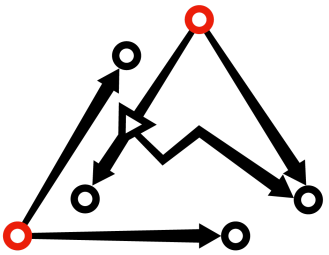

Least Authority

PRIVACY MATTERS

Kotlin and Swift Payment URI Prototypes

Security Audit Report

ZCG

Final Audit Report: 11 June 2025

Table of Contents

Overview

Background

ZCG has requested that Least Authority perform a security audit of the Kotlin and Swift Payment URI prototypes, as defined in ZIP-321.

Project Dates

- **February 10, 2025 - February 26, 2025:** Initial Code Review *(Completed)*
- **February 28, 2025:** Delivery of Initial Audit Report *(Completed)*
- **June 10, 2025:** Verification Review *(Completed)*
- **June 11, 2025:** Delivery of Final Audit Report *(Completed)*

Review Team

- Nikos Iliakis, Security Researcher and Engineer
- Michael Rogers, Security Researcher and Engineer
- Burak Atasoy, Project Manager
- Jessy Bissal, Technical Editor

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Kotlin and Swift Payment URI prototypes followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- kotlin-payment-uri: https://github.com/zecdev/zcash-kotlin-payment-uri
- swift-payment-uri: https://github.com/zecdev/zcash-swift-payment-uri

Specifically, we examined the following Git revisions for our initial review:

- zcash-kotlin-payment-uri: `2a68ef50318930c49748caa9fd74c8bc89b72337`
- zcash-swift-payment-uri: `c24dd72a60cf0f2bca4a9a6503ef63e079c96f77`

For the verification, we examined the Git revision:

- zcash-kotlin-payment-uri: `5da8f0e48ef2b3a3bf04550790088a97832274dd`
- zcash-swift-payment-uri: `b9a103a821cc91a3b12c5ca6e4d983dcf34b777c`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- https://github.com/LeastAuthority/zcash-kotlin-payment-uri
- https://github.com/LeastAuthority/zcash-swift-payment-uri

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- ZCG Forum Dev Grant Updates:
  https://forum.zcashcommunity.com/t/grant-updates-zcash-wallet-community-developer/45562/5
- ZIP-321: Payment Request URIs
  https://zips.z.cash/zip-0321

In addition, this audit report references the following documents:

- RFC 2119 | Key Words for Use in RFCs to Indicate Requirement Levels
  https://www.rfc-editor.org/rfc/rfc2119
- Zcash Protocol Specification, Version 2024.5.1-112-gcf7a5c | Section 5.6: Encoding of Addresses and Keys:
  https://zips.z.cash/protocol/protocol.pdf#addressandkeyencoding
- ZIP-316 | Unified Addresses and Viewing Keys:
  https://zips.z.cash/zip-0316
- RFC 2234 | Augmented BNF for Syntax Specifications | ABNF, section 6.1: Core Rules:
  https://www.rfc-editor.org/rfc/rfc2234#section-6.1
- Jazzer: Coverage-guided, in-process fuzzing for the JVM
  https://github.com/CodeIntelligenceTesting/jazzer
- OWASP dependency-check-gradle plugin
  https://plugins.gradle.org/plugin/org.owasp.dependencycheck

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Vulnerabilities within each component and whether the interaction between the components is secure;
- Whether requests are passed correctly to the network core;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution;
- Protection against malicious attacks and other ways to exploit;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security audit of a library for processing Zcash Payment URIs, as defined in ZIP-321. The core functionality of this library, which is implemented in Kotlin for Android and in Swift for iOS, is to construct and parse payment request URIs, enabling users to express payment intents in a standardized format. This format can be recognized by wallets and other applications within the Zcash ecosystem, facilitating transactions through links or QR codes.

During our review of ZIP-321, we examined its implementation for correctness, security, and adherence to the specification. We assessed URI parsing, address validation, amount handling, memo encoding, required parameters, duplicate field handling, and security resilience. While our evaluation indicated that the implementation generally complies with the ZIP-321 specification, we identified several deviations, as reported below (Issue A, Issue B, Issue C, Issue D, Issue E, Issue F, Issue G, and Issue H).

