specified in the documentation and no real discrepancies from the documentation were identified during the audit.

During the audit, the development team worked diligently to remediate any significant findings identified by the FYEO team.

SHA2 FUNCTION USED TO DERIVE ENCRYPTION KEY FROM KEY MATERIAL

Finding ID: FYEO-CENSO-IOS-01
Severity: **Medium**
Status: **Remediated**

Description

The key to encrypt the approver (guardian) key is the sha256 hash of the sub field in the JWT token.

Using sha256() to directly generate a symmetric key is generally not recommended for secure key generation. While SHA-256 produces a fixed-length hash value from an input, it doesn't provide the necessary properties for generating secure symmetric keys. Keys generated in this way would not be truly random or unpredictable.

Proof of Issue

File name: vault-ios/Censo/KeyManagement/EncyptionKey.swift **Line number:** 52 - 72

```
    public func decrypt(base64EncodedString: Base64EncodedString) throws -> Data {
        let algorithm: SecKeyAlgorithm =
.eciesEncryptionCofactorVariableIVX963SHA256AESGCM

        guard SecKeyIsAlgorithmSupported(secKey, .decrypt, algorithm) else {
            throw SecKeyError.algorithmNotSupported
        }
```

Severity and Impact Summary

Symmetric key generation is simplified, which increases the chances of decrypting the Approver key.

Recommendation

We recommend reading the Recommendation for Cryptographic Key Generation and considering the possibility of generating a symmetric key through the KDF function. As a KDF, you may consider modern sponge-based functions, for example, the Keccak function family.

References

https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-133r2.pdf

https://keccak.team/obsolete/Keccak-main-2.1.pdf

BASE58 AND BASE32 IMPLEMENTATIONS NOT COVERED IN TESTS

Finding ID: FYEO-CENSO-IOS-02
Severity: **Low**
Status: **Remediated**

Description

Base58 and Base32 are implemented in source code of Censo app, not relying on third-party or integrated solution because of its absence. As these algorithms are used for key encoding in Policy and Recovery, primitives correct work should be covered by UT.

Proof of Issue

File name: vault-ios/Censo/Helpers/Base32.swift
Line number: 1 - 373

File name: vault-ios/Censo/Helpers/Base58.swift
Line number: 1 - 122

```
sample code
```

Severity and Impact Summary

Any unhandled issues in encoding primitives may cause problems on higher levels in classes using the primitive, which may lead to unexpected behavior.

Recommendation

Add UT modules in CensoTests for base58 and base32, verifying expected behavior: correct encoding/decoding, exception handling, and handling incorrect method arguments.

CUSTOM IMPLEMENTATION OF TOTP GENERATOR

Finding ID: FYEO-CENSO-IOS-03
Severity: **Low**
Status: **Open**

Description

A custom TOTP implementation is used to generate TOTP

Proof of Issue

File name: vault-ios/Censo/Helpers/TotpUtils.swift **Line number:** 11 - 56

```
struct TotpUtils {

    static let period = TimeInterval(60)
    static let digits = 6

    static func getOTP(date: Date, secret: Data) ->  String {
        var counter = UInt64(date.timeIntervalSince1970 / period).bigEndian
        // Generate the key based on the counter.
        let key = SymmetricKey(data: Data(bytes: &counter, count:
MemoryLayout.size(ofValue: counter)))
        let hash = HMAC<Insecure.SHA1>.authenticationCode(for: secret, using: key)

        var truncatedHash = hash.withUnsafeBytes { ptr -> UInt32 in
            let offset = ptr[hash.byteCount - 1] & 0x0f

            let truncatedHashPtr = ptr.baseAddress! + Int(offset)
            return truncatedHashPtr.bindMemory(to: UInt32.self, capacity: 1).pointee
        }

        truncatedHash = UInt32(bigEndian: truncatedHash)
        truncatedHash = truncatedHash & 0x7FFF_FFFF
        truncatedHash = truncatedHash % UInt32(pow(10, Float(digits)))

        return String(format: "%0*u", digits, truncatedHash)
    }

    static func getRemainingSeconds(date: Date) -> Int {
        let remainder = Int(UInt64(date.timeIntervalSince1970.rounded()) %
UInt64(period.rounded()))
        return Int(period.rounded()) - remainder
    }

    static func getPercentDone(date: Date) -> Double {
        let remainder = UInt64(date.timeIntervalSince1970.rounded()) %
UInt64(period.rounded())
        return Double(remainder) / period
    }

    static func signCode(code: String, signingKey: SigningKey) -> (UInt64,
Base64EncodedString)? {
        let timeMillis = UInt64(Date().timeIntervalSince1970 * 1000)
```

```
        guard let codeBytes = code.data(using: .utf8),
              let timeMillisData = String(timeMillis).data(using: .utf8),
              let signature = try? signingKey.signature(for: codeBytes +
timeMillisData) else {
            return nil
        }

        return (timeMillis, signature)
    }
}
```

