BSV Blockchain TS-SDK

Security Assessment

January 14, 2026

Prepared for:
Darren Kellenschwiler
BSV Association

Prepared by: **Opal Wright, Marc Ilunga, Robin Sandblom, and Paul Bottinelli**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Tara Goodwin-Ruffus, Project Manager
tara.goodwin-ruffus@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:

Opal Wright, Consultant
opal.wright@trailofbits.com

Marc Ilunga, Consultant
marc.ilunga@trailofbits.com

Paul Bottinelli, Consultant
paul.bottinelli@trailofbits.com

Robin Sandblom, Consultant
robin.sandblom@trailofbits.com

Project Timeline

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

Date	Event
July 18, 2025	Pre-project kickoff call
July 25, 2025	Status update meeting #1
August 1, 2025	Status update meeting #2
August 13, 2025	Report readout meeting
September 12, 2025	Delivery of final comprehensive report
January 14, 2026	Completion of fix review

Executive Summary

Engagement Overview

BSV Association engaged Trail of Bits to review the security of the cryptographic primitive and wallet implementations in its TypeScript SDK (TS-SDK). This code is used to secure cryptocurrency transactions in multiple contexts and is intended as a stand-alone, zero-dependency cryptography library. The library also implements several functionalities needed for wallets on the BSV blockchain.

A team of four consultants conducted the review from July 21 to August 8, 2025, for a total of six engineer-weeks of effort. Our review and testing efforts focused on the primitives implemented in the SDK and their use within the wallet implementation, which is also included in the SDK. With full access to source code and documentation, we performed static and dynamic testing of the TS-SDK, using automated and manual processes.

Observations and Impact

Overall, the library suffers from several flaws and is generally not implemented robustly. We found several issues related to the correctness of the algorithms, and we found that timing side channels are pervasive throughout the codebase. We also noted some inappropriate algorithm choices and silent behaviors that could lead to security problems in the future. Given the time-boxed nature of this engagement and the large number of findings reported on, we did not achieve a comprehensive review of all the files in scope.

We found that several primitive implementations do not follow the specification (e.g., TOB-BSV-9, TOB-BSV-12, TOB-BSV-15). For example, the AES-GCM implementation is noncompliant for large inputs (TOB-BSV-9), which may lead to the inability to decrypt data in certain circumstances.

Moreover, several issues highlight general poor input validation, which could enable signature forgery (TOB-BSV-21) and has resulted in non-robust implementations of certain functionalities, such as type conversions (TOB-BSV-20, TOB-BSV-25, TOB-BSV-26) and validation of inputs to cryptographic primitives (TOB-BSV-10, TOB-BSV-13).

We also found several instances of missing countermeasures against timing vulnerabilities, (e.g., TOB-BSV-3, TOB-BSV-4, TOB-BSV-5, TOB-BSV-8). These issues can severely undermine the safe use of the library for sensitive operations. For example, we found a high-severity issue stemming from secret comparisons that are not done in constant time, which could leak secret information to attackers (TOB-BSV-8).

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that BSV Association 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 adopting external dependencies for cryptographic primitives.** We recommend that the TS-SDK undergo significant changes to replace current cryptographic primitive implementations with more battle-tested code available in the open-source ecosystem. This could take the form of integrating external code into the codebase or adding external dependencies to the TS-SDK. We provide additional guidance in appendix B.
- **Improve testing infrastructure.** Develop extensive tests for all cryptographic primitives, with a strong eye toward "unhappy path" execution: handling of invalid parameters, mismatched parameters, invalid points, incorrect ciphertext lengths, missing tags and initialization vectors (IVs), and so on. Automated fuzzing tools and test vector suites like Wycheproof can help find edge cases that are improperly handled.
- **Consider conducting a subsequent review of the codebase after remediating the issues reported during this review.** Doing so will ensure that the issues we reported and their high-level causes have been systematically addressed.

Finding Severities and Categories

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

EXPOSURE ANALYSIS

Severity	*Count*
High	6
Medium	5
Low	3
Informational	13
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Cryptography	23
Data Validation	4

Project Goals

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

- Are the cryptographic primitives appropriate for the intended use?
- Are the cryptographic primitives implemented correctly?
- Are the cryptographic primitives implemented securely?
- Are the cryptographic primitives used appropriately by the wallet modules?

Project Targets

The engagement involved reviewing and testing the following target.

ts-sdk

Repository	https://github.com/bsv-blockchain/ts-sdk/
Version	`7109a74c14847164fa02430b7314231dab4d2527`
Type	TypeScript

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 the cryptographic primitives implemented in the `primitives` folder
- Manual review of the use of primitives in the wallet implementation under the `wallet` folder
- Running of the AES-GCM implementation against the Wycheproof test vector suite
- Comparison of the cryptographic primitive implementations to published standards, including the following:
    - AES
    - ECDSA
    - HKDF
    - HMAC-DRBG

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:

- The Bignum implementation and some reduction procedures were given only a cursory review due to their size and relatively lower complexity.
- The parsing and decoding methods were not fully reviewed, primarily due to the large number of formats supported.
- Our review of the wallet covered only the use of cryptographic primitives.
- The remaining components of the library were not in the scope of the review.

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in-house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Semgrep	An open-source static analysis tool for finding bugs and enforcing code standards when editing or committing code and during build time	Appendix D

Areas of Focus

Our automated testing and verification work focused on the following:

- Assessing the correctness of the implementation
- Identifying common misuse patterns

We identified no findings through our automated testing efforts.

Summary of Findings

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

ID	Title	Type	Severity
1	deriveSymmetricKey does not use a key derivation function	Cryptography	Informational
2	toKeyShares can result in key leakage or unrecoverable keys	Cryptography	Informational
3	Large-integer arithmetic is susceptible to timing attacks	Cryptography	Informational
4	Elliptic curve operations are susceptible to timing attacks	Cryptography	Informational
5	AES implementations are susceptible to cache-timing attacks	Cryptography	Informational
6	GCM computations are susceptible to cache-timing attacks	Cryptography	Informational
7	HMAC-DRBG is not forward-secure	Cryptography	Informational
8	Secret comparisons are not constant time	Cryptography	High
9	AES-GCM implementation is noncompliant for large inputs	Cryptography	Medium
10	decrypt does not validate the length of ciphertexts	Data Validation	Informational
11	Several issues with the message encryption protocol	Cryptography	High
12	Spurious zero-block padding is not compliant with AES-GCM standard	Cryptography	Medium

13	AES-GCM implementation does not reject empty IV	Cryptography	**Informational**
14	Silent zero-padding of AES key is insecure	Cryptography	**Medium**
15	SHA-512 padding is noncompliant and could lead to collisions	Cryptography	**Medium**
16	DRBG seed concatenation leads to colliding outputs	Cryptography	**Informational**
17	Lenient Jacobian point constructor allows subtle attacks	Cryptography	**High**
18	Encoding the point at infinity triggers an assertion error	Cryptography	**Low**
19	The htonl function is not conditional	Cryptography	**Informational**
20	Hex string conversion is fragile	Cryptography	**High**
21	Big integer representation of messages allows signature forgery	Cryptography	**High**
22	ECDSA nonce range check is incorrect	Cryptography	**Informational**
23	Point decoding fails to ensure point is on the curve	Cryptography	**High**
24	Point addition resulting in infinity renders scalar multiplication incorrect	Cryptography	**Medium**
25	Base64 decoding is not robust	Data Validation	**Low**
26	Missing bounds checks in UTF-8 encoding	Data Validation	**Low**
27	Missing parameter checks in Chaum-Pedersen proofs and ECDSA signatures	Data Validation	**Informational**

Detailed Findings

1. deriveSymmetricKey does not use a key derivation function	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BSV-1
Target: `src/wallet/KeyDeriver.ts`	

Description

The `KeyDeriver.deriveSymmetricKey` method derives a shared point via an elliptic curve Diffie–Hellman (ECDH) computation, and then returns the x-coordinate of the point directly as a 256-bit symmetric key.

This approach does not adhere to best practices. Elliptic curve points include significant structure. While attacks involving this structure do not currently exist, it may be exploitable in the context of a symmetric encryption key, and most cryptographic protocols avoid using them directly. Rather, they use a key derivation function (KDF) to extract a uniform random value to be used as a key.

```
return new SymmetricKey(
   derivedPrivateKey.deriveSharedSecret(derivedPublicKey)?.x?.toArray() ?? []
 )
```

Figure 1.1: Final value returned by `deriveSymmetricKey` (`src/wallet/KeyDeriver.ts#L200–L202`)

Recommendations

Short term, switch to using a KDF to derive symmetric keys. NIST SP 800-56C describes several KDF techniques, including a variant of HKDF from RFC 5869.

Long term, ensure that clear delineations are drawn between symmetric keys and values derived from asymmetric operations.

2. toKeyShares can result in key leakage or unrecoverable keys	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BSV-2
Target: `src/primitives/PrivateKey.ts`	

Description
The `PrivateKey.toKeyShares` method does not check that randomly generated x-coordinates are unique or that none of the x-coordinates are equal to zero.

If an x-coordinate is equal to zero, the corresponding y-coordinate would be equal to the secret being shared, meaning that the secret itself could be distributed to a party.

If two or more shares have the same x-coordinate, the number of distinct shares may drop below the reconstruction threshold, making secret recovery impossible.

The severity of this issue has been marked as informational since the likelihood of either event (a random x-coordinate being zero or two random x-coordinates being equal to each other) is negligible.

Exploit Scenario
A bug in the system-provided RNG causes it to return a repeating pattern, or all zeros. This leads to repeated x-coordinates or to zeroed x-coordinates. When the resulting shares are distributed, the key is either leaked or unrecoverable.

Recommendations
Short term, add checks to ensure that x-coordinates are unique and nonzero.

Long term, consider mitigation techniques for the zero-share issue, including transforming the x-coordinates via a nonzero function (e.g., `x = new BigNum("2", "hex").pow(x)` or similar), and using a counter to ensure that x-coordinates are unique.

3. Large-integer arithmetic is susceptible to timing attacks	
Severity: **Informational**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-3
Target: `src/primitives/BigNumber.ts`, `src/primitives/ReductionContext.ts`	

Description

Several methods in the `BigNumber` and `ReductionContext` classes do not run in constant time, which could enable timing side-channel attacks. The following are some of the affected methods:

- `ReductionContext.pow()`: The conditional statement in the sliding-window exponentiation leaks information about the number of leading zeros in the exponent.
- `BigNumber._invmp()`: It uses the extended Euclidean algorithm (EEA) to compute multiplicative inverses, leaking information about the size of the integer with respect to the modulus.
- `ReductionContext.add()`: The conditional subtraction leaks whether the sum exceeds the modulus.

Additionally, the `BigNumber` library relies on the `bigint` type, which does not provide any constant-time guarantees, according to Mozilla:

> *JavaScript BigInts therefore could be dangerous for use in cryptography without mitigating factors.*

In most cases, non-cryptographic large integer libraries do not support constant-time multiplication, division, remaindering, addition, or subtraction, so the list of functions above is not exhaustive.

Timing side channels could be used to recover secret values used in cryptographic operations. In this case, the attack is made more difficult by the fact that JavaScript is a just-in-time compiled, garbage collected language. We have marked the severity of this issue as informational since it is difficult to completely mitigate side-channel leakages in languages like JavaScript, and it is not clear if side-channel attacks fall within the threat model of the library.

Exploit Scenario
An attacker is able to observe the timing of operations with known and unknown inputs and uses the timing differences to recover the unknown (secret) inputs.

Recommendations
Short term, move away from use of the `bigint` type and replace variable-time methods with constant-time equivalents (such as using exponentiation for inversion instead of the EEA).

Long term, reconsider the project's zero-dependency goal. Instead, consider integrating a large-integer library that supports constant-time operations.

4. Elliptic curve operations are susceptible to timing attacks	
Severity: **Informational**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-4
Target: `src/primitives/Point.ts`, `src/primitives/JacobianPoint.ts`	

Description

Several methods in the `Point` and `JacobianPoint` classes run in variable time. Analysis of timing information could allow attackers to glean secret key material, particularly in the case of the `mul` method.

The `add`, `dbl`, and `mul` methods in both classes make conditional branches and perform very different numbers of operations depending on those branches. Because the `mul` operation is at the heart of ECDH, this is a particularly high-risk area for side channels.

Timing side channels could be used to recover secret values used in cryptographic operations. In this case, the attack is made more difficult by the fact that JavaScript is a just-in-time compiled, garbage collected language. We have marked the severity of this issue as informational since it is difficult to completely mitigate side-channel leakages in languages like JavaScript, and it is not clear if side-channel attacks fall within the threat model of the library.

Exploit Scenario

An attacker is able to observe the timing of scalar multiplications with a known public input and uses the timing information to recover a user's private key.

Recommendations

Short term, look into the feasibility of using constant-time scalar multiplication algorithms like Montgomery ladders, as well as complete formulae that do not require branching.

Long term, reconsider the project's zero-dependency goal. Instead, consider integrating a widely tested elliptic curve library such as noble-curves.

5. AES implementations are susceptible to cache-timing attacks	
Severity: **Informational**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-5
Target: `src/primitives/AESGCM.ts`, `src/compat/ECIES.ts`	

Description
There are two implementations of the AES block cipher in the codebase. The first, in the `AESGCM` module, uses a hard-coded lookup table for the S-box. The second, in the `ECIES` module, computes a lookup table for the S-box at initialization.

AES implementations with table lookups are subject to cache-timing attacks that can leak the symmetric key.

Timing side channels like this could be used by an attacker to recover secret values used in cryptographic operations, such as the AES symmetric key. In this case, the attack is made more difficult by the fact that JavaScript is a just-in-time compiled, garbage collected language. We have marked the severity of this issue as informational since it is difficult to completely mitigate side-channel leakages in languages like JavaScript, and it is not clear if side-channel attacks fall within the threat model of the library.

Exploit Scenario
An attacker coerces a client to decrypt multiple ciphertexts under the same key while closely monitoring the timing. Using that information, the attacker recovers the AES key.

Recommendations
Short term, eliminate conditional branches and data-dependent lookups in the AES codebases.

Long term, reconsider the project's zero-dependency goal. Instead, consider using an external library like noble-ciphers. Additionally, consider moving to an ARX cipher like ChaCha20, which involves no data-dependent table lookups or branches.

6. GCM computations are susceptible to cache-timing attacks

Severity: **Informational**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-6
Target: `src/primitives/AESGCM.ts`	

Description

The finite field arithmetic used for the GCM computations, in particular the multiplication in $GF(2^{128})$, includes conditional branches, which lead to variable-time operation, leaking information about the authentication key.

```
export const multiply = function (block0: number[], block1: number[]): number[] {
  const v = block1.slice()
  const z = createZeroBlock(16)

  for (let i = 0; i < 16; i++) {
    for (let j = 7; j >= 0; j--) {
      if ((block0[i] & (1 << j)) !== 0) {
        xorInto(z, v)
      }

      if ((v[15] & 1) !== 0) {
        rightShift(v)
        xorInto(v, R)
      } else {
        rightShift(v)
      }
    }
  }

  return z
}
```

Figure 6.1: GCM multiplication, with conditional branches highlighted (`src/primitives/AESGCM.ts#L248–L268`)

Timing side channels like this could be used by an attacker to recover secret values used in cryptographic operations, such as the AES-GCM authentication key. In this case, the attack is made more difficult by the fact that JavaScript is a just-in-time compiled, garbage collected language. We have marked the severity of this issue as informational since it is difficult to completely mitigate side-channel leakages in languages like JavaScript, and it is not clear if side-channel attacks fall within the threat model of the library.

Exploit Scenario
An attacker monitors the timing of encryption and authentication tag computation, learning information about the authentication key.

Recommendations
Short term, consider replacing the affected arithmetic with other methods of arithmetic in $GF(2^{128})$, such as the method used in BearSSL.

Long term, reconsider the project's zero-dependency goal. Instead, consider using an external library like noble-ciphers.

7. HMAC-DRBG is not forward-secure	
Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BSV-7
Target: `src/primitives/DRBG.ts`	

Description

The HMAC-DRBG implementation used in ECDSA nonce generation is not forward-secure unless additional input is supplied with each call to the `generate` function. Details are available in this paper on the NIST SP 800-90A standard (see page 13).

The HMAC-DRBG implementation is used only for deterministic ECDSA nonce generation, with a single call to `generate`, so this does not currently present an issue. The selection of HMAC-DRBG keeps the ECDSA implementation compliant with appendix A.3.3 of FIPS 186-5.

However, the DRBG module is an exposed API, and if it were used in other contexts, the lack of forward-secrecy could pose a threat.

Recommendations

Short term, consider making the HMAC-DRBG implementation private or adding a disclaimer at the top of the `DRBG.ts` file to indicate potential pitfalls.

Long term, ensure APIs that do not need to be exposed are not exposed, or that security concerns specific to exposed APIs are clearly documented.

8. Secret comparisons are not constant time

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-8
Target: `src/totp/totp.ts`, `src/wallet/ProtoWallet.ts`	

Description

Secret values, including HMAC-derived values, are compared using the === operator, which does not run in constant time. This can leak information about secret values.

```
const valid =
   Hash.sha256hmac(key.toArray(), args.data).toString() ===
   args.hmac.toString()
```

Figure 8.1: HMAC comparison (src/wallet/ProtoWallet.ts#L226–L228)

```
for (const c of counters) {
   if (passcode === generateHOTP(secret, c, _options)) {
     return true
   }
 }
```

Figure 8.2: HOTP comparison (src/totp/totp.ts#L70–L73)

Exploit Scenario

An attacker is able to submit multiple passcodes to the TOTP validation system. By varying the first byte of the passcode and carefully monitoring the time between passcode submission and rejection, the attacker is able to learn when the first byte of the passcode is correct. The attacker repeats this process to learn subsequent bytes until the entire passcode is recovered and the attacker gains access.

