Waves Platform Security Audit

Preliminary Report, 2017-10-08

FOR PUBLIC RELEASE

Contents

1 Summary

Waves (`https://wavesplatform.com/`) is a blockchain-based platform for the exchange of digital assets, launched in 2016. This document reports on our security audit of the main Waves software, the node application. The audit aimed to discover security shortcomings in the Waves platform and help implement relevant mitigations.

We did not find any critical security issues, but report four medium-severity issues and six low-severity issues, which can all be easily fixed. The Waves platform otherwise shows good security engineering, good choice of cryptographic components with reliable implementations thereof, and has a clear design and code that facilitate auditing.

The audit was lead by Dr. Jean-Philippe Aumasson, Principal Research Engineer at Kudelski Security between June and August 2017, and took approximately 50 hours of work.

2 Introduction

This security audit covers the Waves node application, with a focus on its cryptography components and security protocols. The audit aimed to discover security shortcomings in the protocols' logic as well as implementation errors and potential for abuse from malicious users, in order to help Waves improve the security of its platform. The main components reviewed include the core cryptographic algorithms, consensus protocol, matcher algorithm, wallet, and network security.

The source code reviewed is in the repository `github.com/wavesplatform/Waves` (commit `3fb0d4d`, June 29th 2017), and includes approximately 18 kLoC of Scala. We also reviewed components of `github.com/wavesplatform/WavesGUI` (commit `3574353`, June 29th, 2017) and `github.com/wavesplatform/wavesplatform.core.js` (commit `81a5e2f`, June 30th) as requested by Waves (wallet protection, passphrase service), as well as components of `github.com/input-output-hk/scrypto` (commit `b9ea55c`, June 29th, 2017) used by the node application.

Most of the work consisted in static analysis of the source code. When needed, we ran simulations to verify some of our observations.

The audit was performed by Dr. Jean-Philippe Aumasson, Principal Research Engineer, with support from Yolan Romailler, Cryptography Researcher. Approximately 50 hours of work were spent on the engagement.

We report the following findinds:

- 4 medium-severity issues
- 6 low-severity issues

- 5 non-security, minor observation

Due to time constraints, proof-of-concepts were not created for all issues found. Some findings may therefore be false alarms, or more benign than expected. But in doubt, we prefer to report too many than too few issues.

Conversely, as with any security audit, we probably missed some issues affecting the application. For example, yet unknown security issues in the Scala language, or subtle protocol edge cases that could be discovered through formal verification.

In particular, attacks of the following type may require further investigation: possible timejacking/time-shifting attacks; attacks when the attacker controls a large (yet $< 50\%$) fraction of the nodes, as in eclipse attacks; attacks exploiting database concurrency issues or variables synchronization issues.

3 Findings

3.1 WAVES-001: Weak password-based key derivation

Severity: Medium

Description

The database including private keys and seeds is encrypted using a key derived from a password. The database creation is called from `Wallet.scala` (`createMVStore(file, Some(password))`) and defined in `wavesplatform/utils/package.scala`:

```scala
def createMVStore(file: Option[File], encryptionKey: Option[Array[Char]] = None):
↪  MVStore = {
    val builder = file.fold(new MVStore.Builder) { p =>
      p.getParentFile.mkdirs()
      new MVStore.Builder()
        .fileName(p.getCanonicalPath)
        .autoCommitDisabled()
        .compress()
    }

    val store = encryptionKey match {
      case Some(key) => builder.encryptionKey(key).open()
      case _ => builder.open()
    }

    store.rollback()

    store
  }
```

Here the password is not directly used as cryptography key. Instead, a key is derived within H2's `MVStore.Builder`. The key is derived using PBKDF2-HMAC-SHA256 with only 10 iterations, as explained and justified (?) in `main/org/h2/store/fs/FilePathEncrypt.java`:

```
           * The number of iterations. It is relatively low; a higher value would
           * slow down opening files on Android too much.
           */
7          private static final int HASH_ITERATIONS = 10;
```

However 10 iterations is way too low to offer a reasonable defense against bruteforce or dictionary attacks.

Furthermore, the salt is only 64-bit long and is chosen randomly. Therefore a collision of salts is expected after about four billion files are encrypted, which means that two files will then be encrypted with the same key.

We verified in the latest version of H2's code at `https://github.com/h2database/h2database` that PBKDF2-HMAC-SHA-256 is indeed used, with 10 iterations.

Recommendation

H2's API does not allow to specify the number of PBKDF2 iterations, so the only current solution is to use a patched version of H2. With Waves' agreement, we can also report the issue directly to H2, who may fix its subsequent version.

We recommend at least 10000 iterations.