System Design

Our team examined the Kotlin and Swift implementations of the Zcash Swift Payment URI library, which is designed with a modular architecture that focuses on the construction and parsing of Zcash Payment URIs, as per ZIP-321. It includes core data structures for encapsulating payment details, a parser for interpreting URI strings, and a renderer for generating URIs from structured data. During our review, we found that security has been taken into consideration as demonstrated by robust validation, error handling, and a suite of unit tests, which supports reliability and system integrity.

Our team notes that the Kotlin and Swift implementations of the library do not cover all of the recommendations in ZIP-321, some of which require extensive semantic validation of the payments encoded in a payment URI to ensure that they can form a valid transaction. Such validation would require implementations of ZIP-321 to be able to parse and interpret all types of Zcash addresses, as well as being aware of other limitations on the validity of transactions, such as limits on the maximum number of payments per transaction for any combination of address types.

We have not recommended modifying the implementation to accommodate these recommendations in cases where doing so would substantially increase its complexity or scope. However, we have proposed one modification that would accommodate a recommendation of ZIP-321 with a reasonable amount of effort (Suggestion 6), and two changes to meet the strict requirements of ZIP-321, even though doing so may increase the complexity and scope of the implementation (Issue A and Issue B). This reasoning is based on the meanings attributed to the uppercase “SHOULD” and “MUST” in RFC 2119, as stated in ZIP-321.

Our analysis indicates that a more effective solution to the problem of semantic validation is to divide ZIP-321 into two standards: one defining the syntax of payment URIs and the other defining the conditions for multiple payments to form a valid transaction. This would allow for a separation of concerns between input handling and transaction marshalling. However, we recognize that changes to the standard are beyond the scope of this review.

Dependencies

Our team analyzed the Swift and Kotlin codebases for dependencies with known vulnerabilities. While no vulnerable dependencies were detected in Swift, we identified four in Kotlin, which we recommend updating (Suggestion 8).

Code Quality

The Swift codebase is well-structured, adhering to standard Swift practices with a focus on maintainability and readability. The codebase effectively uses Swift's Error protocol for error handling and includes tests to verify correctness and reliability. Coding standards are enforced through `SwiftLint`, promoting consistent style and best practices. Additionally, the code makes idiomatic use of Swift features, such as optionals and closures. While generally well-implemented, the codebase could be improved by addressing outstanding TODOs and would benefit from regular updates to maintain its quality.

The Kotlin codebase is also generally well-structured and well-written, although we found several implementation bugs that indicate that not all of the parameter combinations permitted by ZIP-321 have been tested.

Tests

The codebase includes tests that verify the parsing logic's functionality. These tests cover various scenarios to confirm correctness and reliability. However, numerous cases are not covered by tests. Better test coverage could have detected Issue C, Issue E, Issue F, Issue H, Issue I, and Issue J. We also recommend adding fuzzing tests (Issue L).

Our team has added a branch to Least Authority's fork of the Kotlin repository that contains tests for several issues identified in this report, to help with verifying that the issues have been resolved and to avoid regressions in the future.

Documentation and Code Comments

The codebase lacks extensive documentation. However, given its small size, the existing comments and structure provide sufficient clarity for understanding and maintaining the code. Additionally, the codebases are moderately commented, which helps clarify their structure and functionality, though not every aspect is covered extensively.

Scope

The scope of this review was sufficient and included all security-relevant components.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: URI Parser Accepts Sprout Addresses	Resolved
Issue B: URI Parser Allows Transparent Unified Addresses To Have Memos	Resolved
Issue C: [Kotlin] Parser Decodes Plus Sign to Space in Parameter Values	Resolved
Issue D: [Kotlin] Parser Uses System's Default Character Set When Decoding Base64	Resolved
Issue E: [Kotlin] Parser Does Not Allow Amount To Be Zero	Resolved
Issue F: Parser Does Not Allow Amount to Be Omitted	Resolved
Issue G: Parser Accepts Any Unicode Letter or Digit in Address	Resolved
Issue H: Parser Does Not Accept Parameters Without Values	Resolved
Issue I: [Kotlin] Encoder Produces Invalid URI for Payment Request With	Resolved