Recommendations

Short term, implement constant-time comparison and use it where appropriate.

Long term, ensure that operations reliant on secret data are identified, documented, and carefully handled in constant time.

9. AES-GCM implementation is noncompliant for large inputs

Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-9
Target: `src/primitives/AESGCM.ts`	

Description

The `getBytes` function in the AES-GCM implementation encodes a numerical value over 4 bytes, in big-endian byte order.

```
export const getBytes = function (numericValue: number): number[] {
  return [
    (numericValue & 0xFF000000) >>> 24,
    (numericValue & 0x00FF0000) >> 16,
    (numericValue & 0x0000FF00) >> 8,
    numericValue & 0x000000FF
  ]
}
```

Figure 9.1: The `getBytes` function (src/primitives/AESGCM.ts#L200–L207)

This function is used to encode the length of different fields during the encryption and decryption processes. For example, the length of the additional authenticated data and of the ciphertext are obtained using this function in the computation of the GHASH input (figure 10.2).

```
plainTag = plainTag.concat(createZeroBlock(4))
   .concat(getBytes(additionalAuthenticatedData.length * 8))
   .concat(createZeroBlock(4)).concat(getBytes(cipherText.length * 8))
```

Figure 9.2: GHASH input computation in the AESGCM function (src/primitives/AESGCM.ts#L373–L375)

The governing NIST standard, NIST SP800-38D, states that these lengths have to be encoded over 8 bytes (see, for example, page 15, step 5 of the standard). The implementation correctly encodes the lengths over 8 bytes, but it does so by truncating the lengths of the relevant fields to 4 bytes and padding them with 4 zero bytes (see the `createZeroBlock(4)` function calls in figure 10.2).

As a result, if a ciphertext or additional authenticated data were ever larger than 2^{32}, the implementation would incorrectly encode their lengths, which does not follow the standard and would lead to incompatibility issues with other implementations.

Exploit Scenario

An uninformed user cannot decrypt large messages that were encrypted using a different, standard-compliant AES-GCM library. Vice versa, they cannot decrypt large ciphertexts produced by the BSV TS-SDK using other libraries.

Recommendations

Short term, update the `getBytes` function to encode lengths over 8 bytes. If encoding over 4 bytes is also necessary in the implementation, either provide two separate encoding functions or modify the function such that a second parameter dictates the number of bytes over which to encode the value. In either case, add a check ensuring that numerical values larger than the largest possible encoding (e.g., $2^{32} - 1$ for 4 bytes) triggers an error.

Long term, add more comprehensive tests for the AES-GCM implementation targeting edge cases like the one described in this finding.

10. decrypt does not validate the length of ciphertexts

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-BSV-10
Target: `src/primitives/SymmetricKey.ts`, `src/primitives/AESGCM.ts`	

Description

The `decrypt` function is used to decrypt AES-GCM ciphertexts. The function expects as input an array of numbers that should encode the concatenation of the IV, the ciphertext core, and the MAC tag. However, `decrypt` uses array slicing to parse the input without checking the length. As a consequence, the core decryption function could be run on invalid inputs.

After parsing the input, decrypt calls `AESGCMDecrypt` on the ciphertext, IV, tag, and (empty) additional data. The array slicing method used to extract the inputs to `AESGCMDecrypt`, shown in figure 11.1, does not guarantee the length of the input.

For example, an input such as `new Array(33).fill(0)` is too short and should be rejected during parsing. However, thanks to slicing, the input will be parsed as an IV, an empty ciphertext with a tag of length one. Ultimately, ciphertext validation fails for such malformed inputs, thanks to the strict comparison in `AESGCMDecrypt`.

```
decrypt (msg: number[] | string, enc?: 'hex' | 'utf8'): string | number[] {
  msg = toArray(msg, enc)
  const iv = msg.slice(0, 32)
  const ciphertextWithTag = msg.slice(32)
  const messageTag = ciphertextWithTag.slice(-16)
  const ciphertext = ciphertextWithTag.slice(0, -16)
  const result = AESGCMDecrypt(
    ciphertext,
    [],
    iv,
    messageTag,
    this.toArray()
  )
  if (result === null) {
    throw new Error('Decryption failed!')
  }
  return encode(result, enc)
}
```

Figure 10.1: Input parsing through array slicing without length validation (`src/primitives/SymmetricKey.ts#L71–L88`)

```
// Generate the authentication tag
const calculatedTag = gctr(ghash(compareTag, hashSubKey), preCounterBlock, key)

// If the calculated tag does not match the provided tag, return null - the
decryption failed.
if (calculatedTag.join() !== authenticationTag.join()) {
  return null
}
```

Figure 10.2: Final GCM tag verification (`src/primitives/AESGCM.ts#L432–L438`)

Recommendations

Short term, enforce that the inputs to `decrypt` have at least 48 bytes.

Long term, ensure that all external inputs are validated before use.

11. Several issues with the message encryption protocol

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-11
Target: `src/messages/EncryptedMessage.ts`	

Description

The encrypted message protocol uses a nonstandard key exchange protocol that does not provide the commonly expected guarantees of authenticated key exchange protocols. In particular, the protocol suffers from missing replay protection, a lack of forward-secrecy, and vulnerabilities to identity-misbinding attacks.

The key exchange protocol derives sessions from parties' long-term keys and a random, per-session invoice number. For example, if Alice and Bob hold the long-term secrets—a, b, respectively—then Alice can initiate a session by picking a random invoice iv and deriving the shared secret session key $S = (a + H(abG, iv))(b + H(abG, iv))G$. In the preceding expression of S, G is the generator and H is the HMAC function.

```
export const encrypt = (
  message: number[],
  sender: PrivateKey,
  recipient: PublicKey
): number[] => {
  const keyID = Random(32)
  const keyIDBase64 = toBase64(keyID)
  const invoiceNumber = `2-message encryption-${keyIDBase64}`
  const signingPriv = sender.deriveChild(recipient, invoiceNumber)
  const recipientPub = recipient.deriveChild(sender, invoiceNumber)
  const sharedSecret = signingPriv.deriveSharedSecret(recipientPub)
  const symmetricKey = new SymmetricKey(sharedSecret.encode(true).slice(1))
  const encrypted = symmetricKey.encrypt(message) as number[]
  const senderPublicKey = sender.toPublicKey().encode(true)
  const version = toArray(VERSION, 'hex')
  return [
    ...version,
    ...senderPublicKey,
    ...recipient.encode(true),
    ...keyID,
    ...encrypted
  ]
```

Figure 11.1: `sharedSecret` is an ECDH output over derived child keys. (`src/messages/EncryptedMessage.ts#L9–L38`)

Figure 12.2 shows the derivation of the child private keys, which follows a similar process.

```
deriveChild (
  publicKey: PublicKey,
  invoiceNumber: string,
  cacheSharedSecret?: ((priv: PrivateKey, pub: Point, point: Point) => void),
  retrieveCachedSharedSecret?: ((priv: PrivateKey, pub: Point) => (Point |
undefined))
): PrivateKey {
  let sharedSecret: Point
  if (typeof retrieveCachedSharedSecret === 'function') {
    const retrieved = retrieveCachedSharedSecret(this, publicKey)
    if (typeof retrieved !== 'undefined') {
      sharedSecret = retrieved
    } else {
      sharedSecret = this.deriveSharedSecret(publicKey)
      if (typeof cacheSharedSecret === 'function') {
        cacheSharedSecret(this, publicKey, sharedSecret)
      }
    }
  } else {
    sharedSecret = this.deriveSharedSecret(publicKey)
  }
  const invoiceNumberBin = toArray(invoiceNumber, 'utf8')
  const hmac = sha256hmac(sharedSecret.encode(true), invoiceNumberBin)
  const curve = new Curve()
  return new PrivateKey(this.add(new BigNumber(hmac)).mod(curve.n).toArray())
}
```

Figure 11.2: Child key derivation (`src/primitives/PrivateKey.ts#L358–L390`)

The protocol suffers from the following shortcomings:

- **Missing forward-secrecy:** Although the per-session invoice is used to create ephemeral ECDH shares, they are deterministically derived from the long-term key. As a consequence, a leak of the long-term key leads to a leak of all session keys.
- **Identity misbinding:** If there is no other mechanism to ensure unique identities, and public keys are the only user identifier, then the protocol suffers from a subtle authentication flaw that violates explicit authentication. A malicious user Eve could present Bob's public key to Alice as her own public key, even if she does not know the corresponding secret key. Alice may encrypt messages to Eve, who then forwards them to Bob. In this case, Alice believes she is talking to Eve, though she is actually talking to Bob. This type of attack is commonly known as an identity-misbinding attack.
- **Missing replay protection:** The session key is a deterministic function of the long-term secret and the invoice. Since the invoice is chosen by the initiating party and communicated in the clear, any recorded communication can be replayed to the receiver.

