SHELLBOXES

Kommunitas Official Telegram Bot

Smart Contract Security Audit

Prepared by ShellBoxes

Aug 5th, 2024 – Aug 6th, 2024

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	Kommunitas
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://github.com/Kommunitas-net/telegram-bot	2d8a729f557e07dcf38069ae0849e9c8c7ebab8f

Re-Audit

Repository	Commit Hash
https://github.com/Kommunitas-net/telegram-bot	df29ec7cadc2cdc9b2b6de7599ebded3a3645adc

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction

Kommunitas engaged ShellBoxes to conduct a security assessment on the Kommunitas Official Telegram Bot beginning on Aug 5th, 2024 and ending Aug 6th, 2024. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About Kommunitas

The Kommunitas Official Telegram Bot is an integral part of the Kommunitas ecosystem. It allows users to seamlessly interact with various platform features, including staking, voting, and participating in the launchpad, all from within the Telegram app. This bot offers a convenient and user-friendly way for the community to engage with the ecosystem, enhancing accessibility and user experience.

Issuer	Kommunitas
Website	`https://www.kommunitas.net`
Type	dApp
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low

Likelihood

2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the Kommunitas Official Telegram Bot implementation. During the first part of our audit, we examine the wallet/account source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and Wallet-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these react components are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include 1 critical-severity, 1 high-severity, 4 medium-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Incorrect Mnemonic Value Stored in HD Wallet Creation	CRITICAL	Fixed
SHB.2. Potential Infinite Loop in WalletsProvider Chains/Tokens Initialization	HIGH	Mitigated
SHB.3. Password Input Vulnerable to Browser Storage	MEDIUM	Fixed
SHB.4. Weak Password Validation	MEDIUM	Fixed
SHB.5. Importing Accounts from Mnemonic is Not Implemented	MEDIUM	Fixed
SHB.6. Lack of Two-Step Confirmation for Mnemonic	MEDIUM	Fixed

3 Finding Details

SHB.1 Incorrect Mnemonic Value Stored in HD Wallet Creation

- Severity : CRITICAL
- Status : Fixed
- Likelihood : 3
- Impact : 3

Description:

The createWallet function is responsible for creating an HD wallet from a provided mnemonic. However, in the current implementation, the mnemonic is incorrectly encrypted using the private key's value (encrypt(privateKey, password)), rather than using the correct encrypted value of the mnemonic itself (_mnemonic). This results in redundant storage of the private key's encrypted value and an incorrect mnemonic value. This error disrupts the logical handling of the mnemonic attribute, which is crucial for wallet recovery and import functionalities.

Files Affected:

SHB.1.1: index.tsx

```
/**
 * create HD wallet form mnemonic
 */
const createWallet = async () => {
    const { privateKey, address } = utils.HDNode.fromMnemonic(
        ↪ mnemonic);
    const _mnemonic = encrypt(mnemonic, password);
    const _wallet: WALLET = {
        name: "main",
        address: address,
        privateKey: encrypt(privateKey, password),
        mnemonic: encrypt(privateKey, password),
        type: WALLET_TYPES.MASTER,
        balance: 0
    };
    await setCurrentWallets([_wallet], _mnemonic);
    await setCurrentWallet(_wallet);
    setAddress(address);
    // notification
    showNotification("Your wallet has been imported successfully", "
        ↪ success");
    setStep(STEPPER.DONE);
}
```

Recommendation:

Modify the createWallet function to correctly encrypt and store the mnemonic value. Update the code to use mnemonic: _mnemonic instead of encrypt(privateKey, password). This change will ensure that the mnemonic is correctly encrypted and stored, preserving wallet recovery and import functionality.

Updates

The Kommunitas team has fixed this issue by removing the mnemonic attribute from the WALLET object. With this implementation, each wallet account is now linked to a specific privateKey and is characterized by its name and address.

SHB.2 Potential Infinite Loop in WalletsProvider Chains/Tokens Initialization

- Severity : HIGH
- Status : Mitigated

- Likelihood : 2
- Impact : 3

Description:

The init function in the walletsProvider has a potential for an infinite loop due to its error handling logic. When retrieving or storing chains and tokens from the LevelDB database, if an error occurs in the try block, the function attempts to handle the error by retrying the operation within the catch block. Specifically, the lines await _db.put('chains', chainsDefault) and await _db.put('tokens', INITIAL_TOKENS) are executed if an error is caught. Since these operations can fail under the same conditions that caused the initial error, this can lead to continuous retries without resolution, creating a potential infinite loop, consuming resources and potentially leading to application crashes or unresponsive behavior.

Files Affected:

SHB.2.1: walletsProvider.tsx

```
const init = async () => {
  const _db = new Level<string, any>('xkom', { valueEncoding: 'json' });

  try {
    setChains(chainsDefault);
    const _chains = await _db.get("chains");
    if (!_chains) {
      await _db.put('chains', chainsDefault);
      setChains(chainsDefault);
    } else {
      setChains(_chains);
    }
  } catch (err) {
    await _db.put('chains', chainsDefault);
    setChains(chainsDefault);
  }


  try {
    const _tokens = await _db.get("tokens");
    if (!_tokens) {
      await _db.put('tokens', INITIAL_TOKENS);
      setTokens(INITIAL_TOKENS);
    } else {
      setTokens(_tokens);
    }
  } catch (err) {
    await _db.put('tokens', INITIAL_TOKENS);
    setTokens(INITIAL_TOKENS);
  }
```

Recommendation:

Consider implementing a robust error handling to avoid potential infinite loops. Instead of retrying the same operation within the catch block, log the error and provide a fallback mechanism or user notification to handle the issue gracefully.

Updates

The Kommunitas team has mitigated the risk by adding a check in the catch block specifically for the error Entry not found. This prevents the retry loop from executing when the key is genuinely missing in the database. However, The team also invokes an asynchronous function to store chainsDefault and INITIAL_TOKENS in the database, which could still encounter errors. While the risk of an infinite loop has been significantly reduced, there remains a possibility of failure due to other exceptions.

SHB.3 Password Input Vulnerable to Browser Storage

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The application's password input field is susceptible to vulnerabilities related to browser storage mechanisms. When users enter sensitive information such as passwords, browsers like Firefox and Chromium may automatically save this data to disk to support features like session restoration or auto-fill. This can result in passwords being stored in plain text on the user's device. If an attacker gains physical or logical access to the device, they could retrieve these stored credentials and potentially gain unauthorized access to the user's account or wallet.

Exploit Scenario:

An attacker with physical or remote access to a user's device can exploit this vulnerability by accessing saved browser data, including passwords. This can occur if the user has not disabled the browser's password-saving feature or if the stored data is not adequately secured.

Files Affected:

SHB.3.1: passwordEditor.tsx

```
                <CustomInput
                    label='Password'
                    value={password}
                    setValue={setPassword}
                    placeholder='Input your password'
                    warning='input password'
                    valid={valid}
                    eye={true}
                />
```

Recommendation:

To mitigate this risk, ensure that the autocomplete attribute for password input fields is set to off to prevent browsers from storing the password.

Updates

The Kommunitas team has fixed this issue by adding the autoComplete=off attribute to the PasswordEditor input component. This change prevents browsers from storing or suggesting the password, thereby mitigating the risk of unintended password storage and enhancing the security of sensitive user input.

SHB.4 Weak Password Validation

- Severity : MEDIUM
- Status : Fixed

- Likelihood : 1
- Impact : 3

Description:

The current password validation for the wallet application only ensures that the password consists of letters and numbers and has a minimum length of greater than 6 characters. This level of validation is insufficient for wallet security, where a stronger password policy is necessary. Weak passwords that meet only basic criteria (letters and numbers with a minimum length) can be easily compromised through brute force or other attack methods. In the context of wallet security, where protection of sensitive financial data is critical, weak passwords pose a significant risk of unauthorized access and potential loss of funds.

Files Affected:

SHB.4.1: passwordEditor.tsx

```
const isValidPassword = (text: string) => {
    // Regular expression to check if the password contains at least
       ↪ one number and one character
    const passwordRegex = /^(?=.*[0-9])(?=.*[a-zA-Z]).+$/;
    return passwordRegex.test(text);
}
```

Recommendation:

Implement a more robust password validation policy that includes requirements for a mix of uppercase letters, lowercase letters, numbers, and special characters. Additionally, consider enforcing a minimum length of 12 characters or more and implementing checks for password strength to ensure that passwords are sufficiently complex. This will enhance security and better protect sensitive wallet information.

Updates

The Kommunitas team has fixed the issue by implementing a password strength scoring system. The system now evaluates passwords based on length (minimum 8 characters), as well as the presence of uppercase and lowercase letters, numbers, and special characters. This ensures improved password security and aligns with best practices.

SHB.5 Importing Accounts from Mnemonic is Not Implemented

- Severity : MEDIUM
- Status : Fixed

- Likelihood : 3
- Impact : 1

Description:

The handleImport function is designed to handle the import of accounts using either a private key or a mnemonic phrase. However, the functionality for importing accounts from a mnemonic phrase is currently not operational because the code for invoking _importFromMnemonic is commented out. Although the _importFromMnemonic function is implemented, the handleImport function only processes private keys and disregards mnemonic phrases, leading to incomplete import functionality. As a result, users who attempt to import using a mnemonic phrase are not accommodated, leading to incomplete or non-functional import features.

Files Affected:

SHB.5.1: index.tsx

```
    const handleImport = async () => {
        setValid(true);
        if (!name) {
            showNotification("You must input account name", "warning");
        } else if (!priveKey) {
            showNotification("You must input private key or mnemonic to
                ↪ import", "warning");
        } else if (!password) {
            showNotification("You must input your password", "warning");
        } else if (wallets.map((_wallet: WALLET) => _wallet.name).
            ↪ includes(name)) {
```

```
            showNotification(`Name '${name}' already exists in your accounts`, "warning");
        } else if (priveKey.includes(" ")) {
            // if (utils.isValidMnemonic(priveKey)) {
            // _importFromMnemonic(priveKey);
            // } else {
            // showNotification(`Invalid mnemonic phrase was provided. Please try again with a different one`, "warning");
            // }
            showNotification(`Invalid private key was provided. Please try again with a different one`, "warning");
        } else if (!priveKey.includes(" ")) {
            _importFromPrivateKey(priveKey);
        }
    }
```

Recommendation:

Consider properly integrating the _importFromMnemonic function into the handleImport function to enable account imports using mnemonic phrases.
Alternatively, if the focus is solely on private key imports, update the frontend to clearly indicate that only private keys should be entered, ensuring users are aware of the input requirements and preventing confusion.

Updates

The Kommunitas team has fixed this issue by removing the conditional logic related to importing accounts using a mnemonic. This change resolves the problem by eliminating the unsupported functionality.

SHB.6 Lack of Two-Step Confirmation for Mnemonic

- Severity : MEDIUM
- Status : Fixed

- Likelihood : 1
- Impact : 3

Description:

The wallet creation process lacks a two-step confirmation mechanism to ensure users have securely saved their mnemonic phrases. Users can generate a random mnemonic and create their wallet account without confirming that the mnemonic has been recorded. Unlike MetaMask for example, which requires users to re-enter or confirm their seed phrase to verify that it has been saved, this implementation does not provide such a safeguard.

Recommendation:

Introduce a two-step confirmation process similar to MetaMask's approach, where users are required to re-enter or validate the mnemonic to confirm they have saved it.

Updates

The Kommunitas team has implemented a two-step confirmation process for the mnemonic phrase during wallet creation, enhancing security.

4 Best Practices

BP.1 Implement Proper Error Handling

Description:

Several functions in the walletsProvider file are missing adequate error handling. Functions such as setCurrentChains, setCurrentTokens, setCurrentWallets, and setCurrentWallet perform asynchronous operations with potential for failure, but they do not handle errors effectively. Currently, the catch blocks are empty, which means errors are not logged or managed, leading to potential issues being unnoticed and unresolved. Consider implementing a comprehensive error handling in all functions that perform asynchronous operations, ensuring that errors are logged or reported appropriately to facilitate debugging and maintenance.

Files Affected:

BP.1.1: walletsProvider.tsx

```
const setCurrentChains = async (_chains: Record<number | string, CHAIN
  ↪ >) => {
  const _db = new Level<string, any>('xkom', { valueEncoding: 'json'
    ↪ });
  try {
    await _db.put('chains', _chains);
    setChains(_chains);
  } catch (err) {
  } finally {
```

BP.1.2: walletsProvider.tsx

```
const setCurrentTokens = async (_tokens: Record<number | string, TOKEN
  ↪ []>) => {
  const _db = new Level<string, any>('xkom', { valueEncoding: 'json'
    ↪ });
  try {
      await _db.put('tokens', _tokens);
      setTokens(_tokens);
    } catch (err) {
    } finally {
```

BP.1.3: walletsProvider.tsx

```
  const setCurrentWallets = async (_wallets: WALLET[], _mnemonic?: string) => {
    const _db = new Level<string, any>('xkom', { valueEncoding: 'json' });
    try {
      await _db.put('wallets', _wallets);
      setWallets(_wallets);
      if (_mnemonic) {
        await _db.put('mnemonic', _mnemonic);
      }
    } catch (err) {
    } finally {
```

BP.1.4: walletsProvider.tsx

```
  const setCurrentWallet = async (_wallet: WALLET) => {
    const _db = new Level<string, any>('xkom', { valueEncoding: 'json' });
    try {
      await _db.put('wallet', _wallet);
      setWallet(_wallet);
    } catch (err) {
    } finally {
```

Status - Fixed

BP.2 Remove Dead Code

Description:

The audited project files contain commented-out code that serves no purpose in the current implementation. This dead code adds unnecessary clutter to the codebase, which can reduce readability and increase the risk of confusion and maintenance issues. It is best practice to remove any commented-out code to streamline the codebase and maintain a clean and manageable code environment. Regularly review the codebase to identify and eliminate any obsolete or unnecessary code. Note: The provided files are examples of dead code, and there may be additional commented-out code throughout the codebase that should also be removed.

Files Affected:

BP.2.1: accountSelector.tsx

```
    const handleClose = () => {
        // if (closable) {
        // close();
        // } else {
        // return;
        // }
        close();
    }
```

Status - Fixed

5 npm audit Results

```
{
  "auditReportVersion": 2,
  "vulnerabilities": {
    "@ethersproject/providers": {
      "name": "@ethersproject/providers",
      "severity": "high",
      "isDirect": false,
      "via": [
        "ws"
      ],
      "effects": [
        "ethers"
      ],
      "range": "<=5.7.2",
      "nodes": [
        "node_modules/@ethersproject/providers"
      ],
      "fixAvailable": {
        "name": "ethers",
        "version": "6.13.2",
        "isSemVerMajor": true
      }
    },
    "ethers": {
      "name": "ethers",
      "severity": "high",
      "isDirect": true,
      "via": [
        "@ethersproject/providers"
      ],
      "effects": [],
      "range": "5.0.0-beta.119 - 5.7.2",
      "nodes": [
        "node_modules/ethers"
      ],
      "fixAvailable": {
        "name": "ethers",
        "version": "6.13.2",
        "isSemVerMajor": true
      }
    },
    "ws": {
      "name": "ws",
      "severity": "high",
      "isDirect": false,
      "via": [
        {
          "source": 1098393,
          "name": "ws",
          "dependency": "ws",
          "title": "ws affected by a DoS when handling a request with
              ↪ many HTTP headers",
          "url": "https://github.com/advisories/GHSA-3h5v-q93c-6h6q",
          "severity": "high",
          "cwe": [
            "CWE-476"
          ],
          "cvss": {
            "score": 7.5,
            "vectorString": "CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H
                ↪ "
          },
          "range": ">=7.0.0 <7.5.10"
        }
      ],
      "effects": [
        "@ethersproject/providers"
      ],
      "range": "7.0.0 - 7.5.9",
      "nodes": [
        "node_modules/ws"
      ],
      "fixAvailable": {
        "name": "ethers",
        "version": "6.13.2",
        "isSemVerMajor": true
      }
    }
  },
  "metadata": {
    "vulnerabilities": {
      "info": 0,
      "low": 0,
      "moderate": 0,
      "high": 3,
      "critical": 0,
      "total": 3
    },
    "dependencies": {
      "prod": 456,
      "dev": 226,
      "optional": 11,
      "peer": 0,
      "peerOptional": 0,
      "total": 692
    }
  }
}
```

6 Conclusion

In this audit, we reviewed the design and implementation of the Kommunitas Official Telegram Bot contract and identified several issues of varying severity. The Kommunitas team resolved 5 issues and implemented mitigation measures for the remaining ones. Shellboxes' auditors advised the Kommunitas team to remain vigilant and keep these mitigated issues in mind, as a precaution, to avoid any potential future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
app/src/components/account/accountSelector.tsx	c913fe12beecd6f4b8ff0427f1ffedd3
app/src/components/account/import/index.tsx	4564a28ff88f1bf6429f967daa02ef6b
app/src/components/account/add/index.tsx	6219bee08f03933f114917287906116c
app/src/components/wallet/detail.tsx	77c0d86ad994ffdaec625967ab4775d6
app/src/components/wallet/importPhrase.tsx	abc180a139450d50c3d1d371aed9dbf2
app/src/components/wallet/passwordEditor.tsx	54829d761fbafb1197b70def49615b79
app/src/components/wallet/new/index.tsx	13dfade5f1f59bc70cb2d39ac32e69cd
app/src/components/wallet/import/index.tsx	c82be734016c1c23e11dcabbad784711
app/src/components/wallet/export/index.tsx	d655a33ec61206ffc0739d72018fdea3
app/src/components/wallet/create/index.tsx	fbf5311a9ab3217099bbbf397441f95e
app/src/providers/walletsProvider.tsx	40d29ba4a295541362f7cec38ded5235

7.2 Re-Audit

Files	MD5 Hash
app/src/components/account/accountSelector.tsx	c23acafee76809cfd4825d7519a0f294
app/src/components/account/import/index.tsx	743a298e34a52fea9f0e7f9e0e034062

app/src/components/account/add/index.tsx	98fc849fdf17a12ba29049f0baf8d4b4
app/src/components/wallet/detail.tsx	77c0d86ad994ffdaec625967ab4775d6
app/src/components/wallet/importPhrase.tsx	abc180a139450d50c3d1d371aed9dbf2
app/src/components/wallet/passwordEditor.tsx	1a925ecd59035f469f3e48f72a71b401
app/src/components/wallet/passwordValidator.tsx	ab777d0a837b7b088832804e485ef9af
app/src/components/wallet/new/index.tsx	21ec71409b8d6433dd31e6fe318d8d98
app/src/components/wallet/import/index.tsx	cd70692f0ee1b2158322c1d9c85c9cf2
app/src/components/wallet/export/index.tsx	d655a33ec61206ffc0739d72018fdea3
app/src/components/wallet/create/index.tsx	080c008850932862762ce648b024b8d0
app/src/providers/walletsProvider.tsx	cd0759cbd68b8b42cb3ff8c642a1a42b

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology’s proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don’t offer any kind of investing advice and shouldn’t be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com