The Brave browser includes a self-custodial crypto wallet. Brave Wallet is integrated within the browser so does not require extensions, and guarantees the privacy of its users and the security of their funds. Our Security & Privacy strategy follows high standards. It comprises a thorough internal Security & Privacy review process - frequently advancing the industry with security standards[1], external penetration testing on the more sensitive components, and a thriving external bug bounty program https://hackerone.com/brave.

As part of our open security practices, the following pages will guide you through a pentest exercise executed on the Brave Wallet concluded on 11/11/2022. At Brave, we perform pen-testing activities with reputable partners, such as Doyensec.

From the pentest emerged nine (9) security issues, of which one denial-of-service was marked as High severity. Besides one informational severity issue, Brave Software addressed all of the security vulnerabilities in a timely manner. Brave advises addressing the residual informational severity issue by employing hardware crypto wallets associated with the browser, such as Ledger.

[1] EIP contributions, in the case of the Wallet

DOYENSEC

Security Auditing Report

Brave Wallet

Prepared for: Brave Software, Inc.
Prepared by: Mykhailo Baraniak, Adrian Denkiewicz
Date: 11/11/2022

Table of Contents

Revision History

Version	Date	Description	Author
1	07/22/2022	First release of the final report	Mykhailo Baraniak Adrian Denkiewicz
2	07/24/2022	Peer review	Luca Carettoni
3	11/11/2022	Retesting	Sercan Sayitoglu
4	11/11/2022	Peer review	Anthony Trummer

Contacts

Company	Name	Email
Brave Software, Inc.	Yan Zhu	yan@brave.com
Brave Software, Inc.	Andrea Brancaleoni	abrancaleoni@brave.com
Doyensec, LLC	John Villamil	john@doyensec.com
Doyensec, LLC	Luca Carettoni	luca@doyensec.com

Executive Summary

Overview

Brave Software engaged Doyensec to perform a security assessment of the Brave Wallet. The project commenced on 07/11/2022 and ended on 07/22/2022 requiring two security researchers. The project resulted in nine (9) findings of which one (1) was rated as High severity.

The project consisted of a manual application security assessment performed on the desktop and mobile platforms.

On 11/11/2022, Doyensec security engineers retested the browser against all outstanding vulnerabilities. This document summarizes the results after this investigation.

Testing was conducted remotely from Doyensec's EMEA and US offices.

Scope

Through meetings with Brave Software the scope of the project was clearly defined. The agreed upon assets are listed below:

- Brave Wallet for Android
- Brave Wallet for iOS
- Brave Wallet for Desktop

The testing took place using a development build of the Brave Browser using the latest version of the software at the time of testing. In detail, this activity was performed on the following releases:

- Brave-core
  - 52aa957fcb6999f85b44107084a0326832c693f2 on the branch *wallet-pentest*
- Brave-ios
  - d30d8a7c3793880177d187299dc14a9955e4929b on the branch *development*

Scoping Restrictions

During the engagement, Doyensec did not encounter any difficulties while testing the application. Brave Software was very responsive in debugging any issues to ensure a smooth assessment.

Findings Summary

Doyensec researchers discovered and reported nine (9) vulnerabilities in the Brave Wallet. While most of the issues were departures from best practices and Low-severity flaws, Doyensec identified two (2) issues rated respectively as Medium and High severity.

It is important to reiterate that this report represents a snapshot of the environment's security posture at a point in time.

The findings included several Denial of Service scenarios which can either result in termination of the renderer or browser processes. We also identified four Insecure Design issues which expose the Wallet to various risks and observed two Information Exposure issues.

Overall, the security posture of the tested components was found to be in line with industry best practices.

Doyensec found the system to be well architected with the exclusion of the following aspects:

- The UI presented to the users incorrectly handled Unicode characters and overly long messages
- Sensitive information, such as the user's private key, inconsistently remained in the heap and could be exposed to other Browser components

Recommendations

The following recommendations are proposed based on studying the Brave Wallet's security posture and the vulnerabilities discovered during this engagement.

Short-term improvements

- Work on mitigating the discovered vulnerabilities. You can use **Appendix B** - Remediation Checklist to make sure that you have covered all areas
- Decrease the maximum lock-out time for the wallet. The Desktop and Android wallets can remain unlocked for an entire week. The iOS wallet locks after 30 minutes

Long-term improvements

- Ensure the exposed provider objects are always trusted and can be safely used by DApps
- We observed that provider objects could be used as a `Content-Security-Policy` bypass mechanism. They are not restricted by any policies, and thus could be used to, for example, exfiltrate sensitive data to the disallowed endpoints. As many different CSP bypasses could be used instead we decided to not report this as a finding. Nevertheless, it is recommended to improve this behavior
- On mobile devices, no integrity checks to prevent execution on rooted or jail-broken devices are made. The wallet's sensitivity is a good reason to detect and block its usage on vulnerable devices

Retesting Overview

Apart from one informational severity issue, all of the security vulnerabilities were addressed in a timely manner by Brave Software. Brave advises to address the residual informational severity issue by employing hardware crypto wallets associated to the browser, such as Ledger.

Methodology

Overview

Doyensec treats each engagement as a fluid entity. We use a standard base of tools and techniques from which we built our own unique methodology. Our 30 years of information security experience has taught us that mixing offensive and defensive philosophies is the key to standing against threats. Thus we recommend a *whitebox* approach combining dynamic fault injection with an in-depth study of the source code to maximize the ROI on bug hunting.

During this assessment, we have employed standard testing methodologies (e.g., OWASP Testing guide recommendations), as well as custom checklists, to ensure full coverage of both code and vulnerability classes.

Setup Phase

Brave Software provided access to the source code repositories, debugging symbols, and dedicated binaries for all in-scope operating systems.

The brave-core repository, hosted at https://github.com/brave/brave-core, contains the source code used by the Android and Desktop versions of the wallet.

The source code of the Brave Wallet for iOS is hosted at https://github.com/brave/brave-ios.

The official Brave Wallet's developer documentation can be found on this page: https://wallet-docs.brave.com/.

Tooling

When performing assessments, we combine manual security testing with state-of-the-art tools in order to improve efficiency and efficacy of our effort.

During this engagement, we used the following tools:

- Burp Suite
- WinDbg Preview
- Android Studio
- Xcode
- Apktool
- Curl, netcat and other Linux utilities

Web Application and API Techniques

Web assessments are centered around the data sent between clients and servers. In this realm, the principle audit tool is Burp Suite. However, we also use a large set of custom scripts and extensions to perform specific audit tasks. We focus on authorization, authentication, integrity and trust. We study how data is interpreted, parsed, stored, and relayed between producers and consumers.

The exposed Web API primitives were tested from the perspective of a compromised web page. We inspected the data exchanged between the Web API and configured EVM chains.

Mobile Application Techniques

During mobile security assessments, we treat the entire device as an untrusted environment. We study an application's use of cryptography to secure data, in transit and at rest, to protect user's privacy. For the in-scope servers, we attack the remote mobile endpoints using our web testing techniques and methodology.

Having a great understanding of the architecture and security structure of Android and iOS devices, we evaluate platform specific functionality such as the safe use of intents and broadcast messages, IPC controls, secure sandbox configuration, user protection and confidentiality, and UX interaction.

We audit the design and implementation of cryptography, custom protocols, anti-cheating systems, and jailbreak detection features. In this area, we use physical devices (rooted or jailbroken), emulators and debugging tools to carefully exercise all application functionalities.