Exploit Scenario

Eve gains access to a sender's private key and is able to decrypt all past messages encrypted with that key since the protocol lacks forward-secrecy. She is also able to replay messages. Later, Eve records a conversation from Alice to Bob in which Alice asks Bob to transfer funds to Eve. Eve replays the messages to induce Bob to transfer more funds. Finally, Eve exploits the weak identity binding to cause Alice to send the message she did not intend to send to Bob.

Recommendations

Short term, warn users about the risk of using the encrypted message protocol for sensitive operations. Note that the "key exchange with authentication" variant suggested in the documentation still fails to provide adequate authentication guarantees.

Long term, consider implementing a strong authenticated key exchange protocol such as X3DH, Noise, or SIGMA.

12. Spurious zero-block padding is not compliant with AES-GCM standard

Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-12
Target: `src/primitives/AESGCM.ts`	

Description

When supplied with an empty plaintext, ciphertext, or additional authenticated data (AAD), the AES-GCM encryption or decryption function appends a block of zero bytes to the GHASH input. This can be seen in the following code excerpt, where, if the resulting ciphertext or the AAD is empty, a zero block is appended to the `plainTag` variable.

```
if (additionalAuthenticatedData.length === 0) {
  plainTag = plainTag.concat(createZeroBlock(16))
} else if (additionalAuthenticatedData.length % 16 !== 0) {
  plainTag = plainTag.concat(createZeroBlock(16 -
(additionalAuthenticatedData.length % 16)))
}

plainTag = plainTag.concat(cipherText)

if (cipherText.length === 0) {
  plainTag = plainTag.concat(createZeroBlock(16))
} else if (cipherText.length % 16 !== 0) {
  plainTag = plainTag.concat(createZeroBlock(16 - (cipherText.length % 16)))
}
```

Figure 12.1: AESGCM function with superfluous block (src/primitives/AESGCM.ts#L359–L371)

This operation is not compliant with the AES-GCM standard, NIST SP800-38D, which states that the additional authenticated data A and the ciphertext C are each padded with "the minimum number of zero bits, possibly none, so that the bit lengths of the resulting strings are multiples of the block size." Thus, when the ciphertext or the AAD length is zero, the required padding length is zero and no padding block should be added.

This superfluous computation is also performed twice within the corresponding decryption function, in addition to the two instances shown above.

Exploit Scenario

An uninformed user cannot use the BSV TS-SDK to interoperate with other libraries when encrypting or decrypting particular messages.

Recommendations

Short term, remove the conditional statements appending the superfluous zero-block. For example, the code excerpt in figure 12.1 could be rewritten as follows.

```
if (additionalAuthenticatedData.length % 16 !== 0) {
  plainTag = plainTag.concat(createZeroBlock(16 -
(additionalAuthenticatedData.length % 16)))
}

plainTag = plainTag.concat(cipherText)

if (cipherText.length % 16 !== 0) {
  plainTag = plainTag.concat(createZeroBlock(16 - (cipherText.length % 16)))
}
```

Figure 12.2: Fixed AESGCM function

Long term, add comprehensive tests that exercise edge cases such as the one described in this finding.

13. AES-GCM implementation does not reject empty IV

Severity: **Informational**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-13
Target: `src/primitives/AESGCM.ts`	

Description

The implementations of the AES-GCM encryption and decryption functions fail to reject an empty IV. This is not compliant with the governing NIST standard, NIST SP800-38D, which, as stated in section 5.2.1.1, requires the IV's bit length to satisfy $1 \leq len(IV) \leq 2^{64} - 1$, as shown in figure 13.1.

> The bit lengths of the input strings to the authenticated encryption function shall meet the following requirements:
>
> - $\text{len}(P) \leq 2^{39}\text{-}256$;
> - $\text{len}(A) \leq 2^{64}\text{-}1$;
> - $1 \leq \text{len}(IV) \leq 2^{64}\text{-}1$.

Figure 13.1: Excerpt of NIST SP800-38D, section 5.2.1.1

Recommendations

Short term, modify the `AESGCM` and `AESGCMDecrypt` functions to return an error when an empty IV is supplied.

Long term, add comprehensive tests that exercise edge cases such as the one described in this finding.

14. Silent zero-padding of AES key is insecure

Severity: **Medium**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-14
Target: `src/primitives/AESGCM.ts`	

Description

The implementation of the AES block cipher silently pads short keys with zeros, as shown in the code snippet in figure 14.1.

```
// Since the BigNumber representation of keys ignores big endian zeroes,
// extend incoming key arrays with zeros to the smallest standard key size.
const ekey = Array.from(key)
if (ekey.length <= 16) {
  while (ekey.length < 16) ekey.unshift(0)
  roundLimit = 11
} else if (ekey.length <= 24) {
  while (ekey.length < 24) ekey.unshift(0)
  roundLimit = 13
} else if (key.length <= 32) {
  while (ekey.length < 32) ekey.unshift(0)
  roundLimit = 15
} else {
  throw new Error('Illegal key length: ' + String(key.length))
}
```

Figure 14.1: AES function padding the key (`src/primitives/AESGCM.ts#L146–L160`)

This allows users of the library to encrypt data using short keys with low amounts of entropy, giving them a false sense of security.

Exploit Scenario

An uninformed user can encrypt and decrypt data using keys that provide limited security, without their knowledge. Additionally, such users cannot use the BSV TS-SDK to interoperate with other libraries when encrypting or decrypting messages with keys of nonstandard lengths.

Recommendations

Short term, modify the AES function to return an error when a key of invalid length is supplied.

15. SHA-512 padding is noncompliant and could lead to collisions

Severity: **Medium**	Difficulty: **Undetermined**
Type: Cryptography	Finding ID: TOB-BSV-15
Target: `src/primitives/Hash.ts`	

Description

The SHA-512 implementation does not follow the NIST specification for padding. According to the specification, the length field should be encoded over 128 bits. However, the implementation uses a common padding for SHA-256 and SHA-512, where the length is encoded over 64 bits.

```
 private _pad (): number[] {
    //
    let len = this.pendingTotal
[...]
    // Append length
    len <<= 3
[...]
    if (this.endian === 'big') {
      [...]
      res[i++] = 0
      res[i++] = 0
      res[i++] = 0
      res[i++] = 0
      res[i++] = (len >>> 24) & 0xff
      res[i++] = (len >>> 16) & 0xff
      res[i++] = (len >>> 8) & 0xff
      res[i++] = len & 0xff
    } else {
[...]
      res[i++] = 0
      res[i++] = 0
      res[i++] = 0
      res[i++] = 0
[...]
  }
}
```

Figure 15.1: `src/primitives/Hash.ts#L169–L214`

Furthermore, the bit size length calculation uses a logical shift by 3. However, logical bit shifts handle only 32-bit values and may overflow for larger values. As a consequence, two different messages may result in a collision in this implementation. In particular, certain

carefully chosen messages whose length encoding has the same value in the lower 64 bits will collide. Such messages would not collide in a compliant implementation.

Recommendations

Short term, implement the padding scheme specified by NIST for SHA-512.

Long term, ensure that all primitives are tested against invalid inputs. Use robust arithmetic operations.

16. DRBG seed concatenation leads to colliding outputs

Severity: **Informational**	Difficulty: **N/A**
Type: Cryptography	Finding ID: TOB-BSV-16
Target: `src/primitives/DRBG.ts`	

Description

The DRBG is based on the deterministic EDCSA design that uses HMAC as a PRF to generate signing nonces. As a stand-alone DRBG, the DRBG seed is the concatenation of input entropy and a nonce. Due to simply using concatenation, several (entropy, nonce) pairs lead to the same seed. This is not an issue for ECDSA since entropy and nonces are of fixed size, but stand-alone uses of the DRBG should be considerate of the risk of seed collisions.

```
constructor (entropy: number[] | string, nonce: number[] | string) {
  entropy = toArray(entropy, 'hex')
  nonce = toArray(nonce, 'hex')

  if (entropy.length < 32) {
    throw new Error('Not enough entropy. Minimum is 256 bits')
  }
  const seed = entropy.concat(nonce)

  this.K = new Array(32)
  this.V = new Array(32)
  for (let i = 0; i < 32; i++) {
    this.K[i] = 0x00
    this.V[i] = 0x01
  }
  this.update(seed)
}
```

Figure 16.1: ts-sdk/src/primitives/DRBG.ts#L21–L37

Recommendations

Short term, implement a more defensive encoding of the DRBG seed, which can be accomplished by appending the length of the input entropy and the nonce, both over a fixed size. Note that the encoding we recommend is not compliant with RFC 6979. This fact is not an issue for general uses of ECDSA; however, the lack of compatibility may cause issues in specific use cases.