Multiple Payments	
Issue J: [Kotlin] Encoder Produces Invalid URI for Single Address With No Other Parameters	Resolved
Issue K: [Kotlin] Parser Does Not Declare the Exceptions It Throws	Resolved
Issue L: [Kotlin] Parser Throws Unexpected Exception Types for Various Invalid Inputs	Partially Resolved
Suggestion 1: [Kotlin] Use Package Declarations Consistently	Implemented
Suggestion 2: [Kotlin] Correctly Implement Deserialization for Singleton Objects	Implemented
Suggestion 3: [Kotlin] Remove Redundant Code	Not Implemented
Suggestion 4: [Kotlin] Correctly Define Character Sets	Not Implemented
Suggestion 5: [Kotlin] Use Idiomatic Kotlin	Partially Implemented
Suggestion 6: Pass Context to the Parser	Not Implemented
Suggestion 7: Add Configuration for Future req Parameters	Not Implemented
Suggestion 8: [Kotlin] Use a Plugin To Check for Vulnerable Dependencies	Partially Implemented

Issue A: URI Parser Accepts Sprout Addresses

Location

`zecdev/zip321/ZIP321.kt#L187`

`Sources/ZcashPaymentURI/Parser.swift#L412`

Synopsis

The parser accepts Sprout addresses, which the specification states must not be accepted.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

ZIP-321 states that, "*Sprout addresses MUST NOT be supported in payment requests.*" The URI parser performs no validation of addresses other than checking that they contain only Unicode letters and digits. Therefore, it does not prevent Sprout addresses from being used in payment requests.

Mitigation

The parser optionally accepts a function from the calling code that can be used to validate addresses. Applications that use the parser can work around this issue by supplying an address validation function that rejects Sprout addresses.

Remediation

There are two options for remediation, either of which would be suitable:

1. The URI parser could specifically reject Sprout addresses by recognizing their prefixes, which are described in section 5.6 of the Zcash Protocol Specification.
2. The URI parser could specifically accept address types that are known to be allowed by ZIP-321 and reject all other addresses. If this option is chosen, then the parser should be updated whenever new address types are added to the Protocol Specification. ZIP-321 states that, "*New address formats may be added to [the Protocol Specification] in future, and these SHOULD be supported whether or not this ZIP is updated to explicitly include them.*"

Status

The Kotlin and Swift team has updated the Kotlin and Swift codebases to address this issue.

Verification

Resolved.

Issue B: URI Parser Allows Transparent Unified Addresses To Have Memos

Location

`zip321/model/RecipientAddress.kt#L36`

`Sources/ZcashPaymentURI/RecipientAddress.swift#L32`

Synopsis

The parser does not consider transparent unified addresses to be transparent. As a result, it allows them to have memos.

Impact

This is an implementation correctness issue rather than a security vulnerability. Since a Zcash transaction containing a memo for a transparent address cannot be constructed, the memo field will not be sent in the clear.

Technical Details

ZIP-321 states that, "*Parsers MUST consider the entire URI invalid if the address associated with the same* `paramindex` *[as the memo parameter] does not permit the use of memos (i.e. it is a transparent address).*" The URI parser attempts to ensure this by checking whether the address associated with a memo parameter is transparent. This is achieved by checking whether the first character of the address is "`t`." However, this check fails to detect universal addresses with transparent receivers, as described in ZIP-316.

Remediation

In order to reliably detect transparent addresses as required by ZIP-321, we recommend modifying the URI parser logic to decode any unified address that has an associated memo field and check whether any receiver of the address is transparent.

Status

The Kotlin and Swift team has updated the Kotlin and Swift codebases to address this issue.

Verification

Resolved.

Issue C: [Kotlin] Parser Decodes Plus Sign to Space in Parameter Values

Location

`zip321/extensions/StringEncodings.kt#L20`

Synopsis

The URI parser decodes the plus sign character to the space character in parameter values. This results in an incorrect representation of the parameter value being returned to the calling code. If the incorrect representation is converted back into a payment URI, the space character is encoded as the string "%20." Consequently, the re-encoded URI is not semantically equivalent to the original URI.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

The URI parser uses the Java standard library method `URLDecoder#decode(String, String)` to decode parameter values containing percent-encoded characters. However, this method also replaces plus sign characters with space characters. ZIP-321 defines the plus sign character to be one of the characters allowed in encoded parameter values, but does not define it as an encoding of the space character. Therefore, the conversion performed by the standard library method is not suitable for this purpose.

Remediation

When decoding parameter values, we recommend using an approach similar to the one used for encoding: characters explicitly allowed in encoded parameter values by ZIP-321, including the plus sign, should not be modified by encoding or decoding.

Status

The Kotlin and Swift team has updated the Kotlin codebase to address this issue by implementing decoding of percent-encoded characters without relying on the Java standard library method.

