X XChat

Security Assessment

October 28, 2025

Prepared for:
Michael Anderson
X

Prepared by: **Scott Arciszewski and Opal Wright**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Jim Miller, Engineering Director, Cryptography
james.miller@trailofbits.com

The following consultants were associated with this project:

Scott Arciszewski, Consultant
scott.arciszewski@trailofbits.com

Opal Wright, Consultant
opal.wright@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
June 3, 2025	Pre-project kickoff call
June 10, 2025	Status update meeting #1
June 17, 2025	Delivery of report draft
June 17, 2025	Report readout meeting
October 28, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

X engaged Trail of Bits to review the security of the XChat end-to-end encrypted messaging and chat service.

A team of two consultants conducted the review from June 3 to June 16, 2025, for a total of four engineer-weeks of effort. Our testing efforts focused on the implementation of the cryptography and protocol design. With full access to source code and documentation, we performed static and dynamic testing of the web and phone clients, using automated and manual processes.

Observations and Impact

The XChat protocol and implementation has multiple serious security vulnerabilities that need to be addressed immediately.

Our review attempted to focus solely on vulnerabilities that have not been documented publicly; beyond the high-severity findings we identified, several independent open-source analyses have identified additional issues. Many of the issues we identified with XChat's system directly correspond to design flaws discovered during Trail of Bits' Direct Messaging design analysis in 2023.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that X take the following steps:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactor that may occur when addressing other recommendations.
- **Consider performing a formal analysis of the XChat protocol.** Several of the design review and implementation findings could have been identified ahead of time by using a formal analysis tool such as Verifpal.
- **Consider temporarily pausing the XChat feature to prioritize addressing the identified security concerns.** We are aware of several high-priority vulnerabilities that require the team's attention. We have separately provided X with long-term recommendations to remediate these high-priority issues. Addressing these security gaps will help protect both your users and your company's reputation.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	3
Medium	1
Low	0
Informational	1
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	6

Project Goals

The engagement was scoped to provide a security assessment of the XChat service. Specifically, we sought to answer the following non-exhaustive list of questions:

- Are the cryptographic primitives selected appropriate for XChat's security goals?
- Are the cryptographic primitives being used correctly?
- Is the current protocol implementation secure?
- Is data at rest stored securely?
- Is data in transit properly authenticated?
- Is data in transit kept confidential?
- Can the server mount attacks on clients?

Project Targets

The engagement involved reviewing and testing the targets listed below.

x-android

Repository	https://github.com/trailofbits/audit-xchat/commits/x-android
Version	`5ef50b60f81936e533b50a46d93742f208bd0ec1`
Type	Kotlin
Platform	Android

x-web

Repository	https://github.com/trailofbits/audit-xchat/commits/x-web
Version	`4dbf0abbcb93726f40fe82355e283350cb2aa0b8`
Type	Javascript / Typescript
Platform	Web

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- Manual review of source code, focusing on the following:
  - Key management
  - Signature generation and validation
  - Cryptographic primitive usage
- Automated analysis using Semgrep

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Security of underlying cryptographic libraries such as BouncyCastle
- The full XChat protocol

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Encrypted conversation keys are not validated	Cryptography	High
2	Long-term identity keys without signatures are not rejected	Cryptography	High
3	Use of truncated SHA-1	Cryptography	Undetermined
4	Conversation keys are not checked for server replay	Cryptography	High
5	Group Key Inconsistencies Can Lead to Invisible Salamander Attacks	Cryptography	Medium
6	Confused Deputy Attack in Avatar / Attachment Encryption	Cryptography	Informational

Detailed Findings

1. Encrypted conversation keys are not validated

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-XCHAT-1

Target: `subsystem/dm/keymanagement/src/javaMain/kotlin/com/x/dms/EncryptionUtils.kt`, `subsystem/dm/keymanagement/src/javaMain/kotlin/com/x/dms/JvmKeyFactory.kt`

Description

Conversation keys are not authenticated after retrieval from the server, which could allow a malicious server to select the conversation key for all participants. While the API has been written to include optional signatures, signatures are not required, and are not validated when present. This finding corresponds closely with finding TOB-TWDM-4 in our Direct Messaging design analysis from 2023.

```
fun constructConversationKeyObject(keyBytes: ByteArray): SecretKey? {
    return try {
        // NOTE: this doesn't actually validate that the key is legit
        SecretKeySpec(keyBytes, CONVERSATION_KEY_ALGORITHM)
    } catch (e: Throwable) {
        XLog.i(KTAG, e) { "Unable to reconstruct cKey" }
        null
    }
}
```

Figure 1.1: Conversation key conversion from bytes

Exploit Scenario

Alice wants to initiate a new conversation with Bob. She attempts to retrieve the conversation key from the server, and as part of this protocol, she requests the conversation metadata from the server. The server generates a new conversation key and encrypts it against each public key registered by Alice and Bob.