Long term, reconsider the project's zero-dependency goal. Instead, consider using a separate battle-tested DRBG construction that has received appropriate security analyses.

17. Lenient Jacobian point constructor allows subtle attacks

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-17
Target: `src/primitives/JacobianPoint.ts`	

Description

In the implementation under review, elliptic curve points can be represented in either affine or Jacobian coordinates. In Jacobian representation, a point corresponds to a triplet of coordinates, (X, Y, Z), and the point at infinity is defined to be any triplet where the Z-coordinate is equal to 0, as shown in the code excerpt below.

```
/**
 * Checks whether the JacobianPoint instance represents a point at infinity.
 * @method isInfinity
 * @returns Returns true if the JacobianPoint's z-coordinate equals to zero (which
represents the point at infinity in Jacobian coordinates). Returns false otherwise.
 *
 * @example
 * const point = new JacobianPoint('5', '6', '0');
 * console.log(point.isInfinity()); // Output: true
 */
isInfinity (): boolean {
  return this.z.cmpn(0) === 0
}
```

Figure 17.1: Infinity testing function for `JacobianPoint` (`src/primitives/JacobianPoint.ts#L446–L457`)

The constructor for a Jacobian point is fairly lenient; it does not check that the coordinates correspond to a point on the curve, which could lead to subtle arithmetic issues and enable attacks. Additionally, the constructor essentially allows arbitrary representations for the point at infinity. Figure 17.2 showcases a test where several representations of the point at infinity can be created.

```
test('should detect Point at Infinity', () => {
  const inf1 = new JPoint(
    new BigNumber(0),
    new BigNumber(0),
    new BigNumber(0)
  )
  expect(inf1.isInfinity()).toBe(true)
```

```
const inf2 = new JPoint('0', '0', '0');
expect(inf2.isInfinity()).toBe(true)

const inf3 = new JPoint(null, null, null);
expect(inf3.isInfinity()).toBe(true)
```

Figure 17.2: Unit test showcasing the creation of multiple infinity representations

An interesting side-effect is that representations of the point at infinity where the X- and Y-coordinates are equal to zero are deemed equal to all other points. Figure 17.3 showcases the continuation of the above test, where a "good" point is deemed equal to both the `inf1` and `inf2` variables.

```
const good = new JPoint(
  new BigNumber('e7789226...', 16),
  new BigNumber('4b76b191...', 16),
  new BigNumber('cbf8d990...', 16)
)
expect(inf1.eq(good)).toBe(true)
expect(inf2.eq(good)).toBe(true)
expect(inf3.eq(good)).toBe(false)
```

Figure 17.3: Continuation of the unit test showcasing the incorrect equality comparison

This issue is attributable to the implementation of the equality function for Jacobian points (i.e., the `eq` function in `src/primitives/JacobianPoint.ts#L355`), which does not check that either point is the point at infinity.

Exploit Scenario

An attacker provides a Jacobian representation of the point at infinity in a sensitive computation, such as for an elliptic curve public key recovery operation from a signature, which results in signature forgery.

Recommendations

Short term, update the `eq` function to identify whether either point is the point at infinity, to return true if they both are equal to the point at infinity, and to return false if only one of the inputs is the point at infinity. Additionally, consider updating the point constructor to perform an optional curve check, and ensuring the point at infinity has only one possible representation.

Long term, ensure all elliptic curve points supplied by an untrusted party are checked to be on the curve.

18. Encoding the point at infinity triggers an assertion error

Severity: **Low**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-18
Target: `src/primitives/Point.ts`	

Description

In affine coordinates, the point at infinity can be created by supplying `null` for the X- and Y-coordinates of a point. When the code tries to encode this point, it throws a failed assertion error, as shown in the excerpt in figure 18.1.

```
const good = new Point(null, null)
good.encode()

> fromRed works only with numbers in reduction context

>  351 |   }
>  352 |
> 353 |   private assert (val: unknown, msg: string = 'Assertion failed'): void { if
(!(val as boolean)) throw new Error(msg) }
```

Figure 18.1: Example of the error triggered when encoding the point at infinity

Exploit Scenario

An attacker triggers a denial of service for a target user by providing a point at infinity that the target user attempts to encode.

Recommendations

Short term, update the `encode` function to handle the point at infinity gracefully.

Long term, add unit tests covering all possible edge cases, such as the one discussed in this finding.

19. The htonl function is not conditional

Severity: **Informational**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-19
Target: `src/primitives/Hash.ts`	

Description

The `htonl` function, excerpted in figure 19.1 for reference, takes a 32-bit integer in the machine's native byte order and rearranges its bytes into network order (i.e., big-endian).

```
function htonl (w: number): number {
  const res =
    (w >>> 24) |
    ((w >>> 8) & 0xff00) |
    ((w << 8) & 0xff0000) |
    ((w & 0xff) << 24)
  return res >>> 0
}
```

Figure 19.1: Snippet of the `htonl` function (`src/primitives/Hash.ts#L287–L294`)

This implementation could be confusing since, in C, the standard `htonl` function is conditional: on a big-endian machine, it returns the argument unchanged. In contrast, the implementation above swaps the byte order unconditionally, regardless of the underlying hardware.

Recommendations

Short term, rename the function in order to properly describe its behavior. For example, `swapBytes` would be a good alternative.

Long term, consider adding a check of the host's endianness and implementing a conditional byte swap.

20. Hex string conversion is fragile

Severity: **High**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-20
Target: `src/primitives/Hash.ts`	

Description

The implementation of the `toArray` function silently discards non-hexadecimal characters when converting a hexadecimal string to a byte array, as highlighted in the code snippet in figure 20.1.

```
msg = msg.replace(/[^a-z0-9]+/gi, '')
if (msg.length % 2 !== 0) {
  msg = '0' + msg
}
for (let i = 0; i < msg.length; i += 2) {
  res.push(parseInt(msg[i] + msg[i + 1], 16))
}
```

Figure 20.1: A portion of the `toArray` function (`src/primitives/Hash.ts#L270–L276`)

This behavior is fragile and may lead to unexpected issues.

Note that the `hexToArray` function in `src/primitives/utils.ts` performs a similar conversion and should also be updated according to the recommendation below.

Exploit Scenario

An attacker forges a message signature by appending non-hexadecimal characters to an existing signed message.

Recommendations

Short term, have the `toArray` and `hexToArray` functions return an error when encountering non-hexadecimal characters.

Long term, review the codebase and delete duplicate functions.

21. Big integer representation of messages allows signature forgery

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-21
Target: `src/primitives/ECDSA.ts`	

Description

The ECDSA signature verification and generation performed in the `ECDSA.ts` file uses big integers to represent the message to be verified or signed, respectively, as shown in the function signature in figure 21.1, for reference.

```
/**
 * Verifies a digital signature of a given message.
 *
 * Message and key used during the signature generation process, and the previously
computed signature
 * are used to validate the authenticity of the digital signature.
 *
 * @function verify
 * @param msg - The BigNumber message for which the signature has to be verified.
 * @param sig - Signature object consisting of parameters 'r' and 's'.
 * @param key - Public key in Point.
 * @returns Returns true if the signature is valid and false otherwise.
 *
 * @example
 * const msg = new BigNumber('2664878', 16)
 * const key = new Point(new BigNumber(10), new BigNumber(20)
 * const signature = sign(msg, new BigNumber('123456'))
 * const isVerified = verify(msg, sig, key)
 */
export const verify = (msg: BigNumber, sig: Signature, key: Point): boolean => {
```

Figure 21.1: Signature of the ECDSA verification function (`src/primitives/ECDSA.ts#L383–L401`)

It seems that the function is intended to be called with a pre-hashed message (since no hashing is performed within the function itself), but no bounds checks are performed on this parameter; a message of arbitrary length can be verified or signed by the implementation.

Crucially, the governing standard (SEC 1: Elliptic Curve Cryptography, Version 2.0) defines the verification operation with a message parameter being an octet string and performs the hash of the message within the verification function, as shown in the excerpt of the standard in figure 21.2.

4.1.4 Verifying Operation

Entity V must verify signed messages from entity U using ECDSA using the keys and parameters established during the setup procedure and the key deployment procedure as follows:

Input: The verifying operation takes as input:

1. An octet string M which is the message.
2. Entity U's purported signature $S = (r, s)$ on M.
3. Optional: extra information to recover R efficiently from r (see below).

Output: An indication of whether the purported signature on M is valid or not — either "valid" or "invalid".

Actions: Verify the purported signature S on M as follows:

1. If r and s are not both integers in the interval $[1, n-1]$, output "invalid" and stop.
2. Use the hash function established during the setup procedure to compute the hash value:

$$H = Hash(M)$$

of length *hashlen* octets as specified in Section 3.5. If the hash function outputs "invalid", output "invalid" and stop.