Desktop and Server Applications

Doyensec has extensive experience finding flaws in thick clients, standalone binaries, and server daemons. We write customized tools to map out control flow and study an application's behavior and internals. Mapping out attack surface, whether local or remote, is paramount to a successful engagement. Doyensec also studies the application's ecosystem, looking for potential pitfalls and common misconceptions.

We deconstruct the application looking for privacy leaks and secrets. We understand the storage, transmission, and protection of user information is critical, along with the server-side handling of user provided data.

The web browser has a unique threat model where the process browser is considered trusted, and multiple renderer processes are always assumed to be compromised. Given that the wallet is implemented in the trusted process, we analyzed the attack scope from the perspective of a malicious web page and a fully compromised renderer.

Project Findings

The table below lists the findings with their associated ID and severity. The severity ranking and vulnerability classes are defined in **Appendix A** at the end of this document. The vulnerability class column groups the entry into a common category, while the status column refers to whether the finding has been fixed at the time of writing.

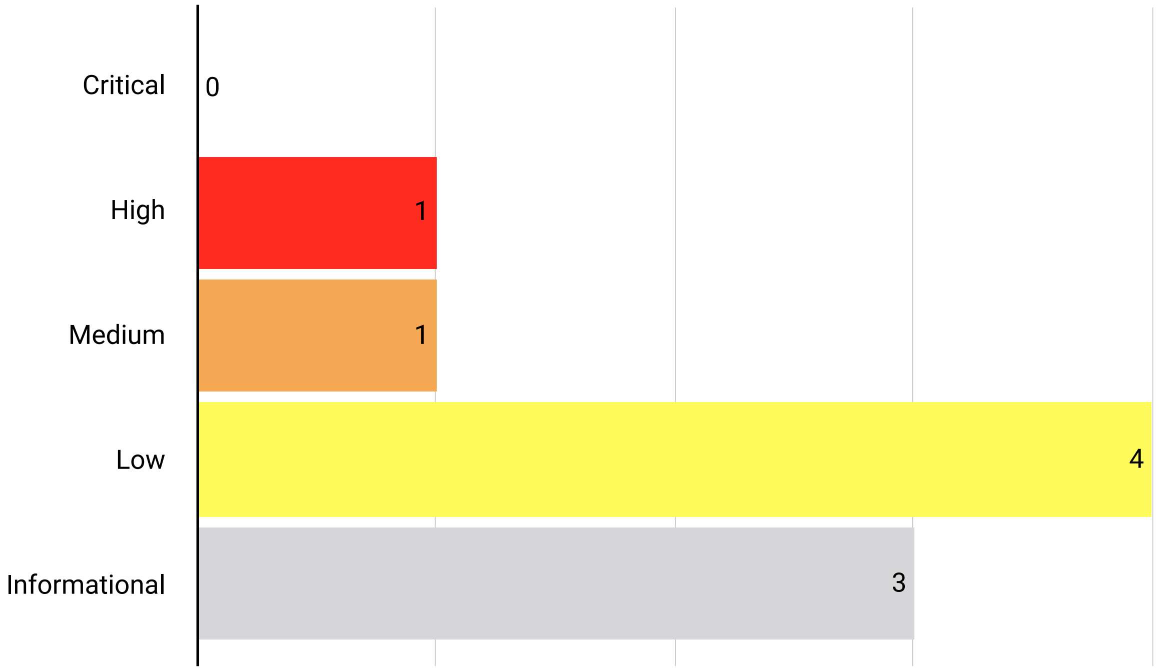
This table is organized by time of discovery. The issues at the top were found first, while those at the bottom were found last. Presenting the table in this fashion has a number of benefits. It inherently shows the path our auditing took through the target and may also reveal how easy or difficult it was to discover certain findings. As a security engagement progresses, the researchers will gain a deeper understanding of a target which is also shown in this table.

Findings Recap Table

ID	Title	Vulnerability Class	Severity	Status
BRA-Q322-1	Malformed Ethereum Method Crashes The Browser	Denial of Service (DoS)	High	Closed
BRA-Q322-2	Sensitive Functionality Is Not Password Protected	Insecure Design	Low	Closed
BRA-Q322-3	Misleading Blockchain Names	Insecure Design	Informational	Closed
BRA-Q322-4	Inconsistent Use of SecureZeroData to Protect Private Keys	Insecure Design	Low	Closed
BRA-Q322-5	Insufficient Number of PBKDF Iterations	Insufficient Cryptography	Informational	Closed
BRA-Q322-6	Prototype Pollution Against Window Provider Objects	Insecure Design	Low	Closed
BRA-Q322-7	Misleading Signing Request Message	Insecure Design	Medium	Closed
BRA-Q322-8	The Wallet Details Are Exposed On brave://prefs-internals	Information Exposure	Informational	Open
BRA-Q322-9	Missing Blurring for Recovery Phrase Screen on iOS	Information Exposure	Low	Closed

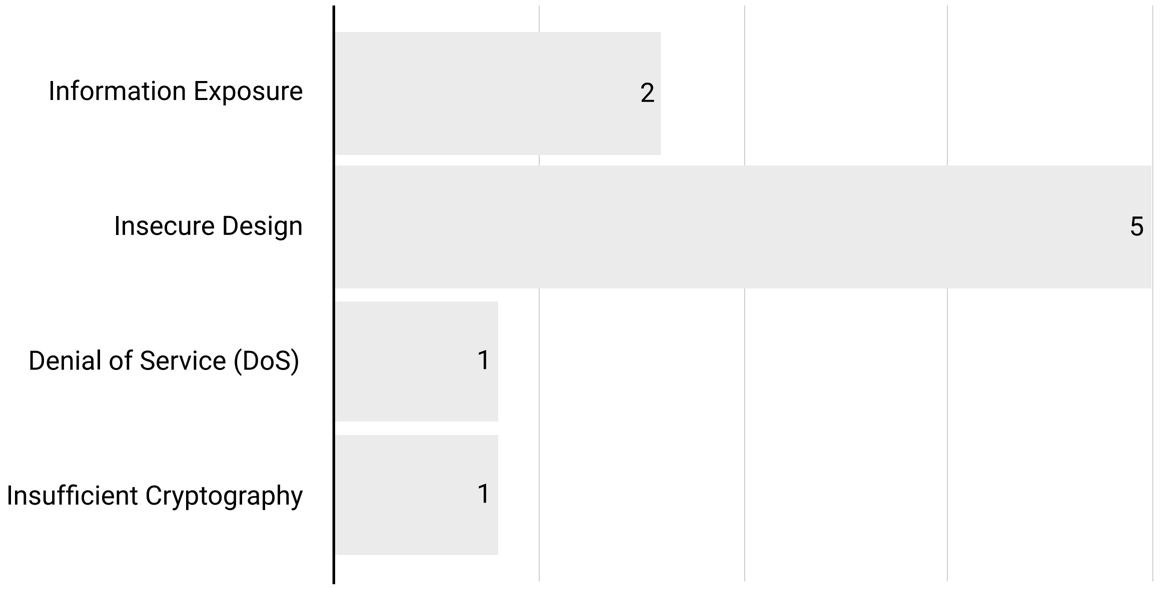
Findings per Severity

The table below provides a summary of the findings per severity.



Findings per Type

The table below provides a summary of the findings per vulnerability class.