Now, even though it is a new conversation, the server sets the conversation status to `EXISTING` and includes the known conversation key encrypted by the server as the `encrypted_conversation_key` field in the conversation metadata. Alice caches the

conversation key and encrypts and sends her first message using the conversation key generated by the server.

When Bob attempts to retrieve the conversation key to decrypt Alice's message, the server similarly answers that the conversation is an existing conversation and provides the known conversation key, encrypted using Bob's public key.

Recommendations

Short term, enforce the use of signatures for all conversation keys, and validate all signatures before trusting the conversation keys. See TOB-TWDM-4 in the design review report.

Long term, consider using an authenticated key exchange protocol like X3DH to negotiate a unique conversation key for each pair of devices. Such protocols provide a number of benefits. First, each pair of devices negotiates a unique conversation key that is bound to the long-term identity keys of the two devices. This means that devices would no longer need to sign encrypted messages, as the negotiated conversation key would implicitly authenticate the two parties. It would also allow the encrypted DMs protocol to avoid having to rotate the long-term conversation key if a device is lost or stolen. The remaining devices would simply delete the conversation key for the lost device when the device is deregistered, which would also ensure that any future messages sent to the conversation cannot be decrypted using the compromised conversation key.

2. Long-term identity keys without signatures are not rejected

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-XCHAT-2
Target: `subsystem/dm/core/src/commonMain/kotlin/com/x/dms/PublicKeyRepo.kt`	

Description

Long-term keys without signatures are not validated before being stored in cache, which is treated as a trusted source of key information. This means that the server can inject an invalid or corrupted public key into the client's cache simply by leaving off the signature.

```
private suspend fun validateAndStoreKeysInCache(
        keys: GraphqlPublicKeys,
        userId: UserIdentifier,
    ): PublicKeysForUser? {
        val version = keys.version
        val identityPubKeyBytes = keys.identity.decodeAsBase64()
        val identity = identityPubKeyBytes?.let {
keyFactory.reconstructPublicKeyFromBytes(it) }
            ?: return null
        val signingKeyBytes = keys.signing?.decodeAsBase64()
        val signing = signingKeyBytes?.let {
keyFactory.reconstructPublicKeyFromBytes(it) }
        val identityKeySignature = keys.identityKeySignature
        val identityKeySignatureBytes = identityKeySignature?.decodeAsBase64()
        if (signing != null && identityKeySignatureBytes != null &&
identityKeySignatureBytes.isNotEmpty()) {
            val isSignatureValid =
                signatureValidator.isSignatureValid(signing,
identityKeySignatureBytes, identityPubKeyBytes)
            if (!isSignatureValid) {
                XLog.d(TAG) { "got invalid signature, rejecting keypair $version for
${userId.userId}" }
                return null
            }
        } else {
            // TODO (eventually) reject if signature is missing
        }
[...]
```

Figure 2.1: Invalid signature handling does not result in rejection

Exploit Scenario

The server can generate its own user public keys for the given user, without signature, to a requesting party. Because there is no signature, these keys will be accepted, enabling person-in-the-middle attacks by the server.

Recommendations

Short term, reject unsigned keys.

3. Use of truncated SHA-1

Severity: **Undetermined**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-XCHAT-3
Target: `common/impl/src/main/kotlin/com/x/common/impl/ClientIdentityImpl.kt`	

Description

The `getGuestIdFromUuid` function uses SHA1 to map UUID values to `GuestID` values. This is an issue for two reasons: first, the use of SHA-1 is discouraged due to its susceptibility to chosen-prefix attacks. Truncating the hash exacerbates this problem. Second, the 53-bit GuestID value is simply too small: given the size of the product's user base and the assumption that SHA-1 ensures that that `GuestID` values are uniform, it is all but guaranteed that several guests will have the same `GuestID`, leading to protocol problems.

```
   private fun getGuestIdFromUuid(clientUuid: String): Long {
        // Use the SHA digest truncated to 53 bits - the value must be below Javascript's
maximum number.
        // We start at 13 to use the last 56 bits of the 160 bit digest (it's 20 bytes
long).

        // 20 byte SHA returned in String format (40 characters)
        val sha1 = BaseDataUtils.sha1Digest(clientUuid)!!
        val sha1Truncated = sha1.substring(sha1.length - LENGTH_DIGEST_UUID / BITS_PER_BYTE
* 2)

        // 2**53 is the largest integer that Javascript can represent.
        // Mask guest IDs by 2**53-1.
        return sha1Truncated.toLong(16) and JAVASCRIPT_MAX_PRECISION_MASK
    }
```

Figure 3.1: SHA-1 generation of `GuestID` values

Exploit Scenario

Two users with the same `GuestID` are invited to an encrypted chat session. When adding participant information to a hash map, the information for the first user (such as public key information) is overwritten by information for the second user, leaving participants unable to send messages to the first user with the shared `GuestId`.

Recommendations

Short term, switch to a better format for `GuestID` values. The `Long` data type is simply insufficient for the size of the product's userbase. We recommend at least a 128-bit value.

Long term, phase out the use of SHA-1 in any context.

4. Conversation keys are not checked for server replay	
Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-XCHAT-4
Target: `subsystem/dm/core/src/commonMain/kotlin/com/x/dms/ConversationManager.kt`	

Description

This issue tracks with finding TOB-TWDM-10 in the DM design review. As stated in that finding:

> *If a device is stolen or lost, the protocol provides functionality to reset the conversation key. The client requests a reset of the current conversation key from the server, which causes the server to delete the encrypted conversation keys for the conversation. The next time a device attempts to retrieve the encrypted conversation key to send a new message, the server will respond with conversation data with the conversation status field set to `UNINITIATED`, which will cause the device to generate a fresh conversation key for the conversation.*
>
> *However, if clients do not track conversation keys, there is no way for participating devices to know that the encrypted conversation keys were actually deleted by the server. The server could simply ignore the request and return the same conversation key the next time a participating device requests it.*

When retrieving encrypted conversation keys from the server, the `MessageSendHandler` code does not verify that the retrieved key is not a previously reset key. The key rotation code *does* ensure that the server has stored a new conversation key by ensuring that the server responds with the new encrypted key immediately after the reset, but this does not prevent the server from providing the old key in response to future requests.

Exploit Scenario

Alice and Bob have an ongoing encrypted conversation. Alice suspects that the server has obtained the conversation key and can read their encrypted messages, so she decides to reset the conversation key. Alice requests a reset from the server, and the server responds correctly to her next request. The next time Bob retrieves the conversation metadata to send a message to Alice, the server responds with the conversation status `EXISTING` and returns the old encrypted conversation key to Bob. Bob reuses the old and potentially compromised key to encrypt his message to Alice.

Recommendations

We make the same recommendations as in the design review report.

Short term, have the device include a new set of encrypted conversation keys whenever it requests a reset of the conversation key from the server. To ensure that all devices have updated to the new conversation key whenever a key is reset, include the conversation key in the safety number computed by each device, and alert users to ensure that they know that they need to reverify the safety number for the conversation.

Long term, continuously update the conversation key using either the double ratchet mechanism used by Signal or ratchet trees used by MLS, both of which provide post-compromise security. This will ensure that future messages cannot be decrypted even if a conversation key is compromised.

5. Group key inconsistencies can lead to Invisible Salamander attacks	
Severity: **Medium**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-XCHAT-5
Targets: `subsystem/dm/keymanagement/src/javaMain/kotlin/com/x/dms/JvmKeyFactory.kt`, `subsystem/dm/core/src/commonMain/kotlin/com/x/dms/ConversationManager.kt`	

Description

An attacker can send a single valid ciphertext to a group conversation and cause different participants to decrypt it to a different plaintext value. In the literature, this is referred to as "Invisible Salamanders," due to the original proof of concept that accompanied its discovery.

The variant of the attack relevant to XChat is made possible by two design decisions:

1. The conversation key is generated by the sender, and then independently wrapped with each recipient's public key, using ephemeral-static elliptic curve Diffie-Hellman. As a consequence, an attacker with a modified client can simply generate two conversation keys, then encrypt a different key with each recipient's public key.

2. Messages and attachments are encrypted with authenticated encryption modes that do not provide key-commitment. Neither AES-GCM, Xsalsa20-Poly1305, nor XChaCha20-Poly1305 are key-committing cipher modes.

Exploit Scenario

Alice initiates a group conversation with Bob and Carol. Alice sends a Conversation Key Update that both Bob and Carol will interpret as different keys (key_1, key_2) for the subsequent message.

Alice then uses public cryptographic exploit techniques to craft a ciphertext that will decrypt to one of two valid plaintext (using polyglot techniques to make the garbage sections invisible), depending on which key was used. She then attaches this file to the message.

Finally, she issues another Conversation Key Update with a fresh key to both parties to ensure the conversation continues normally afterwards.

After receiving the message, Bob and Carol will see two different attachments in their DM history, with no indication of inconsistency. The most straightforward attack is to dupe abuse reporting, as outlined in the original Invisible Salamanders paper. A more interesting example is manipulating cryptocurrency trades by sending “buy” to some users and “sell” to others in an encrypted group conversation.

Recommendations

Short term, adopting a ratchet tree approach to group key agreement will prevent the technique needed to set up the exploit, since the client can no longer create a split-brain scenario where different sets of participants have a different conversation key.

Long term, consider adding an additional, domain-separated Key Derivation Function output from the ratcheting step to commit to the key used for each message.

References

- Key Committing AEADs
- Invisible Salamanders Are Not What You Think
- Key commitment - Concepts - AWS Encryption SDK

6. Confused deputy attack in avatar/attachment encryption

Severity: **Informational**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-XCHAT-6
Target: `subsystem/dm/keymanagement/src/commonMain/kotlin/com/x/dms/LibsodiumEncryption.kt`	

Description

Attachments and avatars are encrypted using libsodium's `secretstream` API, which uses the STREAM construction with XChaCha20-Poly1305 under the hood.

One limitation of this API is that it provides only authenticated encryption, not AEAD. This introduces the risk of a confused deputy attack, where two files that were encrypted under the same key can be swapped, and they will still decrypt successfully.

Since multiple categories of attachments use the same underlying encryption function, with the same key, it is possible to replay valid ciphertexts in the wrong context and get them to decrypt.

For example, an attacker with filesystem access could use the lack of context-binding to swap the small avatar for a group conversation with a large photo, potentially slowing the app down for the user.

Recommendations

Short term, verify the size and MIME type of the decrypted avatar and attachments to help mitigate any performance degradation caused by the missing AAD parameter in the libsodium API.

Long term, in conjunction with the ratchet key deployment, introduce an additional key derivation step before attachment encryption that includes some context (protocol version, attachment type, UUID for message ID, etc.) in its calculation.

In addition to using the KDF to derive a unique encryption key for the attachment, the KDF should also output a domain-separated commitment hash to include in the file header. Verify that this matches before attempting to decrypt the file. This is sufficient to ensure that swapping files will trigger a decryption failure and therefore be rejected.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Public Analyses

After the engagement began, independent cryptographers began publishing their own analyses of the XChat system. We have worked to limit our findings to those not already publicly identified, but this appendix documents some of the public results.

Juicebox

Nora Trapp of the Juicebox published a blog post entitled "Don't Put All Your Juice in One Box." She notes that the Juicebox protocol, if implemented correctly, spreads trust across "independent trust boundaries":

> *It's tempting to think that splitting trust across different services you own and operate is good enough. After all, they might live on different machines, use different credentials, or even run different code. But in reality, real attackers don't care about your service boundaries. They care about organizational ones.*
>
> *If a single subpoena, exploit, or admin mistake compromises all your realms at once, they're not independent realms, they're aliases. From the attacker's perspective, it's one big juicy target.*

If XChat self-hosts all its Juicebox servers, this would fall squarely into the scenario described above.