Severity and Impact Summary

Custom implementation of security mechanisms carries potential system security risks. However in this case the development team chose to implement their own version to be able to guarantee compatibility between the android version and iOS version. The implemented version is minimalistic and follows the TOTP specification.

Recommendation

We suggest considering the possibility of using third-party libraries to generate TOTP. For example, SwiftOTP.

CUSTOM IMPLEMENTATION OF SHAMIR'S SECRET SHARING SCHEME

Finding ID: FYEO-CENSO-IOS-04
Severity: **Informational**
Status: **Open**

Description

The Censo application uses a custom implementation of Shamir's secret sharing scheme to split the Intermediate key and recover it.

Proof of Issue

File name: Censo/Recovery/SecretSharer.swift
Line number: 210-287

```
struct SecretSharer {
    var secret: BigInt
    var threshold: Int
    var participants: [BigInt]
    var order = ORDER
    var shards: [Point]

    init(
        secret: BigInt,
        threshold: Int,
        participants: [BigInt],
        order: BigInt = ORDER
    ) throws {
        self.secret = secret
        self.threshold = threshold
        self.participants = participants
        self.order = order
        self.shards = []
        self.shards = try getShares(participants: participants, threshold: threshold,
secret: secret)
    }

    func vandermonde(participants: [BigInt], threshold: Int) -> Matrix {
        SecretSharerUtils.vandermonde(participants: participants, threshold:
threshold, order: order)
    }

    func dotProduct(matrix: Matrix, vector: Vector) -> Vector {
        SecretSharerUtils.dotProduct(matrix: matrix, vector: vector, order: order)
    }

    func recoverSecret(shares: [Point]) -> BigInt {
        SecretSharerUtils.recoverSecret(shares: shares, order: order)
    }

    func addShares(shares: [BigInt], weights: [BigInt]) -> BigInt {
        SecretSharerUtils.addShares(shares: shares, weights: weights, order: order)
    }
```

```
    func decomposeLUP(matrix: Matrix) -> (Matrix, Vector) {
        SecretSharerUtils.decomposeLUP(matrix: matrix, order: order)
    }

    func invertLUP(lu: Matrix, p: Vector) -> Matrix {
        SecretSharerUtils.invertLUP(lu: lu, p: p, order: order)
    }

    enum ShareError: Error {
        case secretIrrecoverable
    }

    func getShares(participants: [BigInt], threshold: Int, secret: BigInt) throws ->
[Point] {
        guard threshold <= participants.count else {
            throw ShareError.secretIrrecoverable
        }

        var vec = Vector(repeating: .zero, count: threshold)

        for i in 0..<vec.count {
            if i == 0 {
                vec[i] = secret
            } else {
                vec[i] = SecretSharerUtils.randomFieldElement(order: order)
            }
        }

        let rowShares = dotProduct(matrix: vandermonde(participants: participants,
threshold: threshold), vector: vec)

        return participants.enumerated().map { i, p in
            Point(x: p, y: rowShares[i])
        }
    }

    func getReshares(newParticipants: [BigInt], newThreshold: Int) throws -> [[Point]]
{
        // generate the reshares from the first threshold participants
        try participants.prefix(threshold).indices.map { i in
            try getShares(participants: newParticipants, threshold: newThreshold,
secret: shards[i].y)
        }
    }
}
```

It is common practice to use well-established and widely accepted implementations of cryptographic algorithms. In the case of Shamir's scheme, such implementations are indeed not available for the iOS platform. However, custom implementation of cryptographic algorithms carries significant risks for the security of the system as a whole. In addition, cryptography is a rapidly evolving field. New attacks and vulnerabilities are discovered, and cryptographic standards are updated accordingly. Maintaining a custom cryptographic implementation requires constant vigilance to stay abreast of the latest developments, which can be a resource-intensive task.