BRA-Q322-1. Malformed Ethereum Method Crashes The Browser

Severity	**High**
Vulnerability Class	Denial of Service (DoS)
Component	components/brave_wallet/browser/json_rpc_service.cc
Status	Closed

Description

The Brave Wallet exposes the `window.ethereum.request` method which is used to submit an RPC request to the remote EVM node. For some methods, the provider itself handles the response, but any other methods (even unrecognized ones) are sent directly to a remote EVM node. This is implemented by the `EthereumProviderImpl::CommonRequestOrSendAsync` method in the `components/brave_wallet/browser/ethereum_provider_impl.cc` file.

```
else {
    json_rpc_service_->Request(normalized_json_request, true, std::move(id),
                               mojom::CoinType::ETH, std::move(callback));
}
```

In this case, the method name is also copied to the `X-Eth-Method` HTTP header inside the new JSON RPC request. This is happening inside the `JsonRpcService::RequestInternal` function, in the `components/brave_wallet/browser/json_rpc_service.cc` file:

```
request_headers["X-Eth-Method"] = method;
```

Note that user-controlled value is assigned without any sanitization. This assignment is the root-cause of the discussed vulnerability.

When a request object is prepared, it is passed to the `APIRequestHelper::Request` method as defined in the `components/api_request_helper/api_request_helper.cc` file. The method calls the `APIRequestHelper::CreateLoader` method in the same file.

The headers are copied to the final object, using the `SetHeader` method:

```
if (!headers.empty()) {
    for (auto entry : headers)
      request->headers.SetHeader(entry.first, entry.second);
}
```

This method is not a part of the `brave-core` repository, but its definition can be found in the public Chromium code:

```
void HttpRequestHeaders::SetHeader(const base::StringPiece& key,
                                   const base::StringPiece& value) {
  // Invalid header names or values could mean clients can attach
  // browser-internal headers.
```

```
    CHECK(HttpUtil::IsValidHeaderName(key)) << key;
    CHECK(HttpUtil::IsValidHeaderValue(value)) << key << ":" << value;
    SetHeaderInternal(key, value);
  }
```

The second CHECK macro is testing the user-supplied value for the presence of any invalid characters such as new lines or a null byte:

```
  bool HttpUtil::IsValidHeaderValue(base::StringPiece value) {
    // Just a sanity check: disallow NUL, CR and LF.
    for (char c : value) {
      if (c == '\0' || c == '\r' || c == '\n')
        return false;
    }
    return true;
  }
```

If the provided method name contained a new line character, the method would return false, thus the CHECK macro would trigger process termination as stated in the Chromium code:

```
// CHECK dies with a fatal error if its condition is not true. It is not
// controlled by NDEBUG, so the check will be executed regardless of compilation
// mode.
```

Note that this happens inside the browser process, thus the entire Brave browser is terminated, not just a single renderer process.

Reproduction Steps

To trigger the crash, serve the following code on an HTTPS-enabled website:

```
  <script>
          function poc() { window.ethereum.request({method: 'foo\n', params:
  []}); }
  </script>

  <button onclick="poc()">CRASH</button>
```

Navigate to the page and click on the “CRASH” button. The browser will immediately exit. The issue can be reproduced on the desktop and mobile platforms.

Impact

High. Visiting a page with a malicious script can crash the browser process. The attack does not require user-action or a configured Brave Wallet.

Complexity

Low. The exploitation requires an attacker to use well-defined API with unexpected input values.

Remediation

Sanitize the method name by ensuring no null byte or new line characters are present inside. The Ethereum EIP-1193 standard suggests to return error 4200 ("Unsupported Method") in such a situation. Alternatively, allow such characters in the request body and encode them in the HTTP header.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

This remediation progress was verified on:

- Brave Android 1.47.56
- Brave Desktop 1.47.55
- Brave IOS 1.45.116

The affected source code has been updated and a sanitization method has been implemented. The correct fix has also been confirmed by dynamic testing.

Resources

- Ethereum.org, EIP-1193: Ethereal Provider JavaScript API
  https://eips.ethereum.org/EIPS/eip-1193

- The Chromium Authors, http_request_headers.cc : SetHeader method
  https://chromium.googlesource.com/chromium/src/+/refs/heads/main/net/http/http_request_headers.cc

- The Chromium Authors, check.h : CHECK macro
  https://chromium.googlesource.com/chromium/src/+/main/base/check.h

BRA-Q322-2. Sensitive Functionality Is Not Password Protected

Severity	Low
Vulnerability Class	Insecure Design
Component	Brave Wallet | Desktop/Android/IOS
Status	Closed

Description

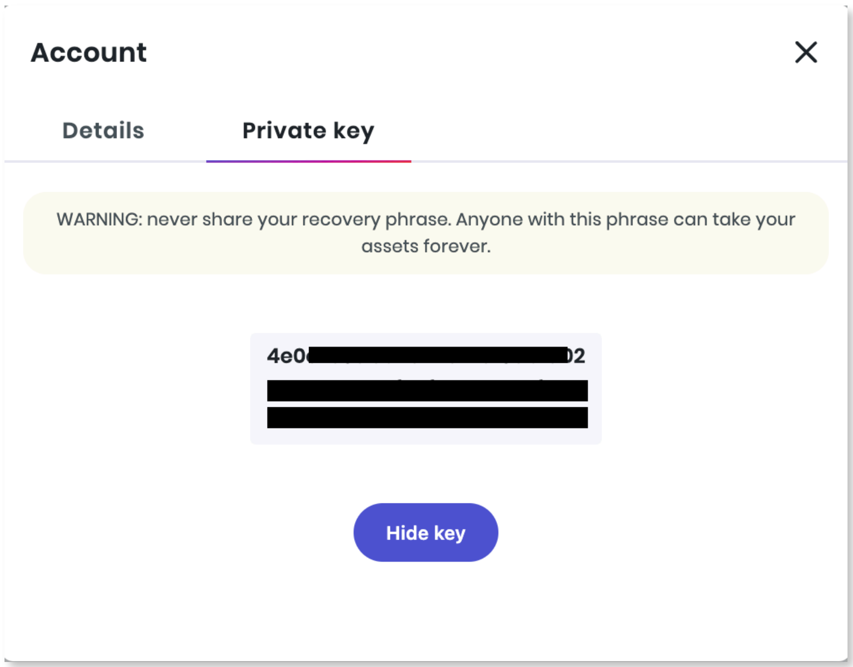
To increase the overall security of crypto wallet applications, it is important to require user credentials before performing sensitive operations such as private key retrieval, seed backup, and account removal.

The Brave Wallet doesn't protect sensitive functionalities with an additional password confirmation dialog. While this is likely a product decision, we believe that such additional step is largely accepted by users, hence implemented by other major wallets.

Reproduction Steps

To reproduce the issue, perform the following steps:

1. A victim user unlocks the Brave wallet and leaves the wallet unmonitored
2. An attacker gets access to the victim's computer and the export private key by clicking on: Accounts -> Select Account1 -> Account details -> Private Key -> Show key



WARNING: never share your recovery phrase. Anyone with this phrase can take your assets forever.