Verification

Resolved.

Issue D: [Kotlin] Parser Uses System's Default Character Set When Decoding Base64

Location

`zip321/model/MemoBytes.kt#L68`

Synopsis

The URI parser's behavior depends on the system's default character set. Parsing of URIs with memo fields will fail on systems with certain unusual character sets.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

ZIP-321 specifies the use of `Base64` for encoding memo fields. The URI parser uses the Java standard library method `Base64.Decoder#decode(byte[])` to decode `Base64`-encoded memo fields, after substituting certain characters as required by ZIP-321 and converting the resulting string to a byte array.

The conversion from string to byte array employs the Kotlin standard library method `String#toByteArray(Charset)`, which uses the specified character set to determine how to represent the string as a byte array. The URI parser passes the system's default character set to this method.

On systems with certain unusual character sets, such as UTF-16 or Windows `1026`, the use of the system's default character set will produce a byte array that the method `Base64.Decoder#decode(byte[])` does not recognize. This will cause the parsing of the URI to fail.

Remediation

Instead of passing the system's default character set to the method `String#toByteArray(Charset)`, we recommend modifying the URI parser logic to omit the optional parameter value, allowing the method to use the default parameter value `Charsets.UTF_8`, which is suitable for this purpose and will work in the same way across all systems.

Status

The Kotlin and Swift team has updated the Kotlin codebase to address this issue by using the default UTF-8 character set.

Verification

Resolved.

Issue E: [Kotlin] Parser Does Not Allow Amount To Be Zero

Location

`zip321/model/NonNegativeAmount.kt#L50`

`zip321/parser/Parser.kt#L293`

Synopsis

The URI parser requires payment amounts to be strictly greater than zero, although the Zcash network accepts payment amounts that are equal to zero.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

The URI parser uses a class called `NonNegativeAmount` to represent payment amounts. Although the name of this class (and the related error type, `NegativeAmount`) would seem to suggest that amounts must be greater than or equal to zero, the implementation actually requires amounts to be strictly greater than zero. This prevents payment URIs from being used for certain use cases, such as sending a memo without transferring any funds.

Additionally, the rejection of payment amounts that are equal to zero causes parsing errors for certain URIs that omit payment amounts, as the parser tries to substitute an amount of zero for the omitted amount, and then rejects this as an invalid amount.

Remediation

We recommend updating the logic to allow payment amounts to be greater than or equal to zero.

Status

The Kotlin and Swift team has updated the Kotlin codebase to address this issue by accepting amounts equal to zero.

Verification

Resolved.

Issue F: Parser Does Not Allow Amount to Be Omitted

Location

`zip321/model/Payment.kt#L4`

`zip321/parser/Parser.kt#L293`

`Sources/ZcashPaymentURI/Model.swift#L19`

`Sources/ZcashPaymentURI/Parser.swift#327`

Synopsis

The parser rejects payment URIs that do not have an amount specified for every payment, except in the special case of a URI that contains a single address in the hierarchical part of the URI, with no URI parameters.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

The URI parser uses a class called `Payment` to represent each payment in the URI being parsed. This class has a field of type `NonNegativeAmount` that is not allowed to be null, so constructing an instance of the `Payment` class requires an amount to be specified. However, ZIP-321 allows payment amounts to be omitted from URIs. (This makes it possible for a user to publish a URI for receiving payments, without disclosing the amount of any specific payment that may be received.)

When the parser parses a URI, it collects the parameters that relate to each payment and then uses those parameters to construct an instance of the `Payment` class. If no `amount` parameter has been specified, then an amount of zero is used (see Issue E). However, an amount of zero is not semantically equivalent to an omitted amount.

Remediation

We recommend allowing the `nonNegativeAmount` field of the `Payment` class to be null, indicating that no amount has been specified for the payment.

Status

The Kotlin and Swift team has updated the Kotlin and Swift codebases to address this issue.

Verification

Resolved.

Issue G: Parser Accepts Any Unicode Letter or Digit in Address

Location

`zip321/parser/AddressParser.kt#L6`

Synopsis

The URI parser in Kotlin accepts any Unicode letter or digit as a valid character in a Zcash address, whereas the syntax specification in ZIP-321 specifies that only `ASCII` letters or digits should be allowed.

In Swift, in the construction of a request, there is no validation of characters passed to the address input for constructing a URI. The URI parsing works as intended.