3.2 WAVES-002: Predictable account nonces

Severity: Medium

Description

A Wallet object may include multiple PrivateKeyAccount objects, where each account seed is generated from the Wallet's 512-bit seed an a per-account 32-bit nonce, using the `generateNewAccount()` function. The Wallet's nonce is an `Int` value (32-bit) incremented by one for each new PrivateKeyAccount, and initialized to zero. Once a PrivateKeyAccount is created, its account key is stored in a persistent database ("privkeys").

This approach creates unnecessary risks:

- If the Wallet seed is compromised, an attacker can determine all account seeds, since they know which nonces were used (namely 0, 1, 2, and so on). The attacker then does not need access to the "privkeys" database.
- If the "privkeys" database is compromised, an attacker does not need the Wallet seed in order to compromise an account.

Recommendation

We propose the following changes in order to eliminate the above risks:

- Use nonces of 128 bits instead of 32 bits (for example, as a `ByteStr` object).
- Pick a random nonce for each new account, instead of incrementing the latest nonce. This eliminates the need to store the last nonce. The risk of nonce collision is negligible with a 128-bit nonce.
- Store the account nonces in the database, instead of the account seeds, and only compute account seeds when needed, using the Wallet seed.

3.3 WAVES-003: Password stored in clear in config file

Severity: Medium

Description

A wallet's password (required to unlock the private key) is stored in cleartext in the config file, e.g. `waves-mainnet.conf`:

```
# Wallet settings
wallet {
  # Password to protect wallet file
  password = "some string as password"
```

This is equivalent to not encrypting the private key, since the password required to decrypt the database is readable to anyone accessing the file system.

Recommendation

The password should be provided by the user at the application launch, temporarily stored in RAM if needed, and safely erased afterwards. It should not be stored in clear on the file system.

3.4 WAVES-004: Lack of defense against time-shifting attacks

Severity: Medium

Description

The `correctedTime()` function returns the current time in milliseconds, and is notably critical in the validation of transactions and orders. `correctedTime()` relies on the local time corrected based on the time provided by the NTP server at `pool.ntp.org`. This correction is performed every ten minutes (cf. `TimeTillUpdate`), as implemented in `Time.scala`:

```scala
private val TimeTillUpdate = 1000 * 60 * 10L
private val NtpServer = "pool.ntp.org"

private var lastUpdate = 0L
private var offset = 0L

def correctedTime(): Long = {
  //CHECK IF OFFSET NEEDS TO BE UPDATED
  if (System.currentTimeMillis() > lastUpdate + TimeTillUpdate) {
    Try {
      updateOffSet()
      lastUpdate = System.currentTimeMillis()

      log.info("Adjusting time with " + offset + " milliseconds.")
    } recover {
      case e: Throwable =>
        log.warn("Unable to get corrected time", e)
    }
  }

  //CALCULATE CORRECTED TIME
  System.currentTimeMillis() + offset
}

...

  private def updateOffSet() {
  val client = new NTPUDPClient()
  client.setDefaultTimeout(10000)

  try {
    client.open()

```

```
        val info = client.getTime(InetAddress.getByName(NtpServer))
        info.computeDetails()
        if (Option(info.getOffset).isDefined) offset = info.getOffset
      } catch {
        case t: Throwable => log.warn("Problems with NTP: ", t)
      } finally {
        client.close()
      }
    }
  }
```

However, the platform communicates with the NTP server in unauthenticated cleartext, through an `NTPUDPClient` instance. An attacker can therefore man-in-the-middle the NTP request and provide arbitrarily shifted time to the platform, potentially facilitating attacks such as double spends.

Recommendation

To migitate the risk, the `updateOffset()` should restrict offsets to realistic values, for example by checking that the offset's absolute value is less than a *panic threshold* of 1000000ms, as described in the paper Attacking the Network Time Protocol. Such a value would probably be insufficient though, hence a lower threshold should be implemented, such that all legitimate offset values are below the threshold.

3.5 WAVES-005: Potential DoS through orders flooding

Severity: Low

Description

An order book (object `OrderBook`) does not limit the number of pending orders, hence an attacker may send many orders with maximal `maxLiveTime` attribute (one month) in order to slow down the processing or orders.

Recommendation

A solution is not straightforward, since limiting the size of an order book can lead to other types of DoS: if the order book no longer accepts order after N orders registered, then newer orders will be rejected (LIFO style); if orders rollover is implemented, then older orders will be deleted (FIFO style).

Limiting the number of orders per ID is a weak mitigation, since an attacker could easily create a multitude of IDs.

The best solution may be rate-limiting the number of orders registered, to a frequency minimizing the risk to quality of service, based on measurements of previous activity.