4e0[redacted]2

Impact

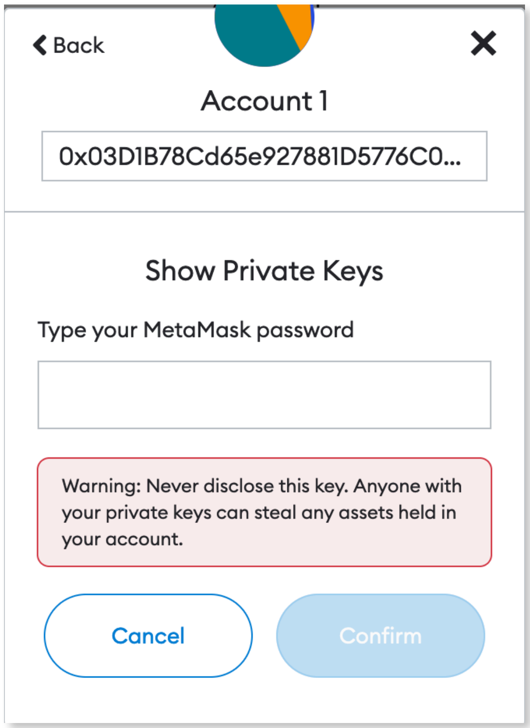
High. An attacker with physical access to the unlocked wallet will be able to completely compromise the account.

Complexity

High. The attacker needs to get physical access to the unlocked wallet. The desktop version of the wallet enables arbitrary lock time settings, which might mitigate the issue if the user sets a small timeframe.

Remediation

Protect, with a password verification window, all sensitive functionalities such as private key retrieval, seed backup, account removal, etc. Implement a similar protection mechanism as in other major crypto wallets (MetaMask is shown below):



Warning: Never disclose this key. Anyone with your private keys can steal any assets held in your account.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

This remediation progress was verified on:

- Brave Android 1.47.56
- Brave Desktop 1.47.55

- Brave IOS 1.45.116

The affected source code has been updated and a password challenge method has been implemented. The fix was also confirmed by dynamic testing.

Resources

- MITRE, CWE-284: Improper Access Control
  https://cwe.mitre.org/data/definitions/284.html

BRA-Q322-3. Misleading Blockchain Names	
Severity	Informational
Vulnerability Class	Insecure Design
Component	Brave Wallet | Desktop
Status	Closed

Description

Before rendering a blockchain name, Brave Wallet uses the `reduceNetworkDisplayName` function to reduce the network name.

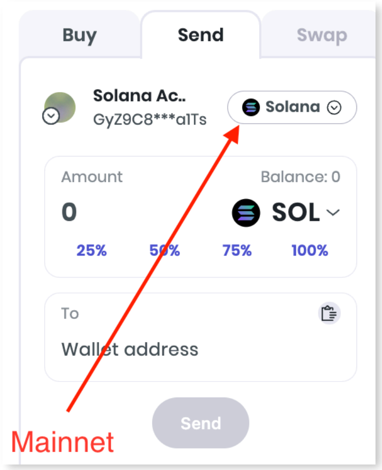
```
export const reduceNetworkDisplayName = (name: string) => {
  if (!name) {
    return ''
  } else {
    const firstWord = name.split(' ')[0]
    if (firstWord.length > 9) {
      const firstEight = firstWord.slice(0, 6)
      const reduced = firstEight.concat('..')
      return reduced
    } else {
      return firstWord
    }
  }
}
```

Such a design decision leads to ambiguity for users and possible unintended transactions. Users must hover over the blockchain name to be sure on which network they will be executing transactions.

Reproduction Steps

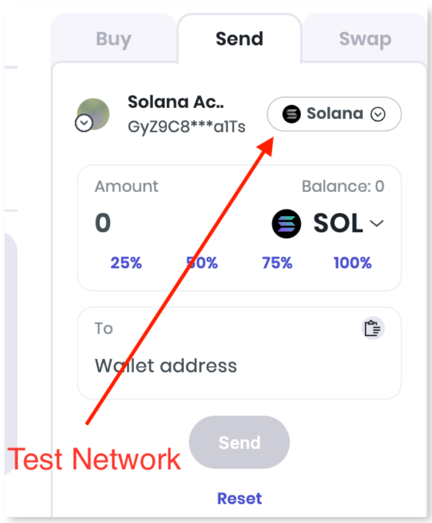
To reproduce the issue, perform the following steps:

1. Unlock the desktop version of the Brave Wallet and select `Solana Mainnet`



2. Verify how the chain name is rendered in the wallet
3. Switch the network to the `Solana Testnet` and verify how it is rendered in the wallet.

   Without a mouse hover over the blockchain name, a user can spot only a very minor difference in the icon color.



Impact

Potentially high. Such insecure design could lead to unintended transactions. For example, executing transaction on the main network instead of using test network.

Complexity

N/A

Remediation

Always show complete blockchain name. Or at least the first and last word from the whole blockchain name. A secure user experience can greatly improve the overall security of the wallet against abuses and accidental mistakes.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

This remediation progress was verified on Brave Desktop 1.47.55.

The affected source code has been updated and the blockchain names are now securely displayed. The fix was also confirmed by dynamic testing.

BRA-Q322-4. Inconsistent Use of SecureZeroData to Protect Private Keys

Severity	**Low**
Vulnerability Class	Insecure Design
Component	components/brave_wallet/browser/internal/hd_key.cc
Status	Closed

Description

The accounts' private keys are stored as private `std::vector<uint8_t>` members of the `HDKey` class as defined in the `components/brave_wallet/browser/internal/hd_key.cc` file. The `HDKey` object is destroyed as soon as a user locks the wallet. The destructor ensures that the private key data is safely zeroed which is not the default behavior of the `std::vector` destructor.

```
HDKey::~HDKey() {
  secp256k1_context_destroy(secp256k1_ctx_);
  SecureZeroData(private_key_.data(), private_key_.size());
}
```

The `SecureZeroData` method is defined in the `components/brave_wallet/common/mem_utils.cc` file. It calls `SecureZeroMemory` on Windows and manually overwrites the memory on other systems.

```
void SecureZeroData(void* data, size_t size) {
  if (data == nullptr || size == 0)
    return;
#if BUILDFLAG(IS_WIN)
  SecureZeroMemory(data, size);
#else
  // 'volatile' is required. Otherwise optimizers can remove this function
  // if cleaning local variables, which are not used after that.
  volatile uint8_t* d = (volatile uint8_t*)data;
  for (size_t i = 0; i < size; i++)
    d[i] = 0;
#endif
}
```

We observed an inconsistency in handling the private key whenever its copy is made. Multiple copies are made using a bare `std::vector` which does not guarantee a secure release of the internal data. A `std::vector` always stores data on the heap. When a `std::vector` is released, the data may be reused by other code. The following methods are using copies of the private key without ensuring that internal memory is safely released.

components/brave_wallet/browser/internal/hd_key.cc file, line 371.

```
std::string HDKey::GetPrivateExtendedKey() const {
  std::vector<uint8_t> key;
  key.push_back(0x00);
  key.insert(key.end(), private_key_.begin(), private_key_.end());
  return Serialize(MAINNET_PRIVATE, key);
}

std::string HDKey::Serialize(uint32_t version,
                             const std::vector<uint8_t>& key) const {
  std::vector<uint8_t> buf;
  ...
  buf.insert(buf.end(), key.begin(), key.end());
}
```

components/brave_wallet/browser/internal/hd_key.cc file, line 378.

```
std::string HDKey::GetEncodedPrivateKey() const {
 return base::ToLowerASCII(base::HexEncode(private_key_));
}
```

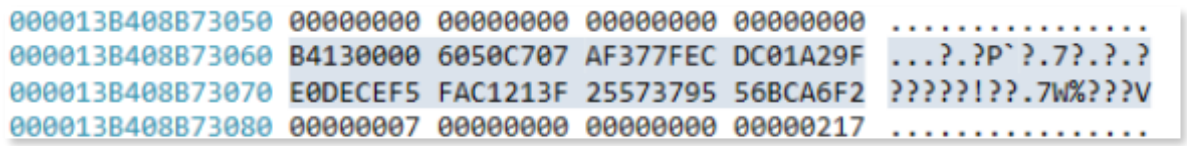
components/brave_wallet/browser/internal/hd_key.cc file, line 472.

```
std::unique_ptr<HDKeyBase> HDKey::DeriveChild(uint32_t index) {
  ...
  std::vector<uint8_t> data;
  ...

  data.insert(data.end(), private_key_.begin(), private_key_.end()); //; BAD
  ...
  if (!private_key_.empty()) {
    // Private parent key -> private child key
    // Also Private parent key -> public child key because we always create
    // public key.
    std::vector<uint8_t> private_key = private_key_;
    ...
  }
}
```

Reproduction Steps

To observe one of the private key copies left in the memory, attach a debugger to the browser process. Make sure to configure the symbols and load the source code files. Set breakpoints on the aforementioned methods and start interacting with the unlocked wallet.



```
000013B408B73050 00000000 00000000 00000000 00000000  ................
000013B408B73060 B4130000 6050C707 AF377FEC DC01A29F  ...?.?P`?.7?.?.?
000013B408B73070 E0DECEF5 FAC1213F 25573795 56BCA6F2  ?????!??.7W%???V
000013B408B73080 00000007 00000000 00000000 00000217  ................
```

When any breakpoint is hit, note down the heap address of the private key copy. Then, step out of debugged function to ensure all objects are destroyed.
Verify that the private key copy remains in the heap memory. Sporadically, the content may be partially or fully overwritten by other threads.

Impact

The private key data may be leaked by other browser features which do not assume any sensitive data can be present in newly allocated memory. The destructor `HDKey` object confirms that the private key is considered very sensitive information.

Complexity

High. The risk of a private key disclosure is lowered due to the strong separation of the browser process and renderer instances. To directly scan the memory, the attacker would need to escape from the browser sandbox.

Remediation

Implement a secure version of a vector that always clears the internal memory or ensure no copies of the private key are ever made. The move semantics or shared pointers could be used to guarantee better control over the private key lifecycle.

Due to the constant and limited size of the private key, `std::array` could be used as a better alternative for the copies. This class always stores its data on the stack, hence the lifetime of the copy is significantly reduced. Note that the implementation does not zero the memory either, hence it does not provide full secrecy.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

The affected source code has been updated with new deleter and allocator methods.

Resources

- MITRE, CWE-226: Sensitive Information in Resource Not Removed Before Reuse
  https://cwe.mitre.org/data/definitions/226.html

BRA-Q322-5. Insufficient Number of PBKDF Iterations	
Severity	**Informational**
Vulnerability Class	Insufficient Cryptography
Component	components/brave_wallet/browser/keyring_service.cc
Status	Closed

Description

The keyring implemented by the Brave Wallet uses `PBKDF2` as its key derivation function. When the standard was written in the year 2000 the recommended minimum number of iterations was 1,000, but the parameter is intended to be increased over time as CPU speeds increase. Since 2001, OWASP recommends using 310,000 iterations for the `PBKDF2-HMAC-SHA256` configuration.

The `CreateEncryptorForKeyring` method, as implemented in the `components/brave_wallet/browser/keyring_service.cc` file, uses a hardcoded number of iterations:

```
const size_t kSaltSize = 32;
const size_t kNonceSize = 12;
const int kPbkdf2Iterations = 100000;
const int kPbkdf2KeySize = 256;

bool KeyringService::CreateEncryptorForKeyring(const std::string& password,
                                               const std::string& id) {
  if (password.empty())
    return false;
  std::vector<uint8_t> salt(kSaltSize);
  if (!GetPrefInBytesForKeyring(kPasswordEncryptorSalt, &salt, id)) {
    crypto::RandBytes(salt);
    SetPrefInBytesForKeyring(kPasswordEncryptorSalt, salt, id);
  }
  encryptors_[id] = PasswordEncryptor::DeriveKeyFromPasswordUsingPbkdf2(
      password, salt, kPbkdf2Iterations, kPbkdf2KeySize);
  return encryptors_[id] != nullptr;
}
```

Reproduction Steps

This is a source code finding.

Impact

The current `PBKDF2` configuration is not considered weak, but it does not follow the applicable recommendation. A weaker encryption scheme can be subjected to brute force attacks that have a reasonable chance of succeeding using current attack methods and resources.

Complexity

High. The attacker must first gain access to the encrypted secret and then perform a password cracking attack. The complexity will vary depending on the password strength.

Remediation

Increase the number of iterations to the recommended value. As this recommendation will change over time, it is important to monitor and keep the `PBKDF2` parameters up to date.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

The affected source code has been updated and the number of `PBKDF2` iterations were increased to 310K.

Resources

- OWASP, "Password Storage Cheat Sheet"
  https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html#pbkdf2

BRA-Q322-6. Prototype Pollution Against Window Provider Objects

Severity	**Low**
Vulnerability Class	Insecure Design
Component	• components/brave_wallet/resources/solana_provider_internal.js • components/brave_wallet/renderer/js_solana_provider.cc • components/brave_wallet/resources/ethereum_provider.js
Status	Closed

Description

The Brave Wallet exposes several provider objects, such as

- `window.ethereum`
- `window.solana`
- `window.braveSolana`
- `window._brave_solana`
- `window.web3`

The configuration on `brave://settings/wallet` controls whether other extensions, such as `MetaMask`, can overwrite the provider objects. When the "*Brave Wallet*" option is selected, no other code should be able to change them.

The following finding describes several places in which prototype pollution can exploit this configuration.

The window._brave_solana object

`window._brave_solana` is initially exposed as an empty dictionary (`{}`). When a Solana account is connected to the page, it contains two additional fields: `createPublickey` and `createTransaction`. The initialization of the object is implemented in the `components/brave_wallet/resources/solana_provider_internal.js` file:

```
(function () {
  if (!window._brave_solana || !!window._brave_solana.createPublickey ||
        !!window._brave_solana.createTransaction) {
    return
 }

  Object.defineProperties(window._brave_solana, {
    createPublickey: {
      value: (base58Str) => {
        const PublicKey = require('@solana/web3.js').PublicKey
        const result = new Object()
        result.publicKey = new PublicKey(base58Str)
        return result
      },
      writable: false
    },
    createTransaction: {
      value: (serializedTx) => {
        const Transaction = require('@solana/web3.js').Transaction
        return Transaction.from(new Uint8Array(serializedTx))
      },
      writable: false
    }
  })
})()
```

At the beginning, the function tests for the presence of `window._brave_solana` and its fields. This condition can be satisfied using prototype pollution for the `Object` class. An attacker can create arbitrary versions of the affected fields.

The attacker-supplied `createPublickey` method is called by the `JSSolanaProvider::CreatePublicKey` method in the `components/brave_wallet/renderer/js_solana_provider.cc` file:

```
v8::Local<v8::Value> JSSolanaProvider::CreatePublicKey(
    v8::Local<v8::Context> context,
    const std::string& base58_str) {
  // Internal object for CreatePublicKey and CreateTransaction
  ExecuteScript(render_frame()->GetWebFrame(), *g_provider_internal_script);
  const base::Value public_key_value(base58_str);
  std::vector<v8::Local<v8::Value>> args;
  args.push_back(v8_value_converter_->ToV8Value(&public_key_value, context));
  v8::MaybeLocal<v8::Value> public_key_result =
      CallMethodOfObject(render_frame()->GetWebFrame(), u"_brave_solana",
                         u"createPublickey", std::move(args));

  return public_key_result.ToLocalChecked();
}
```

The method is executed in the context of the renderer process, thus we don't see any risk for a direct elevation of privileges. However, the attacker controls the returned value, hence a fake public key can be reported to the requesting code. Depending on the DApp, this could result in an invalid transfer or similar types of issues.

The aforementioned `JSSolanaProvider::CreatePublicKey` is also called in numerous places as an argument in the `CHECK()` macro. As described in `BRA-Q322-1`, the macro will crash the process if the condition is not met. If the attacker intentionally provides an illegal object here (e.g., not a function), it will crash the renderer process. A malicious script on the page could exploit this behavior to perform a DoS attack for a targeted user.

Similar attacks could be performed using the hijacked `window._brave_solana.createTransaction` field.

Pollution of toString method

The `JSSolanaProvider::GetSignatures` method defined in the `components/brave_wallet/renderer/js_solana_provider.cc` file contains the following code:

```
v8::MaybeLocal<v8::Value> v8_pubkey =
        CallMethodOfObject(render_frame()->GetWebFrame(), v8_pubkey_object,
                           u"toString", std::vector<v8::Local<v8::Value>>());
```

The `toString` method of the returned public key can also be polluted or redefined in the called object.

Generic prototype pollution

The `window.ethereum` and `window.solana` objects are exposed using standard JavaScript `Proxy` objects. They are populated using the `Object.defineProperty` method. Any missing property will be implicitly inherited from the `Object` prototype, thus opening the providers to prototype pollution attacks.

An attacker who defined the `Object.prototype.get` method will gain control over every requested property once the page is loaded.

Reproduction Steps

Set the "*Brave Wallet*" option on the `brave://settings/` wallet page.

To trigger the crash, serve the following code on a HTTPS-enabled website and visit the page with the unlocked Brave Wallet and a configured Solana account:

```
<script>
Object.prototype.createPublickey = (x) => { };
window.solana.connect(); // should trigger a crash
</script>
```

To return the arbitrary public key value use the following code:

```
<script>
Object.prototype.createPublickey = (x) => { return {publicKey: 'ABCD'}; };
window.solana.connect();
alert(window.solana.publicKey); // should show 'ABCD'
</script>
```

To reproduce a generic prototype pollution use the following code:

```
<script>
Object.prototype.get = (x,y) => {
 if (y == "selectedAddress") return "Fake-address";
 return x[y];
}
alert(window.ethereum.selectedAddress); // should show 'Fake-address'
</script>
```

Impact

Low. The attacker can either manipulate the data returned by the provider objects or trigger a renderer crash. The described attacks are valid even with the hardened settings ("*Brave Wallet*" configuration). The impact will vary depending on the implementation of the affected DApp.

The manipulated data does not affect the functionality of the wallet itself or the information displayed in the wallet window. Every privileged operation still requires confirmation by the user and the spoofed data is not used by the internal wallet methods.

Complexity

Low. Basic web application skills are required in order to exploit this vulnerability. Triggering a crash requires familiarity with the wallet's source code.

Remediation

Ensure the exposed provider objects have frozen prototypes. The objects created in the following fashion will not inherit the default prototype:

```
let obj = Object.create(null);
obj.__proto__ // undefined
obj.constructor // undefined
```

It is also possible to freeze the specific object by calling `Object.freeze` method on it.

It is recommended to replace the `CHECK()` macro with a verification code that handles the exception and does not crash the renderer process.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

This remediation progress was verified on:

- Brave Android v1.47.56
- Brave Desktop 1.47.55
- Brave IOS v1.45.116

The affected source code has been updated; Solana JavaScript api and builtins Wallet script execution safety mechanisms have been implemented to protect against prototype pollution attacks. Deprecated and/or 3rd party scripts were considered out of scope for this mitigation. The results have been also confirmed by dynamic testing.

Resources

- Mozilla, MDN reference: Object.create
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create#object_with_null_prototype

- Mozilla, MDN reference: Object.freeze
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/freeze

BRA-Q322-7. Misleading Signing Request Message

Severity	**Medium**
Vulnerability Class	Insecure Design
Component	Sign Panel
Status	Closed

Description

Brave Wallet is a non-custodial wallet, and only its users have sole control of their private keys. With the help of Brave Wallet, users can conveniently perform multiple cryptographic operations, such as sign ing transactions or even signing arbitrary messages. Such signed messages could be used for a variety of use cases including, but not limited to:

- authenticating users
- signing off-chain messages for on-chain protocols, etc.

Great care should be taken to show a user exactly what the website requires to be signed, and warn in the case of any suspicious scenarios.

In the sign-request window, the Brave Wallet renders all input Unicode characters. This makes the following phishing scenarios possible:

- using new line characters to hide the actual payload in the non visible area of the sign request dialog (no scrollbar is shown to the user until the forcible scrolls in the right area of the window)
- using Right-To-Left character to change direction of the rendered text

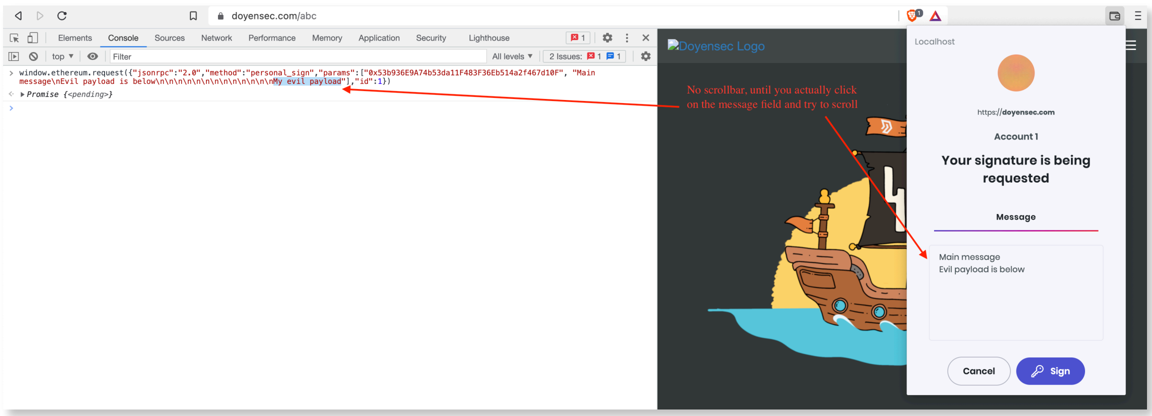
Reproduction Steps

To reproduce the issue, perform the following steps:

1. Unlock the MacOS version of the Brave Wallet and navigate to any connected website
2. Open the developer console and execute the following JavaScript code:

```
window.ethereum.request({“method":"personal_sign","params":["<ADDRESS>",  "Main
Message\nEvil payload is below \n\n\n\n\n\n\n\n\n\n\n\nMy Evil payload"],"id":1})
```

3. Note that `<ADDRESS>` should be changed to your account address
4. Verify the dialog presented to the user



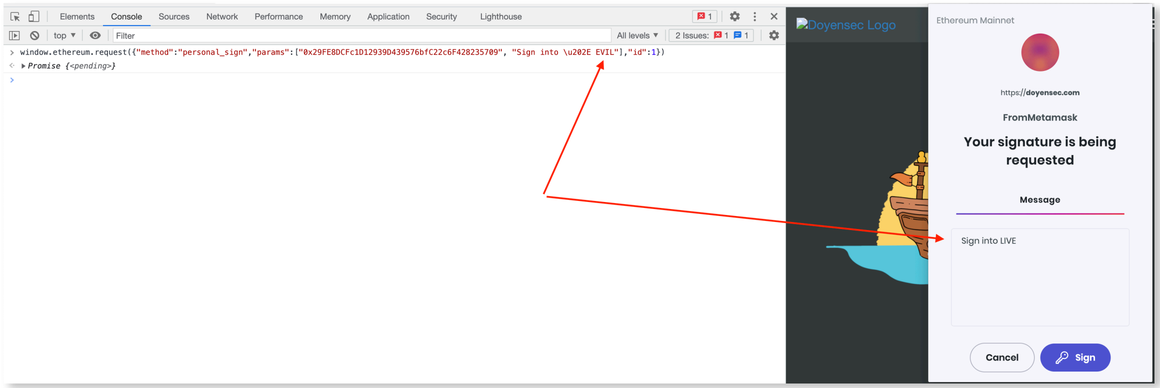
```
> window.ethereum.request({"jsonrpc":"2.0","method":"personal_sign","params":["0x53b936E9A74b53da11F483F36Eb514a2f467d10F", "Main
  message\nEvil payload is below\n\n\n\n\n\n\n\n\n\n\n\nMy evil payload"],"id":1})
<· ▸ Promise {<pending>}
```

No scrollbar, until you actually click on the message field and try to scroll

5. Click Cancel and execute the following JavaScript code:

```
window.ethereum.request({"method":"personal_sign","params":["<ADDRESS>", "Sign into \u202E EVIL"],"id":1})
```

6. Verify a dialog presented to the user. It contains text "`Sign into LIVE`"



```
> window.ethereum.request({"method":"personal_sign","params":["0x29FE8DCFc1D12939D439576bfC22c6F428235709", "Sign into \u202E EVIL"],"id":1})
<· ▸ Promise {<pending>}
```

Impact

High. An attacker can cause the UI to display erroneous data, enabling the attacker to trick the user into performing the wrong action (such as signing an unintended message).

In many DAPs the message sign mechanism is used to authenticate its users. The user can accidentally sign the malicious message and allow account takeover on the DAP, which doesn't strictly validate the entire message content. Users might think that they're signing (authenticating) into one website (live.com), but in reality, the attacker will use that signed message to impersonate the user on another (evil.com) domain.

Complexity

High. An attacker needs to trick a user to visit a malicious page and sign the message. In some cases, operating system settings can impact the layout rendered to the victim, changing the success rate of the attack. For example, MacOS has `Show Scroll Bars` settings which can hide or always show a scroll bar to the user. On a tested Android device, the scroll bars were always hidden.

Considering the social engineering attack scenario and its impact, the issue was rated as having a Medium severity.

Remediation

Show to the user a warning message about non-ASCII characters in the message requested for signing. Alternatively, Hex-encode non-visible characters, so they are always visible to the user. Always show a scrollbar indicating that not a whole message is currently visible to the user.

Additionally, it is recommended to show a warning message in case of the presence of any Unicode characters, which changes the direction of the text.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

This remediation progress was verified on Brave Desktop 1.47.55.

The affected source code has been updated to warn about unicode characters that might be employed in UI redressing attacks. The result has been also confirmed by dynamic testing.

Resources

- MITRE, CWE-451: User Interface (UI) Misrepresentation of Critical Information
  https://cwe.mitre.org/data/definitions/451.html

BRA-Q322-8. The Wallet Details Are Exposed On brave://prefs-internals	
Severity	**Informational**
Vulnerability Class	Information Exposure
Component	brave://prefs-internals
Status	Open - Risk Accepted

Description

Brave is an open-source web browser based on the Chromium web browser. The Chromium browser has its own internal pages, one of which is `prefs-internals.` This page contains the current profile's data preferences in a JSON format.

Considering the presence of Brave's embedded wallet, great care should be taken to not leak sensitive information via any internal browser page.

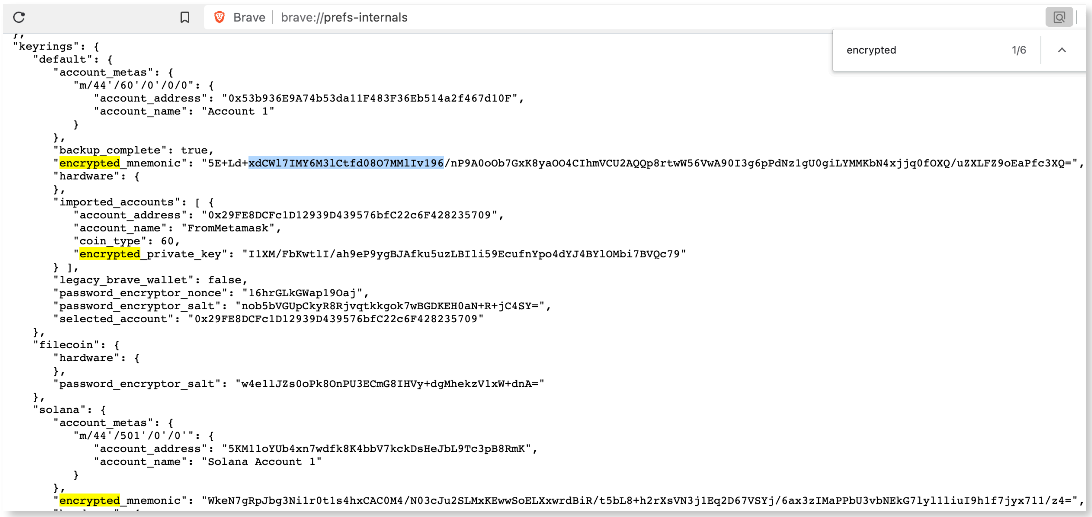
Brave's `brave://prefs-internals` page exposes some wallet information, such as `encrypted_mnemonic`, `encrypted_private_key`, `password_encryptor_nonce`, and `password_encryptor_salt`. The information is present there even when a wallet is locked and certain data, such as account addresses, are not available via the provider objects.

Reproduction Steps

To reproduce the issue, perform the following steps:

1. Lock a Brave Wallet and navigate to the `brave://prefs-internals` page. Note, that the wallet does not have to be unlocked.

2. Search for the wallet object and verify the presence of `encrypted_mnemonic`, `encrypted_private_key`, `password_encryptor_nonce`, and `password_encryptor_salt`



```
"keyrings": {
   "default": {
      "account_metas": {
         "m/44'/60'/0'/0/0": {
            "account_address": "0x53b936E9A74b53da11F483F36Eb514a2f467d10F",
            "account_name": "Account 1"
         }
      },
      "backup_complete": true,
      "encrypted_mnemonic": "5E+Ld+xdCWl7IMY6M3lCtfd08O7MMlIv196/nP9A0oOb7GxK8yaOO4CIhmVCU2AQQp8rtwW56VwA90I3g6pPdNz1gU0giLYMMKbN4xjjq0fOXQ/uZXLFZ9oEaPfc3XQ=",
      "hardware": {
      },
      "imported_accounts": [ {
         "account_address": "0x29FE8DCFc1D12939D439576bfC22c6F428235709",
         "account_name": "FromMetamask",
         "coin_type": 60,
         "encrypted_private_key": "I1XM/FbKwtlI/ah9eP9ygBJAfku5uzLBIli59EcufnYpo4dYJ4BYlOMbi7BVQc79"
      } ],
      "legacy_brave_wallet": false,
      "password_encryptor_nonce": "16hrGLkGWap19Oaj",
      "password_encryptor_salt": "nob5bVGUpCkyR8Rjvqtkkgok7wBGDKEH0aN+R+jC4SY=",
      "selected_account": "0x29FE8DCFc1D12939D439576bfC22c6F428235709"
   },
   "filecoin": {
      "hardware": {
      },
      "password_encryptor_salt": "w4e1lJZs0oPk8OnPU3ECmG8IHVy+dgMhekzV1xW+dnA="
   },
   "solana": {
      "account_metas": {
         "m/44'/501'/0'/0'": {
            "account_address": "5KM1loYUb4xn7wdfk8K4bbV7kckDsHeJbL9Tc3pB8RmK",
            "account_name": "Solana Account 1"
         }
      },
      "encrypted_mnemonic": "WkeN7gRpJbg3Ni1r0t1s4hxCAC0M4/N03cJu2SLMxKEwwSoELXxwrdBiR/t5bL8+h2rXsVN3j1Eq2D67VSYj/6ax3zIMaPPbU3vbNEkG7lyl1liuI9h1f7jyx711/z4=",
```

Impact

Medium. When the attacker obtains encrypted information she can start an offline bruteforce attack and try to recover the private key or the seed phrase. The exposed account data can be used to identify a wallet owner.

Complexity

High. The attacker needs to exploit other vulnerabilities to gain access to the content of the privileged `prefs-internal` page. Considering the high complexity and potential impact, the issue was rated as having Informational severity only.

Remediation

Do not expose wallet's sensitive information on the `prefs-internal` page.

Retesting Results

The risk has been accepted by Brave Software. As a mitigation to local attacks, Brave supports and encourages the use of hardware crypto wallets. As an example, Ledger support is baked in the Brave Wallet by default.

During retesting, Doyensec discovered that the `encrypted_private_key` value has been removed from the `prefs-internal` page but the `encrypted_mnemonic`, `password_encryptor_nonce` and `password_encryptor_salt` values are still visible.

Resources

- MITRE, CWE-1295: Debug Messages Revealing Unnecessary Information
  https://cwe.mitre.org/data/definitions/1295.html

BRA-Q322-9. Missing Blurring for Recovery Phrase Screen on iOS

Severity	**Low**
Vulnerability Class	Information Exposure
Component	Brave Wallet | iOS
Status	Closed

Description

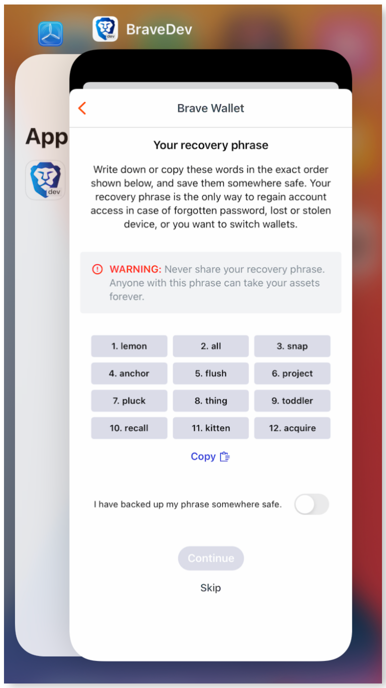
Sensitive application data (e.g., the wallet recovery phrase) can be leaked through screenshots taken by the the operating system or cached during the out of focus state. Blurring the mobile application screen can be used to prevent the sensitive mobile data from being exposed.

Doyensec discovered that Brave Wallet for iOS does not deploy sufficient protections to prevent screens capturing when backgrounding the application.

Reproduction Steps

To reproduce the issue, perform the following steps:

1. Unlock the iOS version of the Brave Wallet and navigate to the `Accounts -> Backup`
2. "Swipe up" , putting the Brave Wallet into the background state
3. "Swipe up" again and look for the Wallet on the list of all open applications
4. Verify that recovery phrase screen is not blurred and is visible to the attacker.
5. The cached files could be found also in `/var/mobile/Containers/Data/Application/<APP_ID>/Library/SplashBoard/Snapshots/com.brave.ios.BrowserBeta - {DEFAULT GROUP}/`



Your recovery phrase

Write down or copy these words in the exact order shown below, and save them somewhere safe. Your recovery phrase is the only way to regain account access in case of forgotten password, lost or stolen device, or you want to switch wallets.

WARNING: Never share your recovery phrase. Anyone with this phrase can take your assets forever.

1. lemon	2. all	3. snap
4. anchor	5. flush	6. project
7. pluck	8. thing	9. toddler
10. recall	11. kitten	12. acquire

I have backed up my phrase somewhere safe.

Impact

High. A recovery phrase is a very sensitive piece of information. If an attacker can get the recovery phrase, she will have access to all of the crypto assets associated with the wallet.

Complexity

High. To extract a cached version of the recovery phrase screen, the attacker will need to compromise the user's device via installing a malicious application or exploiting a critical vulnerability.

Remediation

Blur the recovery phrase screen whenever the wallet switches to the background state. Implement the same mechanism that is already used for the private key screen.

Retesting Results

During retesting, Doyensec confirmed that the issue has been fixed and the vulnerability can no longer be exploited.

This remediation progress was verified on Brave iOS 1.45.116.

The application has been updated to deploy transition-to-background protections in order to avoid accidental data leakage.

Resources

- OWASP, "Mobile Security Testing Guide"
  https://github.com/OWASP/owasp-mstg/blob/master/Document/0x06d-Testing-Data-Storage.md

Appendix A - Vulnerability Classification

Vulnerability Severity	**Critical**
	High
	Medium
	Low
	Informational
Vulnerability Class	Components With Known Vulnerabilities
	Covert Channel (Timing Attacks, etc.)
	Cross Site Request Forgery (CSRF)
	Cross Site Scripting (XSS)
	Denial of Service (DoS)
	Information Exposure
	Injection Flaws (SQL, XML, Command, Path, etc)
	Insecure Design
	Insecure Direct Object References (IDOR)
	Insufficient Authentication and Session Management
	Insufficient Authorization
	Insufficient Cryptography
	Memory Corruption (Buffer and Integer Overflows, Format String, etc)
	Race Condition
	Security Misconfiguration
	Server-Side Request Forgery (SSRF)
	Unrestricted File Uploads
	Unvalidated Redirects and Forwards
	User Privacy
	Time-of-Check to Time-of-Use (TOCTOU)
	Insecure Deserialization

Appendix B - Remediation Checklist

The table below can be used to keep track of your remediation efforts inside this report. Mark the boxes when a fix has been implemented for the vulnerability.

✓	Sanitize the method name by ensuring no null byte or new line characters are present inside
✓	Protect with a password verification window all sensitive functionalities such as private key retrieval, seed backup, account removal, etc
✓	Always show complete blockchain name
✓	Implement a secure version of a vector that always clears the internal memory or ensure no copies of the private key are ever made
✓	Increase the number of iterations for PBKDF2 to the recommended value
✓	Ensure the exposed provider objects have frozen prototypes
✓	Show to the user a warning message about non-ASCII characters in the message requested for signing
☐	Do not expose wallet's sensitive information on `prefs-internal` page
✓	Blur the recovery phrase screen whenever wallet switches to the background state (iOS only)

When done patching the listed vulnerabilities, many clients find it worthwhile to perform a retest. During a retest, Doyensec researchers will attempt to bypass and subvert all implemented fixes. Retests usually take one or two days. Please reach out if you'd like more information on our retesting process.