Impact

The issue makes it possible to construct a payment URI that may appear at first glance to have different semantics from those it actually possesses. However, such a URI could not be used to create a valid Zcash transaction that would result in funds being transferred; therefore, it is unlikely that there will be any significant security impact.

Preconditions

There are no preconditions for creating an invalid payment URI.

Feasibility

An invalid payment URI can be created easily by copying and pasting Unicode characters, but it does not appear feasible to use an invalid payment URI to create a valid Zcash transaction.

Technical Details

The URI parser validates addresses by checking whether all the characters they contain are letters or digits. However, the check that is used for this purpose is designed for use with Unicode, so the parser accepts any Unicode letter or digit as a valid character in an address.

For example, the following URI appears at first glance to have a payment amount of `1.234` ZEC; however, the amount is actually `20` ZEC. The Unicode letters “?”, “ ”, and “ ” are used in place of the delimiters “?”, “=”, and “.”, so all characters up to and including “`amount 1 234`” are handled by the parser as a single address parameter. The unrecognised parameter `Have` is accepted by the parser, as required by ZIP-321, which allows the true payment amount to be concealed among similar-looking text:

```
zcash:tmEZhbWHTpdKMw5it8YDspUXSMGQyFwovpU?amount 1 234?message=Thanks%20for%20
your%20payment%20for%20the%20correct%20&amount=20&Have=%20a%20nice%20day
```

The address that is extracted from the URI by the parser is not a valid Zcash address. Therefore, it does not appear feasible to exploit this issue by tricking users into transferring larger amounts than intended.

The syntax specification in ZIP-321 uses the `ALPHA` and `DIGIT` productions to specify which characters are valid in addresses. The `DIGIT` production matches the digits `0-9`. ZIP-321 refers to RFC 3986 for the definition of the `ALPHA` production. RFC 3986, in turn, refers to RFC 2234, which defines the `ALPHA` production as matching the letters A-Z and a-z.

Remediation

We recommend modifying the parser logic to only accept `ASCII` letters and digits in addresses.

Status

The Kotlin and Swift team has updated the Kotlin and Swift codebases to address this issue.

Verification

Resolved.

Issue H: Parser Does Not Accept Parameters Without Values

Location

`zip321/parser/Parser.kt#L73`

`zip321/parser/Parser.kt#L102`

`Sources/ZcashPaymentURI/Parser.swift#L97`

Synopsis

The URI parser does not accept parameters without values, which are allowed by ZIP-321.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

The syntax specification in ZIP-321 defines five named parameters (`address`, `amount`, `label`, `memo`, and `message`), as well as the syntax for any parameters that may be defined in the future. The syntax for these future parameters uses the productions `reqparam` and `otherparam`, which respectively refer to required parameters and other parameters. Each of these productions matches a parameter name, an optional parameter index, and an optional parameter value preceded by an equals sign. However, the URI parser does not permit the parameter value and equals sign to be omitted. This may cause valid URIs to be rejected.

Remediation

We recommend updating the parsing logic to accept required parameters and other parameters that do not have values.

The presence of a required parameter (with or without a value) already correctly causes the URI to be rejected, as required by ZIP-321. However, we recommend testing the parser with required parameters that do not have values in order to verify that it emits the correct error type when such a parameter is encountered (i.e., that it rejects the URI due to the parameter being required, rather than due to the absence of a value).

Status

The Kotlin and Swift team has updated the Kotlin and Swift codebases to address this issue.

Verification

Resolved.

Issue I: [Kotlin] Encoder Produces Invalid URI for Payment Request With Multiple Payments

Location

zip321/Render.kt#L115

zip321/ZIP321.kt#L130

Synopsis

When encoding a payment request with multiple payments and using the default formatting options, the URI encoder omits the "?" character from the URI. The resulting URI is invalid.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

The URI encoder supports various formatting options: The address of the first payment may be encoded in the hierarchical part of the URI or in a query parameter, and the encoder may use a parameter index for every payment or omit the index for the first payment. The encoder therefore inserts the "?" character at different places in the URI depending on the formatting options that are used.

When encoding multiple payments, if a start index has been specified, then the encoder does not insert the "?" character immediately following the scheme part of the URI ("`zcash:`"). However, the default formatting options for encoding multiple payments specify a start index, resulting in the "?" character being omitted.

Remediation

We recommend correcting the logic for encoding multiple payments and adding tests that check the encoding of multiple payments with all available formatting options.

Status

The Kotlin and Swift team has updated the Kotlin codebase to address this issue.

Verification

Resolved.

Issue J: [Kotlin] Encoder Produces Invalid URI for Single Address With No Other Parameters

Location

zip321/ZIP321.kt#L146

Synopsis

When encoding a payment URI containing a single address and no other parameters, the encoder omits the "?" character from the URI when non-default formatting options are used. The resulting URI is invalid.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

The URI encoder provides a convenience function for encoding a payment URI containing a single address and no other parameters. When using the default formatting options, this function correctly omits the "?" character, which is not needed because the address is encoded in the hierarchical part of the URI, and there are no query parameters. However, when using non-default formatting options that encode the address as a query parameter, the encoder incorrectly omits the "?" character between the scheme part of the URI ("`zcash:`") and the query parameter.

Remediation

We recommend correcting the encoding logic and adding tests for this convenience function that cover all available formatting options.

Status

The Kotlin and Swift team has updated the Kotlin codebase to address this issue.

Verification

Resolved.

Issue K: [Kotlin] Parser Does Not Declare the Exceptions It Throws

Location

`zip321/ZIP321.kt#L187`

Synopsis

The URI parser throws several types of exceptions, some of which are checked exceptions. Since these are not declared with a `@Throws` annotation, Java code that calls the parser cannot catch the checked exceptions.

Impact

This is an implementation correctness issue rather than a security vulnerability.

Technical Details

Libraries written in Kotlin may be called from Kotlin or Java code. Java distinguishes between checked and unchecked exceptions: Java methods must declare the types of any checked exceptions that they throw. It is an error for Java code to attempt to catch checked exception types that are not declared in this manner. In contrast, Kotlin does not require methods to declare the types of any checked exceptions that they throw, but the `@Throws` annotation may optionally be used to make such a declaration, either for documentation purposes or for interoperability with Java. If a Kotlin method throws checked exceptions but does not declare them in a `@Throws` annotation, then Java code that calls the Kotlin method cannot attempt to catch those exception types.

The URI parser has `@Throws` annotations on many of its methods, but the top-level URI parsing method does not have such an annotation. This prevents Java code from catching the checked exception types that it throws, including `ZIP321.Errors` and `NonNegativeAmount.AmountError`.

Mitigation

Java code that uses the parser can circumvent this issue by catching a generic error type such as `Exception`; however, this does not allow the calling code to distinguish between errors of different types.

Remediation

We recommend adding a suitable `@Throws` annotation to the top-level parsing method, `ZIP321.request(String, ((String) -> Boolean)?)`.

Status

The Kotlin and Swift team has updated the Kotlin codebase to address this issue by explicitly declaring that the `Errors` type may be thrown.

Verification

Resolved.

Issue L: [Kotlin] Parser Throws Unexpected Exception Types for Various Invalid Inputs

Location

`zip321/ZIP321.kt#L187`

Synopsis

In addition to the exception types defined in the codebase, the parser throws unexpected exception types for certain invalid inputs. These exception types include:

- `com.copperleaf.kudzu.parser.ParserException`; and
- `java.lang.IllegalArgumentException`.

Impact

While this is primarily an implementation correctness issue rather than a security vulnerability, it could cause an application using the parser to crash when parsing an invalid URI.

Technical Details

Our team used a fuzzing library to test the parser's response to a large number of invalid inputs. In addition to the expected exception types, some inputs caused the parser to throw `com.copperleaf.kudzu.parser.ParserException` or `java.lang.IllegalArgumentException`.

One example of an input that causes a `ParserException` to be thrown is the empty string. An `IllegalArgumentException` is thrown if a percent sign in a parameter value is not followed by two hexadecimal digits.

Mitigation

Java or Kotlin code that uses the parser can circumvent this issue by catching generic error types, such as `Exception` or `RuntimeException`; however, this does not allow the calling code to distinguish between errors of different types.

Remediation

We recommend catching these exceptions close to the point where they are thrown and re-throwing them as error types that are defined in the codebase (and declared in a `@Throws` annotation, as explained in Issue K). This will provide users of the parser with clarity about which exception types to expect.

We further recommend adding fuzzing tests to the codebase to catch any similar issues that may arise in the future. The Jazzer library is straightforward to integrate into Java and Kotlin codebases. Our team has added a branch to Least Authority's clone of the Kotlin repository to demonstrate this integration.

Status

The Kotlin and Swift team has updated the Kotlin codebase to catch the unexpected error type `ParserException` and rethrow it as the expected error type, `ZIP321.Errors`. However, the unexpected error type `IllegalArgumentException` is still thrown, including in recently added code in the file `QCharCodec.kt`. Furthermore, code in the files `Param.kt` and `NonNegativeAmount.kt`, which have been updated since the initial report may now throw the unexpected error types `IllegalArgumentException` and `NumberFormatException`.

Verification

Partially Resolved.

Suggestions

Suggestion 1: [Kotlin] Use Package Declarations Consistently

Location

`zip321/model/MemoBytes.kt#L1`

`zip321/model/NonNegativeAmount.kt#L1`

`zip321/model/Payment.kt#L1`

`zip321/model/PaymentRequest.kt#L1`

`zip321/model/RecipientAddress.kt#L1`

Synopsis

Several files in the Kotlin codebase lack package declarations, indicating that classes declared in those files belong to the default package. Use of the default package is discouraged, as it may lead to name conflicts.

Mitigation

We recommend adding package declarations to all files.

Status

The Kotlin and Swift team has implemented the recommendation by updating the package declarations.

Verification

Implemented.

Suggestion 2: [Kotlin] Correctly Implement Deserialization for Singleton Objects

Location

`zip321/model/MemoBytes.kt#L13`

zip321/model/NonNegativeAmount.kt#L37

zip321/model/RecipientAddress.kt#L4

zip321/ZIP321.kt#L15

Synopsis

The Kotlin codebase effectively utilizes sealed classes for type-safe error handling. However, types that inherit from `Exception` inherit the `Serializable` interface, which does not have sensible default behavior for Kotlin's singleton objects. Attempting to deserialize these types will result in unexpected behavior, as the deserialized object will not be identical to the singleton.

Mitigation

Singleton objects should override the `readResolve` method to return the singleton instance. For example:

```
object InvalidBase64 : Errors() {

   private fun readResolve(): Any = InvalidBase64

}
```

Status

The Kotlin and Swift team has implemented the suggestion by adding `readResolve` methods.

Verification

Implemented.

Suggestion 3: [Kotlin] Remove Redundant Code

Location

zip321/model/NonNegativeAmount.kt#L73

zip321/model/Payment.kt#L13

zip321/parser/IndexedParameter.kt#L7

zip321/model/RecipientAddress.kt#L1

zip321/parser/CharsetValidations.kt#L5

Synopsis

The `NonNegativeAmount` class contains redundant code, with two ways to create a `NonNegativeAmount` from a `BigDecimal`, two ways to create a `NonNegativeAmount` from a `String`, and a function that creates a `BigDecimal` from a `String` while checking some but not all of the `NonNegativeAmount` constraints. This redundancy makes it difficult to reason about which constraints have been checked in each case.

The data classes `Payment` and `IndexedParameter` override the `equals(Any?)` and `hashCode` methods. However, the implementations appear to have the same semantics as those that would be generated automatically for data classes, which renders the use of these methods redundant.

The file `RecipientAddress.kt` contains an unused type alias, `RequestParams`.

The `CharsetValidations` class contains definitions of the `Base58` and `Bech32` character sets that are unused, although these may be intended for use by applications that use the parser, which are allowed to supply their own address validation functions.

Mitigation

We recommend removing redundant and unused code to improve the readability and maintainability of the codebase.

Status

The Kotlin and Swift team has updated some of the relevant Kotlin code. The `NonNegativeAmount` class still contains some redundant code: an unused constructor, a constructor used only in tests, and an extension function used only in tests. We recommend removing this code so that the execution paths covered by the tests match those used in production. Additionally, the redundant code previously noted in other classes has not been removed.

Verification

Not Implemented.

Suggestion 4: [Kotlin] Correctly Define Character Sets

Location

`zip321/parser/CharsetValidations.kt#L18`

`zip321/parser/CharsetValidations.kt#L42`

Synopsis

The `CharsetValidations` class contains definitions of several character sets used by the parser. The definition of the `Bech32` character set incorrectly includes the characters "b" and "`i`", while the definition of the unreserved character set incorrectly includes the character "`!`".

These errors do not have any functional impact at present. The `Bech32` character set is unused, as noted in Suggestion 3, while the unreserved character set is only used in a context where it is combined with another character set that already contains the character "`!`".

Mitigation

Although these errors do not presently have any functional impact, we recommend correcting them to avoid any potential functional impact in the future as the codebase evolves.

Status

The Kotlin and Swift team has not updated the relevant Kotlin code to implement this suggestion.

Verification

Not Implemented.

Suggestion 5: [Kotlin] Use Idiomatic Kotlin

Location

`zip321/extensions/StringEncodings.kt#L3`

zip321/parser/Param.kt#L19

zip321/parser/Param.kt#L140

Synopsis

While the codebase generally demonstrates effective utilization of Kotlin idioms, there are a few instances where this could be improved. Qualified type names are used in various instances, likely as a consequence of IDE-assisted refactoring. Some functions declare nullable return types but never actually return `null`, which forces callers to handle the null case unnecessarily.

Mitigation

We recommend replacing qualified type names with imports. Constructs of the form "`(variable as? Type) != null`" can be replaced with "`variable is Type`". Fields, parameters, and return types should generally not be nullable unless the null case is needed.

Status

The Kotlin and Swift team has updated the Kotlin codebase to implement this suggestion by using idiomatic Kotlin. Our team considers this a partial implementation of the suggestion, as the newly added code contains similar non-idiomatic usages to those noted in the initial report, such as the use of fully-qualified names instead of imports.

Verification

Partially Implemented.

Suggestion 6: Pass Context to the Parser

Synopsis

According to ZIP-321, *"If the context of whether the payment URI is intended for Testnet or Mainnet is available, then each address SHOULD be checked to be for the correct network."* However, the library currently lacks a mechanism to specify whether the transactions are intended for mainnet or testnet.

Mitigation

We recommend creating a variable (that can be optional) to set the context of the parser and specify if it is intended to be used for testnet or mainnet, to allow the parser to perform the proper checks.

Status

The Kotlin and Swift team has updated the Kotlin and Swift codebases to implement the suggestion.

Verification

Implemented.

Suggestion 7: Add Configuration for Future req Parameters

Synopsis

According to ZIP-321, in the forward compatibility section, *"Variables which are prefixed with a req- are considered required. If a parser does not recognize any variables which are prefixed with req-, it MUST consider the entire URI invalid. Any other variables that are not recognized, but that are not prefixed with a req-, SHOULD be ignored."* Hence, all variables that currently start with "`req-`" are considered invalid.

Mitigation

To enhance forward compatibility and flexibility in handling `req-` parameters, we recommend modifying the constructor to accept a list of recognized `req-` parameters. This would allow developers to define which `req-` prefixed parameters are supported by their implementation while ensuring unrecognized `req-` parameters result in a validation failure, as required by ZIP-321, without the need to update the parser.

Status

The Kotlin and Swift team has not updated the relevant Kotlin code to implement this suggestion.

Verification

Not Implemented.

Suggestion 8: [Kotlin] Use a Plugin To Check for Vulnerable Dependencies

Location

`build.gradle.kts`

Synopsis

Our team scanned the Swift and Kotlin codebases for dependencies with known vulnerabilities. No vulnerable dependencies were found for Swift; however, for Kotlin we identified the following vulnerabilities:

Dependency name	Version	Known vulnerabilities
guava	31.1	CVE-2023-2976, CVE-2020-8908
ktlint-cli-reporter-checkstyle	1.0.1	CVE-2019-10782, CVE-2019-9658
logback-classic	1.3.5	CVE-2023-6378
logback-core	1.3.5	CVE-2023-6378, CVE-2024-12798, CVE-2024-12801

Mitigation

We recommend keeping all dependencies updated and using a plugin to check for vulnerable dependencies automatically. For this review, our team used the OWASP dependency-check-gradle plugin. This plugin uses data from the National Vulnerability Database, which can be accessed by applying for a free API key.

We note that three of the four vulnerable dependencies are required by the `JLLeitschuh ktlint-gradle` plugin. The latest version of this plugin still depends on a vulnerable version of `ktlint` that was released in `2023`. If possible, we recommend replacing this plugin with an alternative solution that has more recent dependencies.

Status

The dependencies of the Kotlin codebase have been updated since the initial report. However, the `ktlint-gradle` plugin still has two vulnerable dependencies: `ktlint-cli-reporter-checkstyle` version `1.1.0`, and `logback-core` version `1.3.14`. Our team's suggestions to replace this plugin with a

more current alternative and to adopt a plugin that automatically detects vulnerable dependencies have not been implemented.

Verification

Partially Implemented.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and

distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.