Matthew Garrett

Matthew Garrett published an analysis of the XChat system. Garrett notes XChat's lack of forward secrecy, stating that "it's a long way from the state of the art for a messaging client." He also points out that the four-digit PIN restriction, along with the Argon2 parameters, make it simple for an attacker with backend access to extract private keys "at will." Finally, he points out that XChat does not allow for any form of out-of-band verification of the Juicebox public keys, meaning that servers can perform person-in-the-middle attacks with ease.

C. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not a comprehensive analysis of the system.

From October 20 to October 22, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the X team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the six issues described in this report, X has resolved three issues and has not resolved two issues; the status of the remaining issue is undetermined. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Encrypted conversation keys are not validated	High	Resolved
2	Long-term identity keys without signatures are not rejected	High	Resolved
3	Use of truncated SHA-1	Undetermined	Unresolved
4	Conversation keys are not checked for server replay	High	Undetermined
5	Group key inconsistencies can lead to invisible salamander attacks	Medium	Resolved
6	Confused deputy attack in avatar/attachment encryption	Informational	Unresolved

Detailed Fix Review Results

TOB-XCHAT-1: Encrypted conversation keys are not validated
Resolved in PR #15476. X has updated their software to always drop events with invalid signatures. Unless the server obtains the user's signing key, this mitigates the server's ability to push new keys as `EXISTING`, and therefore from setting up the attack.

Additionally, X has adopted a Safety Number approach to mitigating key substitution attacks; however, its implementation does not actually offer any security.

The order of the two input public keys is determined by which user has the lower user ID. Then, the algorithm is as follows:

```
   suspend fun generateSafetyNumbers(pubKeyA: XChatPublicKey, pubKeyB: XChatPublicKey):
List<Long> {
        val aBytes = pubKeyA.encoded()
        val bBytes = pubKeyB.encoded()
        if (aBytes.isEmpty() || bBytes.isEmpty()) return emptyList()
        val bytes = aBytes + bBytes
        val groupCount = 12
        val result = ArrayList<Long>(groupCount)
        // Use 2-byte windows across the combined bytes to produce 12 groups.
        for (i in 0 until groupCount) {
            val b1 = bytes[(2 * i) % bytes.size].toInt() and 0xFF
            val b2 = bytes[(2 * i + 1) % bytes.size].toInt() and 0xFF
            val combined = ((b1 shl 8) or b2)
            val mixed = (combined + (131 * i)) % 100000 // range 0..99999
            result.add(mixed.toLong())
        }
        return result
    }
```

Figure C.1: Safety number generation algorithm

The length of an encoded public key is 32 bytes, so the `bytes` array has a length of 64. The `for` loop will access values between 0 and 23 (the maximum value is 2 * 11 + 1), inclusive.

Consequently, the only bytes that actually affect the Safety Number calculation are from the public key corresponding to the user with the smallest user ID.

Additionally, the combination and mixing algorithms do not uniformly fill the desired range between 0 and 99999. Assuming b1 and b2 have a value of `0xFF`, and `i` is at its maximum value (11), the value on the left side of the `% 100000` operation will only be 66976.

To consider prior art, the algorithm Signal uses for calculating safety numbers involves iterating SHA-512 over both public keys before extracting numbers that can be displayed to the end user.

TOB-XCHAT-2: Long-term identity keys without signatures are not rejected
Resolved in PR #15476. X has updated their software to always drop events with invalid signatures.

TOB-XCHAT-3: Use of truncated SHA-1
Unresolved as of commit 22d5538. The affected code is still present as of the time of this fix review.

TOB-XCHAT-4: Conversation keys are not checked for server replay
Undetermined. The patch for this issue was not included in the scope of the fix review; however, upon reviewing the patches for previous issues, we observed the presence of a new `RatchetTreeCoordinator` class.

Ratchet trees are an effective method for mitigating post-compromise security risks. However, we did not audit the ratchet tree implementation, so we consider this issue's status to be Undetermined.

TOB-XCHAT-5: Group key inconsistencies can lead to invisible salamander attacks
Resolved. This issue has been mitigated by the adoption of ratchet trees.

TOB-XCHAT-6: Confused Deputy Attack in Avatar / Attachment Encryption
Undetermined as of commit 22d5538. The affected code is still present as of the time of this fix review.

D. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up to date with our latest news and announcements, please follow @trailofbits on X or LinkedIn, and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.