3.6 WAVES-006: Potential DoS through malformed blocks

Severity: Low

Description

The `Block.parseBytes()` and `Block.transParseBytes()` function read variable data length values from the received blocks, for example

```
    val cBytesLength = Ints.fromByteArray(bytes.slice(position, position + 4))
    position += 4
    val cBytes = bytes.slice(position, position + cBytesLength)
```

Other length values reader are `BytesLength` in `parseBytes()`, and `transactionLengthBytes` as well as `transactionBytes` in `transParseBytes()`. Invalid length values will cause an exception `java.lang.IllegalArgumentException`, which doesn't seem to be caught by caller functions.

A similar issue occurs within IssueTransactions.

Recommendation

Perform some basic bound check on the user-provided length, and make sure that any related exception is caught.

3.7 WAVES-007: Suboptimal entropy in passphrase generation

Severity: Low

Description

`wavesplatform.core.js/src/core/passphrase.service.js` generates a random passphrase using a wordlist as follows:

```
                var crypto = $window.crypto || $window.msCrypto;
                var bits = 160;
                var wordCount = 2048;
                var random = new Uint32Array(bits / 32);
                var passPhrase = '';

                crypto.getRandomValues(random);

                var i = 0,
                    l = random.length,
                    n = wordCount,
                    words = [],
                    x, w1, w2, w3;

                for (; i < l; i++) {
                    x = random[i];
                    w1 = x % n;
                    w2 = (((x / n) >> 0) + w1) % n;
                    w3 = (((((x / n) >> 0) / n) >> 0) + w2) % n;

                    words.push(wordList[w1]);
```

```
        words.push(wordList[w2]);
        words.push(wordList[w3]);
    }

    passPhrase = words.join(' ');

    crypto.getRandomValues(random);

    return passPhrase;
```

Here for each of the five random 32-bit words in the `Uint32Array(5)` variable `random`, three 11-bit indices are extracted `w1`, `w2`, and `w3`. These 33 bits of information thus have only 32 bits of entropy, thus the passphrase has 155 bits of entropy instead of 160 bits as intended.

Recommendation

A simple fix would be the following (note the `Uint16Array` type instead of `Uint32Array`),

```
    var crypto = $window.crypto || $window.msCrypto;
    var bits = 160;
    var wordCount = 2048;
    var wordsInPassPhrase = Math.ceil( bits / Math.log2(wordCount))
    var random = new Uint16Array(wordsInPassPhrase);
    var passPhrase = '';

    crypto.getRandomValues(random);

    var i = 0, words = [], index;

    for (; i < wordsInPassPhrase; i++) {
        index  = random[i] % wordCount;
        words.push(wordList[index]);
    }

    passPhrase = words.join(' ');

```

```
                crypto.getRandomValues(random);

                return passPhrase;
```

We assume that the final call to `crypto.getRandomValues()` is intended to erase the sensitive data in the variable `random`.

3.8 WAVES-008: Negative block timestamps undetected

Severity: Low

Description

A negative block's timestamp (`Long` type) in `blockConsensusValidation()` will not directly be detected, and `calcBaseTarget()` will return a negative value. However the final comparison between hit and target will fail (unless the previous block's target was also negative).

Recommendation

Add a check near the following lines:

```
    val blockTime = block.timestamp

    require(blockTime - currentTs < MaxTimeDrift,
↪    s"Block timestamp $blockTime is from future")

    if (blockTime > fs.requireSortedTransactionsAfter) {
      require(block.transactionData.sorted(TransactionsOrdering.InBlock) ==
↪    block.transactionData, "Transactions must be sorted correctly")
    }
```

3.9 WAVES-009: Build supporting insecure dependencies

Severity: Low

Description

The sbt build system defines a number of dependencies without enforcing specific versions (or ranges thereof):

```
libraryDependencies ++=
  Dependencies.network ++
  Dependencies.db ++
  Dependencies.http ++
  Dependencies.akka ++
  Dependencies.serialization ++
  Dependencies.testKit ++
  Dependencies.itKit ++
  Dependencies.logging ++
  Dependencies.matcher ++
  Dependencies.p2p ++
```

However, older versions may include vulnerabilities impacting Waves. For example, in February 2017 akka fixed a potential remote code execution exploitable via akka's ActorSystem (see akka's vulnerability description in `http://doc.akka.io/docs/akka/2.4/security/2017-02-10-java-serialization.html`).

Recommendation

Enforce use of latest subrevision or of a specific version in `build.bst`.

3.10 WAVES-010: Forgery of order history GET requests

Severity: Low

Description

The “Order History by Public Key” GET request optionally take an `amountAsset` ID and a `priceAsset` ID, not covered by the signature. An attacker may therefore intercept a request for given assets, and change it into a request for other IDs. Relevant code is in `MatcherApiRoute.scala`.