Figure 21.2: ECDSA verification procedure from the SEC 1 standard

In practice, the verification function simply reduces the input message (as a big integer) modulo the curve order during some intermediary computation. This allows arbitrary message forgery, as is demonstrated in the Exploit Scenario section below. Given a valid (message, signature) pair, the signature is also valid for modified messages to which multiples of the curve order were added.

Exploit Scenario

Given a valid (message, signature) pair, an attacker forges a message signature by adding the curve order to the message. The test in figure 21.3 (which can be pasted to the `src/primitives/__tests/ECDSA.test.ts` file) demonstrates this signature forgery.

```
it('should sign and verify big number', () => {
  const msg = new BigNumber('deadbeef', 16)
  const key = new BigNumber(
    '1e5edd45de6d22deebef4596b80444ffcc29143839c1dce18db470e25b4be7b5', 16
  )
  const signature = ECDSA.sign(msg, key)
  expect(ECDSA.verify(msg, signature, pub)).toBeTruthy()
  const new_msg = msg.add(new
BigNumber('FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141', 16))
```

```
  expect(ECDSA.verify(new_msg, signature, pub)).toBeTruthy()
})
```

Figure 21.3: Unit test demonstrating a signature forgery

Recommendations

Short term, add comprehensive input validation of all parameters in the sign and verify functions (e.g., by following the governing standard) and add validation to ensure the message encoding is not larger than the agreed-upon hash function output.

Long term, consider moving away from using big integers to represent parameters and update the code to align more closely with the governing standards.

22. ECDSA nonce range check is incorrect

Severity: **Informational**	Difficulty: **High**
Type: Cryptography	Finding ID: TOB-BSV-22
Target: `src/primitives/ECDSA.ts`	

Description

The ECDSA signature generation function operates by generating a fresh nonce value k, in the range $1 \leq k \leq n - 1$. The implementation incorrectly rejects the values $k = 1$ and $k = n - 1$, as shown in the excerpt in figure 22.1.

```
if (k <= one || k >= ns1) {
  if (customK instanceof BigNumber) {
    throw new Error(
      'Invalid fixed custom K value (must be more than 1 and less than N-1)'
    )
  } else {
    continue
  }
}
```

Figure 22.1: Incorrect bounds check in ECDSA sign function (src/primitives/ECDSA.ts#L243–L251)

While the probability that these values would be sampled is negligible, the range check should accept the values $k = 1$ and $k = n - 1$ to strictly follow the specification.

Note that a similar (incorrect) bounds check is performed a few lines below, at ECDSA.ts#L328.

Recommendations

Short term, modify the range check shown in figure 22.1 to make it adhere to the specification.

23. Point decoding fails to ensure point is on the curve

Severity: **High**	Difficulty: **Low**
Type: Cryptography	Finding ID: TOB-BSV-23
Target: `src/primitives/Point.ts`	

Description

In elliptic curve cryptography, a number of subtle attack vectors stem from the failure to ensure that computations are performed on points that lie on the correct curve. The implementation defines a few methods for instantiating points from encoded formats, such as the `fromDER` function, excerpted in figure 23.1. However, this function does not ensure the point is on the correct curve when decoding a point that is in uncompressed format.

```
static fromDER (bytes: number[]): Point {
  const len = 32
  // uncompressed, hybrid-odd, hybrid-even
  if (
    (bytes[0] === 0x04 || bytes[0] === 0x06 || bytes[0] === 0x07) &&
    bytes.length - 1 === 2 * len
  ) {
    if (bytes[0] === 0x06) {
      if (bytes[bytes.length - 1] % 2 !== 0) {
        throw new Error('Point string value is wrong length')
      }
    } else if (bytes[0] === 0x07) {
      if (bytes[bytes.length - 1] % 2 !== 1) {
        throw new Error('Point string value is wrong length')
      }
    }

    const res = new Point(
      bytes.slice(1, 1 + len),
      bytes.slice(1 + len, 1 + 2 * len)
    )
```

Figure 23.1: Snippet of the point decoding function `fromDER` (`src/primitives/Point.ts#L45–L65`)

Exploit Scenario

An attacker submits a public key point that lies on a different curve during an ECDH key exchange; since the point is not checked to be on the same curve, the attacker learns some information about the target user's private key, and eventually the victim leaks their key entirely.

Recommendations

Short term, modify all point decoding functions such that they ensure that the points are valid. Specifically, they should ensure the following:

- The point coordinates are lower than the field modulus.
- The point satisfies the curve equation.
- The point is not the point at infinity.

Long term, review the codebase to ensure *all* functions accepting data from potentially untrusted parties perform strict input validation.

24. Point addition resulting in infinity renders scalar multiplication incorrect

Severity: **Medium**	Difficulty: **Medium**
Type: Cryptography	Finding ID: TOB-BSV-24
Target: `src/primitives/ECDSA.ts`	

Description

The `pointAdd` function defined in `ECDSA.ts` returns `null` when the sum of its operands amount to the point at infinity, as highlighted in the code excerpt in figure 24.1.

```
function pointAdd (
  P: { x: bigint, y: bigint } | null,
  Q: { x: bigint, y: bigint } | null
): { x: bigint, y: bigint } | null {
  if (P === null) return Q
  if (Q === null) return P

  if (P.x === Q.x && P.y === mod(-Q.y, p)) {
    return null // Point at infinity
  }
```

Figure 24.1: Excerpt of the point addition function (`src/primitives/ECDSA.ts#L177–L186`)

The point addition function is called repeatedly when scalar multiplication is performed, via the `scalarMul` function. That function incorrectly handles the point at infinity in the two areas highlighted in figure 24.2; it effectively ignores the addition instead of setting the variable to the point at infinity. This can cause the scalar multiplication function to return an incorrect result in a rare case (i.e., when the underlying point addition function is called with a point and its inverse).

```
while (k > BigInt(0)) {
  if (k % BigInt(2) === BigInt(1)) {
    Q = Q === null ? N : (pointAdd(Q, N) ?? Q)
  }
  N = pointAdd(N, N) ?? N
  k >>= BigInt(1)
}
```

Figure 24.2: Excerpt of the scalar multiplication function (`src/primitives/ECDSA.ts#L216–L222`)

Exploit Scenario

An attacker creates a carefully crafted input that triggers the incorrect computation above, resulting in a leak of sensitive data.

Recommendations

Short term, update the `scalarMul` function such that it correctly propagates the point at infinity.

Long term, write extensive unit tests triggering edge cases like the one described in this finding.

25. Base64 decoding is not robust

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BSV-25
Target: `src/primitives/utils.ts`	

Description

The `base64ToArray` function does not implement robust Base64 parsing. The function fails to properly validate input, accepts malformed Base64 strings, and does not handle invalid characters correctly. It strips trailing padding characters with `/=+$/` indiscriminately and uses `indexOf()` without checking the return value, which could be nonpositive.

```
const base64ToArray = (msg: string): number[] => {
  const base64Chars =
    'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'
  const result: number[] = []
  let currentBit = 0
  let currentByte = 0

  for (const char of msg.replace(/=+$/, '')) {
    currentBit = (currentBit << 6) | base64Chars.indexOf(char)
    currentByte += 6

    if (currentByte >= 8) {
      currentByte -= 8
      result.push((currentBit >> currentByte) & 0xff)
      currentBit &= (1 << currentByte) - 1
    }
  }

  return result
}
```

Figure 25.1: Indiscriminate removal of padding without input validation (`src/primitives/utils.ts#L66–L85`)

Exploit Scenario

An authorization relies on an allowlist to reject payloads with certain Base64 inputs. The attacker bypasses a denylist by presenting an invalid Base64 input that is ultimately decoded to the underlying values of one of the entries in the denylist.

Recommendations

Short term, add validation to check that the input is a proper Base64-encoded input and reject any malformed input.

Long term, ensure all inputs are thoroughly validated, and test the implementation against invalid inputs.

26. Missing bounds checks in UTF-8 encoding

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-BSV-26
Target: `src/primitives/utils.ts`	

Description

The UTF-8 decoder fails to validate array bounds before accessing subsequent bytes in multi-byte sequences, which could lead to buffer overreads. When processing 2-byte, 3-byte, and 4-byte UTF-8 sequences, the code directly accesses `arr[i + n]` without verifying that these indices exist within the array's bounds:

```
  // 2-byte sequence (110xxxxx 10xxxxxx)
  const byte2 = arr[i + 1]
  skip = 1
  const codePoint = ((byte & 0x1f) << 6) | (byte2 & 0x3f)
  result += String.fromCharCode(codePoint)
} else if (byte >= 0xe0 && byte <= 0xef) {
  // 3-byte sequence (1110xxxx 10xxxxxx 10xxxxxx)
  const byte2 = arr[i + 1]
  const byte3 = arr[i + 2]
  skip = 2
  const codePoint =
    ((byte & 0x0f) << 12) | ((byte2 & 0x3f) << 6) | (byte3 & 0x3f)
  result += String.fromCharCode(codePoint)
} else if (byte >= 0xf0 && byte <= 0xf7) {
  // 4-byte sequence (11110xxx 10xxxxxx 10xxxxxx 10xxxxxx)
  const byte2 = arr[i + 1]
  const byte3 = arr[i + 2]
  const byte4 = arr[i + 3]
  skip = 3
  const codePoint =
    ((byte & 0x07) << 18) |
    ((byte2 & 0x3f) << 12) |
    ((byte3 & 0x3f) << 6) |
    (byte4 & 0x3f)

  // Convert to UTF-16 surrogate pair
  const surrogate1 = 0xd800 + ((codePoint - 0x10000) >> 10)
  const surrogate2 = 0xdc00 + ((codePoint - 0x10000) & 0x3ff)
  result += String.fromCharCode(surrogate1, surrogate2)
```

Figure 26.1: Array access without bounds checks (`src/primitives/utils.ts#L167–L195`)

Recommendations

Short term, add bounds checking to the code before it accesses subsequent bytes in multi-byte sequences.

Long term, ensure all array accesses are checked.

27. Missing parameter checks in Chaum-Pedersen proofs and ECDSA signatures

Severity: **Informational**	Difficulty: **Undetermined**
Type: Data Validation	Finding ID: TOB-BSV-27
Target: `src/primitives/Schnorr.ts`, `src/primitives/ECDSA.ts`	

Description

The Schnorr proof validator and ECDSA validator both accept invalid public keys. In particular, both validators will accept the point at infinity as a public key.

Per FIPS 186-5, ECDSA verifiers "should" ensure that public keys are validated according to appendix D.1 of NIST SP 800-186, which requires that the point at infinity be rejected. However, the `verifyECDSA` method does not perform this check.

There is no formal standard for Schnorr proofs. Failures to validate inputs have led to zero-knowledge discrete logarithm proof failures in the past, and public keys like the point at infinity or a zero scalar can result in unexpected behavior. In the case of BSV's Chaum-Pedersen proof, which demonstrates that (A, B, S) is a Diffie-Hellman triple, there does not appear to be a straightforward exploitation.

Recommendations

Short term, add checks to reject public keys or Schnorr proofs that integrate the point at infinity. In the case of Schnorr proofs, also reject any proofs where z = 0.

Long term, ensure that all inputs are validated according to relevant standards and best practices. Tools like Wycheproof can help identify unhandled edge cases and missing checks.

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. Libraries to Consider for Integration

BSV has set a goal of zero dependencies for the TS-SDK. In order to allow BSV to avoid having to reinvent the wheel while also enjoying some of the security advantages of existing software, we have found several libraries that are technically suitable for wrapping or incorporating into the BSV codebase and that are available under permissive licenses.

Symmetric Encryption

- Paul Miller's noble-ciphers
- Paul Miller's noble-hashes
- Node.js crypto
- For browsers, webcrypto

Elliptic Curve Signatures and ECDSA

- Paul Miller's noble-curves
- Node.js crypto

Encrypted Messaging

- Chainsafe libp2p-noise

Advantages and Disadvantages

All of the libraries listed above can provide at least some subset of the functionality implemented in the TS-SDK library, but there are some tradeoffs in each case.

The noble libraries are permissively licensed, and the code can be copied directly into BSV's codebase if desired (as long as credit is given). They are also written to be what the author calls "tree-shakeable": if support for a specific cipher, hash function, or elliptic curve is not needed, the supporting code can be removed easily. However, the noble libraries are written in pure TypeScript, so the anti-side-channel measures taken do not have the strong constant-time properties offered by native libraries.

The webcrypto library has the advantage of being included by default in every major web browser, and the encryption primitives typically run as native code that is likely to have been written with timing attacks, correctness, and several common "footguns" in mind. The disadvantages come in the form of limited APIs (for instance, streaming interfaces for hashes are not supported) and the direct tie to web browsers. For server-side computation, the Node.js crypto library provides similar functionality.

It is worth considering the tradeoffs of bringing in code from external libraries versus simply declaring those libraries as dependencies. As time goes on, the code copied into the

BSV codebase can become outdated, missing security-critical fixes. Specifying the noble libraries as dependencies will make it easier to ensure up-to-date code is being used.

C. Code Quality Findings

This section of the report lists issues that do not correspond to any security problems, but whose resolution will help improve the security and reliability of the codebase.

Duplicated code. The codebase contains several instances of duplicated code. Code duplication makes maintainability more difficult, increases the attack surface, and is generally not in line with best practices. For example, the curve parameters representing the elliptic curve secp256k1 are defined *at least four times* in the codebase, twice in the `ECDSA.ts` file (in the `sign` function and in the `verify` function), once in `Curve.ts`, and once in `Point.ts`.

Iteration variable incremented twice. In the `sign` function in `ECDSA.ts`, an iteration counter is incremented twice, as shown in the snippet in figure C.1. Outside of potential incompatibilities, this does not seem to pose any security risk.

```
while (!validSignature) {
  iter += 1
  validSignature = true
  iter += 1
```

Figure C.1: `src/primitives/ECDSA.ts#L232–L235`

Insecure hash function. The `src/primitives/Hash.ts` file contains an implementation of the SHA-1 hash function. This function is no longer fitting as a general-purpose hash function, as it does not provide collision-resistance. In 2011, NIST deprecated SHA-1 for new digital signatures (NIST SP 800-131A).

Extraneous format conversions. The codebase handles several different representations for the data it handles, and does so very generously. For example, the curve point constructor excerpted below accepts inputs that are of type `BigNumber`, `number`, `number[]`, `string`, and `null`.

```
constructor (
  x: BigNumber | number | number[] | string | null,
  y: BigNumber | number | number[] | string | null,
  isRed: boolean = true
) {
  super('affine')
```

Figure C.2: `src/primitives/Point.ts#L281–L286`

The example above is illustrative, but the codebase generally works on many different types of inputs. This considerably raises the code complexity and introduces potential vulnerabilities, as demonstrated by several findings within this report.

Lack of input validation when the `fromString` function creates a polynomial. Given a purported Base58-encoded input string, `fromString` calls `split(".")` to extract the x and y components. The caller of `fromString` is expected to validate the input, although this fact is not indicated.

```
static fromString (str: string): PointInFiniteField {
  const [x, y] = str.split('.')
  return new PointInFiniteField(
    new BigNumber(fromBase58(x)),
    new BigNumber(fromBase58(y))
  )
}
```

Figure C.3: Missing input validation (src/primitives/Polynomial.ts#L21–L27)

D. Static Analysis Tools

We performed static analysis on the TS-SDK source code using Semgrep to identify low-complexity weaknesses. We used several rulesets, shown in figure D.1. No findings were identified.

```
semgrep --metrics=off --sarif --config="p/insecure-transport”
semgrep --metrics=off --sarif --config="p/trailofbits”
```

Figure D.1: Commands used to run Semgrep

E. 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 comprehensive analysis of the system.

From January 12 to January 14, 2026, Trail of Bits reviewed the fixes and mitigations implemented by the BSV team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

We reviewed a number of commits across a resolution project.

In summary, of the 27 findings described in this report, BSV has fully resolved 19 findings, has partially resolved four, and has not resolved the remaining four. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	deriveSymmetricKey does not use a key derivation function	Informational	Unresolved
2	toKeyShares can result in key leakage or unrecoverable keys	Informational	Resolved
3	Large-integer arithmetic is susceptible to timing attacks	Informational	Partially Resolved
4	Elliptic curve operations are susceptible to timing attacks	Informational	Partially Resolved
5	AES implementations are susceptible to cache-timing attacks	Informational	Unresolved
6	GCM computations are susceptible to cache-timing attacks	Informational	Partially Resolved
7	HMAC-DRBG is not forward-secure	Informational	Resolved
8	Secret comparisons are not constant time	High	Resolved

9	AES-GCM implementation is noncompliant for large inputs	Medium	Resolved
10	decrypt does not validate the length of ciphertexts	Informational	Resolved
11	Several issues with the message encryption protocol	High	Partially Resolved
12	Spurious zero-block padding is not compliant with AES-GCM standard	Medium	Unresolved
13	AES-GCM implementation does not reject empty IV	Informational	Resolved
14	Silent zero-padding of AES key is insecure	Medium	Resolved
15	SHA-512 padding is noncompliant and could lead to collisions	Medium	Resolved
16	DRBG seed concatenation leads to colliding outputs	Informational	Resolved
17	Lenient Jacobian point constructor allows subtle attacks	High	Resolved
18	Encoding the point at infinity triggers an assertion error	Low	Resolved
19	The htonl function is not conditional	Informational	Resolved
20	Hex string conversion is fragile	High	Resolved
21	Big integer representation of messages allows signature forgery	High	Resolved
22	ECDSA nonce range check is incorrect	Informational	Resolved
23	Point decoding fails to ensure point is on the curve	High	Resolved

24	Point addition resulting in infinity renders scalar multiplication incorrect	Medium	Resolved
25	Base64 decoding is not robust	Low	Resolved
26	Missing bounds checks in UTF-8 encoding	Low	Resolved
27	Missing parameter checks in Chaum-Pedersen proofs and ECDSA signatures	Informational	Unresolved

Detailed Fix Review Results

TOB-BSV-1: deriveSymmetricKey does not use a key derivation function

Unresolved. BSV declined to make changes to key derivation. The following information was provided in the relevant bug report.

> *If you take a uniformly random private key k and compute $P = kG$ on an elliptic curve over a finite field $\mathbb{F}_p$, the resulting x-coordinate is almost uniformly distributed over $\mathbb{F}_p$, but not perfectly.*
>
> *A few reasons why:*
>
> > *• The mapping $k \to x(kG)$ isn't one-to-one: each valid x (except a few edge cases) corresponds to two possible points $\pm P$ with the same x but opposite y.*
> >
> > *• The group order n of the curve usually differs slightly from p, so when you multiply kG for all $k \in [1, n-1]$, you don't hit every x value in the field—just a nearly uniform subset.*
> >
> > *• The deviation from uniformity is negligible for cryptographic curves like secp256k1. Statistically, the bias is on the order of $1/p$, which is astronomically small ($\approx 2^{-256}$).*
>
> *So for all practical purposes, the x-coordinate of a random point on the curve behaves as if it were a random 256-bit number, but strictly speaking, it's only approximately uniform within the field.*

It is worth noting that the set of 256-bit keys that can be generated via the $k \to x(kG)$ mapping generates a set of cardinality closer to 2^{255} than 2^{256}, as any x value where $x^3 + 7$ is not a quadratic residue modulo the curve modulus cannot be a valid key. However, as a rule of thumb, the expected cost of computing an elliptic curve discrete logarithm on

secp256k1 is estimated to be about 2^{128} curve operations, so a small reduction in the security of a 256-bit key does not impact the overall security of the system.

TOB-BSV-2: toKeyShares can result in key leakage or unrecoverable keys
Resolved in commit `baaf8a5`. The x values for share generation are now checked to ensure they are unique and not zero.

TOB-BSV-3: Large-integer arithmetic is susceptible to timing attacks
Partially resolved in commit `3b59295`. The `BigNumber._invmp()` function was switched to use Fermat's little theorem for inversion. However, the `add()` and `pow()` functions remain variable-time.

The documentation for the `invmp()` function provides the following context:

```
SECURITY NOTE:
This implementation avoids variable-time extended Euclidean
algorithms to reduce timing side-channel leakage. However,
JavaScript BigInt arithmetic does not provide constant-time
guarantees. This implementation is suitable for browser and
single-tenant environments but is not hardened against
high-resolution timing attacks in shared CPU contexts.
```

Further comments note that variable-time execution is a documented limitation of the system.

TOB-BSV-4: Elliptic curve operations are susceptible to timing attacks
Partially resolved in commit `130819b`. Elliptic curve scalar multiplication was updated to use a Montgomery ladder technique, using a mask-based constant-time swap operation. However, several conditional branches and significant timing variances remain. In particular, converting the scalar to a binary string relies on a variable-time loop, and iterating over the (variable) length of the resulting string will leak information about the magnitude of the scalar. Multiplying a point with a scalar whose most significant bits are zero will be measurably faster than multiplying with a scalar whose most significant bits are one.

TOB-BSV-5: AES implementations are susceptible to cache-timing attacks
Unresolved. The use of the S-box table and data-dependent lookups has not changed. BSV has added the following disclaimer to the top of the `src/primitives/AESGCM.ts` file:

> *SECURITY DISCLAIMER – AES-GCM IMPLEMENTATION*
>
> *This module provides a self-contained AES-GCM implementation intended for functional correctness and portability with minimal dependencies.*

While efforts are made to reduce timing side-channel leakage (e.g. avoiding secret-dependent branches in GHASH), JavaScript does not guarantee constant-time execution. As such, this implementation should not be used in environments where attackers can reliably measure fine-grained execution timing (e.g. shared hosts, co-resident VMs, or untrusted browser contexts).

For high-assurance cryptographic use cases, prefer platform-provided WebCrypto APIs or well-audited constant-time libraries.

TOB-BSV-6: GCM computations are susceptible to cache-timing attacks
Partially resolved in commit `def9588`. The `multiply` function now uses a standard shift-and-mask approach with no data-dependent branches or table lookups. However, it relies on the `rightShift` function, which includes a data-dependent masking operation on line 293. This can be fixed by replacing the conditional statement with the following:

```
block[i] = block[i] | (oldCarry << 7)
```

TOB-BSV-7: HMAC-DRBG is not forward-secure
Resolved in commit `1845727`. The HMAC-DRBG implementation is no longer exported, is scoped only for use in the deterministic ECDSA nonce generation code, and carries appropriate warning documentation.

TOB-BSV-8: Secret comparisons are not constant time
Resolved in commit `13506aa`. The TOTP and wallet code now use a new constant-time comparison function.

TOB-BSV-9: AES-GCM implementation is noncompliant for large inputs
Resolved in commit `923bd42`. Length values are now encoded as 64-bit values.

TOB-BSV-10: decrypt does not validate the length of ciphertexts
Resolved in commit `96f6744` (and subsequently updated for improved clarity). Length checks have been added to ensure that ciphertexts are at least as long as a minimal AES-GCM message.

TOB-BSV-11: Several issues with the message encryption protocol
Partially resolved in commit `0c7e6d2`. Documentation has been updated to describe the identified protocol limitation and to direct SDK users to use more appropriate protocols in other contexts.

TOB-BSV-12: Spurious zero-block padding is not compliant with AES-GCM standard
Unresolved. In lieu of changing the nonstandard behavior, the BSV team added a security note to the `AESGCM.ts` file, noting that the nonstandard behavior is being retained for compatibility purposes and that it should not be used as is for applications that require NIST SP 800-38D compliance.

TOB-BSV-13: AES-GCM implementation does not reject empty IV
Resolved in commit `43a7a25`. Explicit checks for key and IV length have been added.

TOB-BSV-14: Silent zero-padding of AES key is insecure
Resolved in commit `43a7a25`. Explicit checks for key and IV length have been added.

TOB-BSV-15: SHA-512 padding is noncompliant and could lead to collisions
Resolved in commit `b06c400`. Padding has been changed to match the specification by including a `padLen` value in hash objects that is used to determine the length of the length indicators.

TOB-BSV-16: DRBG seed concatenation leads to colliding outputs
Resolved in commit `d1cfa93`. New checks ensure that the entropy and nonce are both 32 bytes long.

TOB-BSV-17: Lenient Jacobian point constructor allows subtle attacks
Resolved in commit `d59e5c7`. The constructor function now canonicalizes the point at infinity, and the `eq` function handles the point at infinity using explicit checks.

TOB-BSV-18: Encoding the point at infinity triggers an assertion error
Resolved in commit `f813464`. The `encode` function now provides a specific encoding for the point at infinity (a single zero byte).

TOB-BSV-19: The htonl function is not conditional
Resolved in commit `80a6b30`. The `htonl` function has been deprecated in favor of a `swapBytes32` function, which clearly documents that it swaps byte order unconditionally.

TOB-BSV-20: Hex string conversion is fragile
Resolved in commit `26f8954`. A new function, `assertValidHex`, is used to verify that a string contains a valid hexadecimal value, causing an error if not.

TOB-BSV-21: Big integer representation of messages allows signature forgery
Resolved in commit `5c0cd27`. Range checks for signature values have been added to the `verify` function.

TOB-BSV-22: ECDSA nonce range check is incorrect
Resolved in commit `6b5b98c`. Nonce range checks have been added to the `sign` function.

TOB-BSV-23: Point decoding fails to ensure point is on the curve
Resolved in commit `0508980`. On-curve checks have been added to the initialization and deserialization functions.

TOB-BSV-24: Point addition resulting in infinity renders scalar multiplication incorrect
Resolved. The `pointAdd` function has been replaced, and points at infinity are now handled correctly during initialization.

TOB-BSV-25: Base64 decoding is not robust
Resolved in commit `5dc38eb`. The `base64ToArray` function has been updated to perform stricter validation of padding and input regularity. Additionally, calls to `indexOf` have been replaced with a case-based decoding system that will throw an error if an invalid character is encountered.

TOB-BSV-26: Missing bounds check in UTF-8 decoding
Resolved in commit `7bd6b76`. Boundary checks are now performed while decoding characters. It should be noted that an earlier version of the fix would throw an error when an invalid UTF-8 sequence was encountered, but with the current version's boundary checks, the last character is silently discarded.

TOB-BSV-27: Missing parameter checks in Chaum-Pedersen proofs and ECDSA signatures
Unresolved. No checks for the point at infinity were added to the Chaum-Pedersen or ECDSA signature verification functions.

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 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

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be 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 the controls and security properties of software. 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. Their use is also limited by the time and resource constraints of a project.