Severity and Impact Summary

A custom implementation of the cryptographic algorithm is used to split and recover the Intermediate key. If there are vulnerabilities in this algorithm, the risk of system compromise increases significantly.

Recommendation

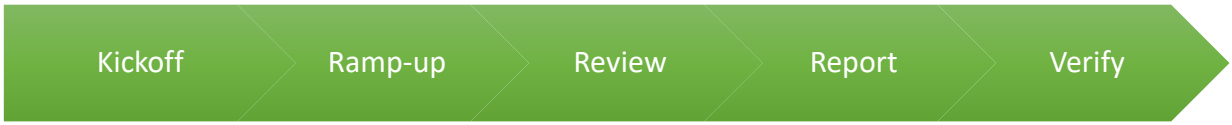
The audit scope did only include an analysis of the implementation of Shamir's secret sharing scheme as in the source code no audit of third party libraries and crypto primitives used were performed.

OUR PROCESS

METHODOLOGY

FYEO Inc. uses the following high-level methodology when approaching engagements. They are broken up into the following phases.

Figure 2: Methodology Flow



KICKOFF

The project is kicked off as the sales process has concluded. We typically set up a kickoff meeting where project stakeholders are gathered to discuss the project as well as the responsibilities of participants. During this meeting we verify the scope of the engagement and discuss the project activities. It's an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there is an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

RAMP-UP

Ramp-up consists of the activities necessary to gain proficiency on the project. This can include the steps needed for familiarity with the codebase or technological innovation utilized. This may include, but is not limited to:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for specific languages
- Researching common flaws and recent technological advancements

Review

The review phase is where most of the work on the engagement is completed. This is the phase where we analyze the project for flaws and issues that impact the security posture. Depending on the project this may include an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol
2. Review of the code written for the project
3. Compliance of the code with the provided technical documentation

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following sections.

Code Safety

We analyzed the provided code, checking for issues related to the following categories:

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- Error management and logging

This list is general and not comprehensive, meant only to give an understanding of the issues we are looking for.

Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

Reporting

FYEO Inc. delivers a draft report that contains an executive summary, technical details, and observations about the project.

The executive summary contains an overview of the engagement including the number of findings as well as a statement about our general risk assessment of the project. We may conclude that the overall risk is low but depending on what was assessed we may conclude that more scrutiny of the project is needed.

We report security issues identified, as well as informational findings for improvement, categorized by the following labels:

- Critical
- High
- Medium
- Low
- Informational

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we perform the audit, we may identify issues that aren't security related, but are general best practices and steps that can be taken to lower the attack surface of the project. We will call those out as we encounter them and as time permits.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

Verify

After the preliminary findings have been delivered, this could be in the form of the approved communication channel or delivery of the draft report, we will verify any fixes within a window of time specified in the project. After the fixes have been verified, we will change the status of the finding in the report from open to remediated.

The output of this phase will be a final report with any mitigated findings noted.

Additional Note

It is important to note that, although we did our best in our analysis, no code audit or assessment is a guarantee of the absence of flaws. Our effort was constrained by resource and time limits along with the scope of the agreement.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination.

THE CLASSIFICATION OF VULNERABILITIES

Security vulnerabilities and areas for improvement are weighted into one of several categories using, but is not limited to, the criteria listed below:

Critical – vulnerability will lead to a loss of protected assets

- This is a vulnerability that would lead to immediate loss of protected assets
- The complexity to exploit is low
- The probability of exploit is high

High - vulnerability has potential to lead to a loss of protected assets

- All discrepancies found where there is a security claim made in the documentation that cannot be found in the code
- All mismatches from the stated and actual functionality
- Unprotected key material
- Weak encryption of keys
- Badly generated key materials
- Txn signatures not verified
- Spending of funds through logic errors
- Calculation errors overflows and underflows

Medium - vulnerability hampers the uptime of the system or can lead to other problems

- Insecure calls to third party libraries
- Use of untested or nonstandard or non-peer-reviewed crypto functions
- Program crashes, leaves core dumps or writes sensitive data to log files

Low – vulnerability has a security impact but does not directly affect the protected assets

- Overly complex functions
- Unchecked return values from 3rd party libraries that could alter the execution flow

Informational

- General recommendations