The actual risk is low, however, since these requests are sent through a TLS tunnel, and the request should be sent within a window of `maxTimeStampDiff` milliseconds, as enforced in the verification step:

```
def checkGetSignature(publicKey: String, timestamp: String, signature: String):
↪  Try[String] = {
  Try {
    val pk = Base58.decode(publicKey).get
    val sig = Base58.decode(signature).get
    val ts = timestamp.toLong
    require(math.abs(ts - NTP.correctedTime()).millis <
↪  matcherSettings.maxTimestampDiff, "Incorrect timestamp")
    require(EllipticCurveImpl.verify(sig, pk ++ Longs.toByteArray(ts), pk),
↪  "Incorrect signature")
    PublicKeyAccount(pk).address
  }
}
```

Recommendation

We recommend to change the signature (verified by `checkGetSignature`) as follows:

- Include the `amountAsset` and/or `priceAsset` parameters, if provided, in the signed data

- Do not include the public key (`pk`) in the signed data, since the validity of the signature implicitly authenticates the public key (an attacker can't turn a valid signature for pk1 into a valid signature for pk2 if they don't know pk2's private key)

3.11 Minor observations

The following points do not address security risks, but potential improvements in terms of consistency, efficiency, or safety:

3.11.1 Automated static analysis

The scapegoat static analyzer did not reveal any security issue, but a number of potential improvements in terms of robustness, clarity, or efficiency.

3.11.2 Potential division by zero

In `base58Length()`:

```
   def base58Length(byteArrayLength: Int): Int = math.ceil(math.log(256) /
↪  math.log(58) * byteArrayLength).toInt
```

This function should first check that `byteArrayLength` is non-zero.

3.11.3 Biased network nonces

Network nonces are computed pseudorandomly using the following function, which returns a number between 0 and 1998000 ($\approx 2^{14.51}$):

```scala
private def randomNonce: Long = {
  val base = 1000

  (Random.nextInt(base) + base) * Random.nextInt(base) + Random.nextInt(base)
}
```

However the values are not uniformly distributed, as shown in Figure 3.1. Consequently, the entropy of the distribution is suboptimal, being ≈ 13.60, against ≈ 14.51 ideally, if the distribution were uniform. It follows that a nonce repetition is expected after approximately 128 nonce generations.

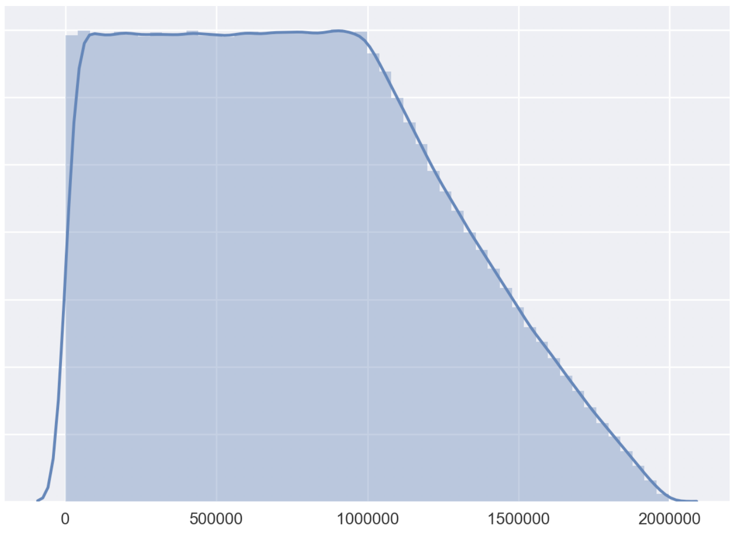

Figure 3.1: Distribution of values returned by `randomNonce()`.

3.11.4 Inconsistent checksum algorithms

Checksums of Account and Message objects use different hash functions (respectively, `SecureCryptographicHash()` and `FastCryptographicHash()`). It would be more consistent to use the same function. Too, the name `FastCryptographicHash()` may be understood as a insecure hash function, whereas it's not the case.

3.11.5 Max order amount does not match the specs

The matcher's specs at https://github.com/wavesplatform/Waves/wiki/Matcher defined a maximal order amount (MaxAmount) of 10^{16}, whereas the code specifies 10^{18}:

```
object Order {
  val MaxLiveTime: Long = 30L * 24L * 60L * 60L * 1000L
  val PriceConstant = 100000000L
  val MaxAmount: Long = 100 * PriceConstant * PriceConstant
```

4 About

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit `https://www.kudelskisecurity.com`.

Kudelski Security
route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland