Digital Identity
LABORATORY

REPORT

HYPERLEDGER URSA CODE REVIEW

ABSTRACT
Contains results of review of the Hyperledger URSA codebase.

DATE
March, 2022

Table of Contents

Executive summary **3**
Engagement Scope and process 3
Findings Summary 4
Copyright Notice & Disclaimer **5**
The Digital Identity Laboratory of Canada **5**
Hyperledger URSA Code review **6**
Introduction 6
Scope in detail 6
Findings in detail **10**
General Observations and Recommendations 10
Security Defects Raised During Review 20
Appendix A - About the Authors **28**

Executive summary

The Digital Identity Laboratory of Canada (IDLab) was engaged to perform a security and cryptography code review of the Hyperledger URSA crypto library.

Hyperledger URSA is a shared cryptographic library used to avoid duplicating other cryptographic work and increase security in the process. The library is an opt-in repository (for Hyperledger and non Hyperledger projects) to place and use crypto.

Project sponsors, several Canadian public sector entities, and Interac, recognise that end users rely on strong security to deliver the high levels of assurance required of important consumer transactions with their governments and their financial institutions. Many of the world's leading advancements in decentralized identity, several originating in Canada, rely on the Hyperledger family of technologies. As such, the URSA library is a key component, depended upon to provide the security overlay for decentralized identity.

Engagement Scope and process

The URSA library review included:

- a code review that examined:
  - entry points
  - coding standards
  - data storage and transfer
  - APIs and their security
  - third party library usage
  - language issues
  - logic flaws
- Cryptography best practices including examination of:
  - cryptography and key management
  - entropy
  - best practice cryptography usage

The review did not include an assessment of the cryptographic algorithms themselves, their math or suitability. We limited examination to assessment of sound implementation within URSA.

The examination and validation process included checkpoints with the project sponsor group as well as ongoing consultation with a key group of URSA community contributors and primary architects of the codebase.

Findings Summary

In general, the review noted few relatively minor security defects and some general observations for library improvement.

These can be briefly described as follows:

- Minor build issues
- Cautions to consider when building, primarily with third party libraries or integrations
- Minor issues related to lack of support for message augmentation
- Minor issues related to subgroup validation
- An issue related to public key validation

Please refer to the "Findings in Detail" section of this report for more detail.

These findings were reviewed with URSA community experts and the Findings detail includes reaction and identification of actions to be taken (e.g., noting the issue for inclusion in the normal workflow for addressing Problem Reports).

Copyright Notice & Disclaimer

The Digital Identity Laboratory of Canada

The Digital Identity Laboratory of Canada (IDLab) is an independent Canadian non-profit entity dedicated to advancing digital trust by breaking down barriers to digital ID adoption. The IDLab promotes technical conformity and interoperability of user-centric digital identity solutions. The IDLab is not an incubator and does not develop or sell digital identity solutions. Our mission is to accelerate the adoption, development and knowledge of compliant and interoperable digital ID solutions.

We accomplish this mission by delivering education, assessment, advisory and sandbox services. When delivering those services, IDLab preserves its neutrality and independence by complying with its Policy on the Limitations of Commercial Activities.

Hyperledger URSA Code review

Introduction

The URSA library is written in Rust with some of its dependencies being implemented in C. We examined the latest code release (V0.3.7) available at the time of project commencement.

The code examined may be found here[1].

Third party dependencies include:

- openssl 1.1.0j
- libsecp256k1

Scope in detail

The code review has involved reading the code in its entirety as well as employing scripts to look for common issues. Code examination included:

1. Entry points - entry points of the libraries are numerous for two principal reasons: firstly, the libraries define a large number of functions with public scope, secondly bindings are also present which constitute further means to call into the library from client code. Approximate entry points:
   - C linkage: ~138
   - Rust: ~2732
2. Coding standards – validate and assess the coding standards utilized to implement the source code and whether this standard is stringently enforced through the use of secure/documented coding inclusion/merge processes.
3. Cryptography and key management, including but not limited to, random number generation, key generation, signing, verification, identify backdoors.
4. Data storage and transfer – assess how the code accesses the local filesystem of the machine the code is being executed on. In this case, since the code is a library meant to be utilized by client code, the security of data

[1] https://github.com/hyperledger/ursa/tree/v0.3.7 (34ef3926b633a5eaff5b220c40005bbc8cd04861)

storage is the responsibility of the client code barring extenuating circumstances (such as caching, temporary file creation, etc., that may be created upon execution flow from client to library code).

5. APIs and API security – the API of the tested libraries is essentially the entire source code base since the source is available and thus modifications and entry points can be created as the developers see fit. The security of the entry points is thus assumed to be performed from the viewpoint of the client programmer linking their code to the library through any entry point.
6. Language issues including but not limited to, memory issues, race conditions, uncaught exceptions.
7. Logical flaws in implementation.
8. Use of/Third Party Libraries - assess their suitability for use by the URSA library.

We also performed a limited assessment of cryptographic best practices, focused on ensuring that the use of cryptography within the library is secure. This included examination of:

1. Entropy – where random values are being generated (e.g. for key generation or for challenge response protocols) we examined the method of random number generation to determine appropriateness.
2. Best practices – we examined key lengths and use of curves (for Elliptic Curve cryptography using external guidance (such as that from NIST) to assess whether best practices are being followed in the use of cryptography.
3. Best practice implementation of published algorithms.

NOTE: we did not assess the cryptographic algorithms themselves or the academic merit of the implementation of those algorithms. For the avoidance of doubt, the intent here was to say we did not review the math foundation of the cryptographic algorithm, we have however assessed that the published algorithm has been implemented correctly within the URSA code library.

Methodology notes

The table below provides additional insight into some of the key components of our examination methodology, identifying specific elements of the URSA library examined. This is not an exhaustive list, noting only some of the more important items as we perceive them to provide additional context for some of the findings in detail to follow.

Component	Comments
Validation on public keys All operations that operate on points in public keys must first validate those keys. We identified all operations that operate on points in public keys, and through code inspection verify that those keys are validated before the operations are performed.	The BBS signature scheme (https://identity.foundation/bbs-signature/draft-bbs-signatures.html) specifies that the sign, verify and proof operation of the BBS scheme must validate public keys being used. These operations are implemented in the following source locations: **Signature generation**: Impl Signature :ursa\libzmix\bbs\src\signature.rs lines 248-269 **Blind Signature generation**: Impl BlindSignature :ursa\libzmix\bbs\src\signature.rs lines 144-189 **Signature verification**: fn Verify : ursa\libzmix\bbs\src\signature.rs lines 277-312 impl PoKOfSignatureProof :ursa\libzmix\bbs\src\pok_sig.rs lines 293-322 **Proof of Knowledge verification**: fn verify \libzmix\bbs\src\pok_sig.rs lines 342-430
Membership checks on verification of a signature When verifying a signature, the verification algorithm should verify that the signature point is an element of the subgroup with order r.	The BBS signature scheme (https://identity.foundation/bbs-signature/draft-bbs-signatures.html) specifies that the membership check in the operation to verify a signature should be performed. The verify operation is implement in fn verify in ursa\libzmix\bbs\src\signature.rs line 277-312
Randomness considerations	The nonces used in signature proofs are not

Nonces in signature proofs should be generated from a trusted randomness source.	generated within the URSA library but are supplied as input data to the URSA library APIS. The URSA library provides a method for securely generating random nonces, and example code documented within the library source code shows their proper usage.
Side channel attack protection Operations on the underlying pairing-friendly elliptic curves should run in constant time, implementing the same sequence of instructions and same memory access to protect secret keys for side channel attacks.	All multiplication and other complex operations like modular exponentiation operating on numbers larger than the native register size will be susceptible to side channel attacks. By the nature of the operation the complexity of the calculation depends upon the complexity and size of the input data. Furthermore, since the URSA library is a software implementation, wherein efficiency is a design factor, no side-channel attack protections are implemented.
Full source code review of Hyperledger URSA library	Best practices code review techniques are too lengthy to elaborate upon. Please refer to the Scope description for additional insight into the considerations used for basic code review.
Assess dependencies for any known security vulnerabilities	Dependencies for the project are one of either native code or RUST crates. While it is possible to ascertain if a given dependency contains a publicly known and disclosed vulnerability, the number of dependencies required by the URSA library and its components is too great to exhaustively review all the code of said dependencies.
Review build environment The source code build environment of the RUST language involves the heavy reliance on third-party components (crates) and their building into the resulting project. Securing the build environment in this case is very important since the malicious inclusion of Rust creates containing potential issues could completely compromise the resulting builds of the Ursa/zmix libraries and thus any software utilizing/linked.	

Findings in detail

The findings below include a set of general observations and recommendations and a number of security defects noted.

General Observations and Recommendations

1. Structure and maturity of the Hyperledger URSA library

Currently the URSA library is organised as two distinct libraries with which other programs can link against, these are as follows:

libursa – the core cryptographic library.

- The library can be built either to require the use or linking of the OpenSSL library for cryptographic support or use various Rust crates implementing the same algorithms while not requiring non memory safe Rust code.

libzmix – the core zero-knowledge proof library, built utilising the tools and algorithms from libursa.

ursa_sharing – cryptographic secret sharing scheme implementations.

The Hyperledger URSA library is stabilising, but the structure of the resulting libraries is in a state of flux. This is exemplified readily by recent merges into the code base that seek to 'crate' the various aspects of the library in order to split the implementation down into sub-crates sufficient to implement specific cryptographic functions and schemes independently and piece meal enough to mean users of the library can import only the crates that they require in order to implement their functionality.

The current source code base includes the above 3 libraries, however, the intention as stated is to convert/break said libraries into five crates: **ursa_accumulators, ursa_core, ursa_encryption, ursa_sharing[1], ursa_shortgroupsignatures, ursa_signatures**.

Windows build – mismatch of library names

- The URSA library documentation on building from source (https://github.com/hyperledger/ursa#building-from-source), indicates that the build will result in artefacts **libursa.dll** and **libursa.lib** for a windows build, a build from the **3.7.0** source code succeeds without reporting any error but creates artefacts **ursa.dll** and **ursa.lib** and not **libursa.dll** or **ursa.lib**.

Cargo build issues – several aspects of the source either fail to build or fail to build unit tests and fail to execute unit tests.

- The 'asm' feature, so-called because it intends to be linked against 'assembler' optimised versions of the underlying cryptographic primitives (leveraging OpenSSL) fails to build from the sources obtained from the GitHub repository using the stable and nightly RUST compilers and associated toolchain.
- A small number of unit tests are currently failing to execute, however upon inspection this is due to the developers making simple mistakes in the definition of the unit tests themselves and there is no serious underlying failure or mistake.

Bindings lacking maintenance – the libraries provide a C linkage binding from which other languages can be bound to the RUST versions of the libraries. However, during the testing of these bindings a number of issues were found with the binding declarations for C linkage mismatching the underlying symbol with respect to the parameters to be provided to the function for correct functioning.

- An example of this issue relates to the function ` random_bytes` function which defines a symbol with a single parameter while the underlying function takes 3 parameters resulting in a crash when the function is called from within a C program:

```
        frame #16: 0x00000001000d2556
ursa-test`random_bytes(output=0x000000000000000a, bytes=140702053824232,
err=0x00007ff7bfeff6f8) at mod.rs:309:15

        frame #17: 0x0000000100000f6e ursa-test`URSARandomBytes + 14

        frame #18: 0x0000000100000f4b ursa-test`main + 27
```

Feedback from initial Hyperledger URSA community review:

There are intentions on the current roadmap to refactor the library to clean up several minor issues. These issues should be addressed during that process and this report has helped inform this upcoming effort. Additional information regarding the URSA's project health and current plans is available in the Q1 2022 Hyperledger URSA project updates report.

2. Dependencies used in Hyperledger Ursa Library

URSA can optionally use the following external dependencies:

- openssl 1.1.0j or greater (Written in C/assembler)
- libsecp256k1 (Written in C)

URSA with ASM

Recommendation: when URSA code is compiled and built with assemblies (ASM) from external dependencies the following aspects must be considered.

- Review if any known vulnerabilities exist in the external dependency (e.g., CVE-2021-23841 OpenSSL versions 1.1.1i and below are affected by this issue)
- Any version change of the external dependency must be audited.

URSA with Rust only and no ASM

Recommendation: with Rust only approach, the following aspects must be considered.

The cryptographic implementation in Rust only code must be audited for correctness.

Feedback from initial Hyperledger URSA community review:

This can be characterized as an implementation caution. There are a number of those noted in this report and we will continue to work with the URSA community to decide how to better expose this type of usage advice.

3. <u>Manual check needed to confirm Nullifying sensitive Rust Objects</u>

In Rust, Zeroize crate/dependency (https://docs.rs/zeroize/latest/zeroize/) nullify the buffers implicitly. However, the Rust objects do not make it clear whether Zeroize is attached. Furthermore, while the Zeroize crate seems to have permeated the URSA library source code, it does not appear to have made its way into the dependency list of the zmix library.

Recommendation: Therefore, we need a manual check of all sensitive objects to ensure Zeroize is applied correctly.

To facilitate this, a process of implementing simplistic 'users' of the URSA library in the C programming language is to be performed wherein calls to the URSA library will be made with known inputs and the outputs verified while memory profiling is performed (in a debugger) can ascertain if memory is cleared upon returning from the library back to the caller. It is expected, the valid outcome, that the resulting memory profile will reveal that while the memory held by the calling program is left intact, the internal processing of the URSA library will leave minimal information lingering in memory after the return of program control to the caller.

Feedback from initial Hyperledger URSA community review:

This will be noted as an issue for further investigation and potential enhancement. At the very least, it will be handled in the same manner as those items we have noted herein as "implementation cautions".

4. Suitability of Rust Programming language for secure implementation

RUST, as a programming language aims to afford the programmer the ability to write low-level, highly performant code, compliable to optimised machine code while focussing on memory safety and thus higher levels of security assurance compared to the traditional C/C++ pair. In essence the utilisation of the Rust programming language effectively shields the resulting application, provided no linking or control flow passes to non-Rust code, from the following types of memory related issues:

- Buffer overflow
- Use-after-free (UAF) - wherein a heap/dynamically allocated pointer is dereferenced after being subjected to the deallocation operation.
- Double-free – wherein a heap/dynamically allocated pointer is subjected to the deallocation operation more than once.
- Null dereference – attempting to access the zero-page.
- Uninitialized memory access – reading data from memory that has not been initialised with a known value.

In modern parlance, the notion of 'safe' and 'unsafe' Rust is taking hold wherein Rust code that offers integration with programming languages that do not offer the same level of memory safety is deemed to be 'unsafe' while pure Rust implementations are deemed 'safe'. In general, the application developer should tend towards utilizing Rust exclusively if that is a potential possibility when developing a secure solution.

Feedback from initial Hyperledger URSA community review:

This can be characterized as an implementation caution. There are a number of those noted in this report and we will continue to work with the URSA community to decide how to better expose this type of usage advice.

5. Suitability of other languages calling into RUST

In a previous section we had highlighted the notion of 'safe' and 'unsafe' Rust code. This determination is almost entirely defined by the use of other programming languages to implement functionality that is either called from Rust code or makes calls into Rust code (when Rust is used to develop a library to be used by other code). While there are no security concerns immediately presented through the use of Rust code or the integration of such code into other projects implemented in other programming languages, it is highly advantageous to keep as much of the implementation in native Rust as much as is possible in order to leverage the memory safe aspects of the language. This is particularly important when implementing and utilizing cryptographic primitives since Rust offers complete control over the allocation and deallocation of memory and the subsequent reference counting required to effectively and efficiently wipe and thus zero memory leaving little information leaked.

Currently the URSA library provides extension and linking to the C programming language by default (this can be disabled, and symbols removed from the resulting libraries). It is thought that this functionality was explicitly chosen by the developers since the C programming language and its symbolic linkage can be integrated into, and called from, almost any other modern programming language. Indeed, the Hyperledger developers distribute 'shims' or slim interoperability libraries for URSA for the following programming languages:

- Python – https://github.com/hyperledger/ursa-python
- Go - https://github.com/hyperledger/ursa-wrapper-go

It would be logical to assume that the developers will seek to add further common programming languages to this list over time with Java and C# being the most logical to include.

Feedback from initial Hyperledger URSA community review:

This will be considered by the URSA community for downstream inclusion in their roadmap when sufficient demand becomes apparent. To date JAVA and C# demand has been very low. BBS+ Signatures, used more widely, does have a wider variety of integration tools.

6. Use of code obfuscation

Utilisation of obfuscation techniques in securing compiled code is typical within the industry when the security of said code is of the utmost importance. In general, however this is only performed when the execution environment cannot be trusted, or could be considered compromised, in the general case (the most likely scenario would be the code running on a mobile device which is portable and of an unknown design/state). Another aspect commonly associated with obfuscation is the use of integrity validation in the execution of the software. This takes the form of trips and validation inserted into resultant binaries which attempt to check the code as it is running in order to ascertain if the execution environment of the code has been compromised or otherwise subverted at execution time.

However, in the case of Ursa/zmix, any obfuscation applied to the resultant libraries would mainly serve to reduce the speed of the resultant library and the cryptographic operations with little to no benefit since the security of the implementation is entirely within the algorithms themselves and nothing is secret.

At the time of writing, no execution environment details are available for review with respect to the potential deployment of Ursa/zmix and as a corollary no defined judgement can be made with respect to their applicability in any project linking to the Ursa/zmix libraries.

Recommendation: use of code obfuscation for the Ursa libraries is not recommended, as it would deliver minimal benefits from a security perspective while having a detrimental impact on performance.

Feedback from initial Hyperledger URSA community review:

Code obfuscation was deemed out of scope for the community. The perceived issue this could raise has to do with timing attacks that can be mitigated in other ways.

7. Security of the build system RE cargo and pulling third-party crates

As is common, and indeed by design, a Rust code base build system requires a large number of third-party code crates which require obtaining from public sources prior to the code being built and the resulting binaries produced. The security of this process is of the utmost importance and should be performed in a secure environment, isolated and automated (CD/CD).

The URSA and zmix libraries are built in the standard mechanism of other Rust code bases and utilize the cargo template wherein dependencies are enumerated in a workspace 'Cargo.toml' file while instructs the cargo build toolchain which crates to obtain and build as part of the build process. In general, this will not pose a problem, and indeed in the case of URSA, the extensive unit tests should be sufficient so as to prove that the code for the crates brought into the build process operate as intended. However, to create certainty, a 'Cargo.lock' should be enforced so as to lock the dependencies pulled in as part of the build process to versions controlled explicitly by the lock file. These are enforced through the use of cryptographic signatures on the crates' contents.

Recommendation

These lock files should ideally be stored in a repository locally when building and optionally included into a fork of the URSA library repository by means of a sub-module.

Feedback from initial Hyperledger URSA community review:

This can be characterized as an implementation caution. There are a number of those noted in this report and we will continue to work with the URSA community to decide how to better expose this type of usage advice. The community will also consider this in more detail for potential enhancement to the library.

8. <u>Lack of timing-based side-channel attack protection</u>

The software nature of the library necessitates the efficient functioning of the library with respect to the cryptographic operations utilized/implemented. As a corollary the algorithms do not implement protections against side-channel attacks since they are implemented to be efficient. The operations used are often by their nature non-constant time (or even linear in the size of the inputs) and as such side-channel attacks are immediately applicable depending on the execution environment of the infrastructure/code linked to the URSA library.

The exploitability of any side-channel attack is speculated to be extremely complex, requiring incredibly high levels of access to the infrastructure the code is being executed on. Thus, the exploitability of the issues is highly dependent on the execution environment the targeted code is being run in and the degree of access a would-be attacker has to the environment and/or network through which any results of the computations are disclosed over.

In the case where the code to be attacked is executed within a server environment, the physical exploitability is speculated to be close to zero if the infrastructure itself can be said to be highly constrained, that is inside a highly secure data centre where physical access to machines is easily controlled. Furthermore, given the high computational throughput, access to the physical network in order to aid with the timing attack is highly unlikely to provide any advantage to the attacker.

However, the exploitability of such issues increases massively when the algorithms are to be executed on a mobile device or a device with minimal/constrained processing power. For instance, should the algorithms be executed on a mobile device, timing attacks become possible, albeit still extremely difficult for a combination of reasons. Firstly, access to the device would still be required and secondly, if the attacker only has access to the network, then network 'jitter[2]' would likely preclude any such attack since these devices typically only possess wireless communications which are inherently more susceptible to network 'jitter'. In reality, networks would need to get so fast with such low latency that the resulting risk is extremely low and there is no recommended mitigation necessary to address this.

[2] 'jitter' as it relates to a network is the time delay between packets or layer-2 and above data can be sent across the network relative to low level protocols required to facilitate the wireless medium.

Feedback from initial Hyperledger URSA community review:

This had been considered in the past and considered to be an extremely low risk. However, we will continue to discuss with the URSA community to come to consensus on whether the points raised herein warrant opening an issue for further investigation. The sentiment is that some enhancements are likely to be defined to eliminate typical approaches to this type of attack. Comprehensive addressing of this issue will not be in scope.

9. BLS implementation does not support message augmentation

From examination of the source code, it would appear that the BLS code within Libursa, does not support message augmentation. In a message augmentation scheme, signatures are generated over the concatenation of the public key and the message, ensuring that messages signed by different public keys are distinct. Message augmentation is one technique to protect against forgery of an aggregate signature by crafting a special public key. The Basic BLS scheme protects against these attacks by insisting all signed messages are different, the addition of proof of possession steps would also protect against such attacks. The use of Basic BLS / message augmentation / proof of possession are options, but since Libursa does not support message augmentation this option would not be available to its users.

Feedback from initial Hyperledger URSA community review:

Further consideration of this comment will be done in the URSA community with the potential addition of some enhancements being defined and added to the roadmap.

Security Defects Raised During Review

1. Known vulnerabilities in OpenSSL

Severity	Low

Description

URSA uses OpenSSL 1.1.0j or greater. There are known vulnerabilities in version 1.1.0j

- CVE-2021-23841 affects OpenSSL versions 1.1.1i and below
  - https://www.cvedetails.com/cve/CVE-2021-23841/
- CVE-2019-1543 affects OpenSSL versions 1.1.0j and below

From the above issues, it is believed that only the CVe-2019-1543 entry would be an issue in the case of URSA when built linked against OpenSSL since the other CVE relates to aspects of the OpenSSL library which are not used nor linked to in URSA.

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-1543

Related Documentation / Code

External Dependencies:

- openssl 1.1.0j

Analysis

When URSA code is compiled and built with assemblies (ASM) from external dependencies they should be checked for any known security vulnerabilities. The OpenSSL versions 1.1.1i and below are found vulnerable to potential denial of service attack (see CVE-2021-23841 for details).
We recommend use of the latest version of OpenSSL, at the time of writing this is 1.1.1j, and also keep track of any vulnerabilities found in OpenSSL.

Recommendation

Use the latest version of openssl 1.1.1j

Feedback from initial Hyperledger URSA community review:

This can be characterized as an implementation caution. There are a number of those noted in this report and we will continue to work with the URSA community to decide how to better expose this type of usage advice.

2. Implementation does not check that a signature is an element of a prime order subgroup

Severity Low

Description

The BBS signature scheme [BBS-SS] specifies that the membership check in the operation to verify a signature should be performed. As described in that specification this check is required for the following reasons:

1. For most pairing-friendly elliptic curves used in practice, the pairing operation e (Section 1.3) is undefined when its input points are not in the prime-order subgroups of E1 and E2. The resulting behaviour is unpredictable, and may enable forgeries.
2. Even if the pairing operation behaves properly on inputs that are outside the correct subgroups, skipping the subgroup check breaks the strong unforgeability property [ADR02].

There is no sign of any validation of the of the Subgroup order of the signature verification algorithm:

fn verify in ursa\libzmix\bbs\src\signature.rs line 277-312

Related Documentation/Library

[BBS-SS] M. Lodder, T. Looker, A. Whitehead, "The BBS Signature Scheme", https://identity.foundation/bbs-signature/draft-bbs-signatures.html

[ADR02] An, J. H., Dodis, Y., and T. Rabin, "On the Security of Joint Signature and Encryption", April 2002, https://doi.org/10.1007/3-540-46035-7_6

Analysis

The implementation of BBS signature verification in the URSA library should implement checks of the signature subgroup, to maintain confidence in the authenticity of signatures. Without such a check confidence in the unforgeability of the signature is weakened which can have potential impact on the non-repudiation of the data.

It should be noted that validation of the subgroup can be computationally expensive although some techniques allow faster subgroup checks [BOW19].

A usable implementation of the sub-group check is available as a Rust crate distributed by the Apache Foundation. Further information can be found in the following location, https://github.com/apache/incubator-milagro-crypto-rust. The resulting Rust crate is titled 'amcl' (https://crates.io/crates/amcl).

Recommendation

Signature verification routine should be changed to implement checks on the signature subgroup.

Feedback from initial Hyperledger URSA community review:

This will be raised as an issue and be subject to normal community workflow to address issues. The community agreed that the classification of this item as low risk was appropriate.

3. BBS Public key validator does not check the subgroup of the public key or its generator

Severity Low

Description

The BBS signature scheme [BBS-SS] specifies that the public key validation procedure (KeyValidate) should include subgroup checks on the public key and its generators. A public key validate method is implemented in the URSA library in libzmix\bbs\src\keys.rs starting at line 149, which checks if the generators of the public key are not the identity element but does not perform any subgroup checks.

The Apache Milagro library used by the URSA library when configured to be compiled using Rust only code, does include an API to perform subgroup checks, however we do not believe these are utilised by the URSA library to validate that the signature is of the appropriate subgroup.

Without sufficient checks inadvertent or deliberate supplying of public key small subgroups could lead to vulnerabilities as described here:

https://eprint.iacr.org/2015/247.pdf

Related Documentation/Library

libzmix\bbs\src\keys.rs starting at line 149

Analysis

The BLS12-381 curve, like many other pairing-friendly curves, has elliptic curve pairing groups G1 and G2 with nontrivial co-factors, which give rise to small subgroup attacks. It is therefore recommended for implementations to check that the points really exist within G1 and G2 as they are being decoded. Without such checks this may cause vulnerabilities as has occurred in other cryptographic schemes see https://ristretto.group/why_ristretto.html#pitfalls-of-a-cofactor.

As described in defect 2 the naïve approach of multiplying by q to ensure it is in the q order subgroup is computationally expensive, other methods have been proposed [BOW19].

The Apache Milagro library used by the URSA library when configured to be compiled using Rust only code, does include an API to perform subgroup checks, however we do not believe these are utilised by the URSA library to validate that the signature is of the appropriate subgroup.

Recommendation

The URSA library should be modified to include subgroup checks during public key validation.

It should be noted that validation of the subgroup can be computationally expensive although some techniques allow faster subgroup checks [BOW19].

Feedback from initial Hyperledger URSA community review:

This will be raised as an issue and be subject to normal community workflow to address issues. The community agreed that the classification of this item as low risk was appropriate.

4. Libursa BLS Module does not check public keys

Severity	Medium

Description

The BLS implementation within the URSA library (libursa\src\bls\mod.rs) does not appear to have any validation of public keys. These are required by the BLS-signature specification [BLS].

The BLS signature scheme [BLS] specifies in its security considerations for validating public keys that:

“All algorithms in Section 2 and Section 3 that operate on public keys require first validating those keys. For the basic and message augmentation schemes, the use of KeyValidate is REQUIRED.”

KeyValidate requires all public keys to represent valid, non-identity points in the correct subgroup. A valid point and subgroup membership are required to ensure that the pairing operation is defined (Section 5.2).

A non-identity point is required because the identity public key has the property that the corresponding secret key is equal to zero, which means that the identity point is the unique valid signature for every message under this key. A malicious signer could take advantage of this fact to equivocate about which message he signed.”

The validation of public keys is required by [BLS] in the following operations; CoreVerify (verification of a signature), CoreAggregateVerify (Verification of an aggregated signature over several public keys), PopVerify (Proof of possession verification).

Related Documentation/Library

[BLS] BLS signatures, D. Boneh, S. Gorbunov, R. Wahby, H. Wee, Z. Zhang. 10 Sep 2020, https://www.ietf.org/archive/id/draft-irtf-cfrg-bls-signature-04.txt

Analysis

As pointed out by [BLS] without verification of the public keys, a malicious signer could use a secret key of zero, and identity public keys, which result in all signatures for those keys also being the identity point. Such a signer could then reasonably argue they did not in fact sign any particular message, and furthermore it is trivial for this attacker to produce a signature for any message. The public key validation should also validate the subgroup. We have described in other defects the importance of subgroup checks.

Recommendation

The URSA library should be modified to include public key validation within the operations using public keys as required by the BLS specification.

Feedback from initial Hyperledger URSA community review:

This has been considered in the past and not implemented due to performance concerns. Discussion resulted in agreement to raise as an issue for further consideration. One mitigation approach suggested was to implement an enhancement that could be implemented as an optional feature. For those concerned with performance, other mitigation approaches in the form of "implementation advice" could be used.

5. Libursa BLS Module does not perform subgroup checks on signatures

Severity	Low

Description

The BLS implementation within the URSA library (libursa\src\bls\mod.rs) does not appear to have any validation signatures. These are required by the BLS-signature specification [BLS].

The validation of public keys is required by [BLS] in the following operations; CoreVerify (verification of a signature), CoreAggregateVerify (Verification of a aggregated signature over several public keys), PopVerify (Proof of possession verification)

Related Documentation/Library

[BLS] BLS signatures, D. Boneh, S. Gorbunov, R. Wahby, H. Wee, Z. Zhang. 10 Sep 2020, https://www.ietf.org/archive/id/draft-irtf-cfrg-bls-signature-04.txt

Analysis

The BLS signature scheme [BLS] specifies in its security considerations for validating public keys (section 5.2) that:

“This check is REQUIRED of conforming implementations, for two reasons:

1. For most pairing-friendly elliptic curves used in practice, the pairing operation e (Section 1.3) is undefined when its input points are not in the prime-order subgroups of E1 and E2. The resulting behaviour is unpredictable and may enable forgeries.
2. Even if the pairing operation behaves properly on inputs that are outside the correct subgroups, skipping the subgroup check breaks the strong unforgeability property [ADR02].”

Therefore, as a defence against possible vulnerabilities that would enable forgeries, we believe the URSA library should include signature validation as specified in the BLS specification.

Recommendation

The URSA library should be modified to include signature validation within the operations using public keys as required by the BLS specification.

Feedback from initial Hyperledger URSA community review:

This will be raised as an issue and be subject to normal community workflow to address issues. The community agreed that the classification of this item as low risk was appropriate.

6. <u>Breaking unlinkability of Identity Mixer using malicious keys</u>

Severity	Low

Description

It is possible to mount an attack against CL-signature based Identity Mixer whereby a malicious issuer can break unlinkability of issuance and disclosure sessions of its credential. This can be achieved by generating a private key in such a way that the discrete log problem becomes partially solvable for the issuer, and then exploiting the fact that Schnorr proofs can be constructed for fractions to pass the key as properly generated. The combined attack demonstrates the importance of correctly proving preconditions of cryptographic systems, especially where these are nontrivial as is the case when using RSA groups.

Related Documentation/Library

Breaking unlinkability of Identity Mizer using malicious keys, D. Venhoek, S. Ringers. 22 Dec 2021

Analysis

This potential exploit requires a malicious Issuer. In most ecosystems, other trust anchors are in place to prevent the participation of such Issuers. This serves to make the likelihood of an exploit of this type very low. There is a model for mitigation that includes generation of a complete proof of proper generation of public keys. This method is computationally expensive and impractical in the context of the implementation in URSA.

Recommendation

Further research into more efficient complete proof generation should be undertaken.

Feedback from initial Hyperledger URSA community review:

This will be raised as an issue and be subject to normal community workflow to address issues. The community agreed that the classification of this item as low risk was appropriate. In the interim, exposure of the issue and advice on mitigation approaches outside the scope of URSA will be investigated.

Appendix A - About the Authors

The Digital Identity Laboratory of Canada (IDLab) is an independent Canadian non-profit entity dedicated to advancing digital trust by breaking down barriers to digital ID adoption. The IDLab promotes conformity and interoperability of user-centric digital identity solutions. The IDLab is not an incubator and does not develop or sell digital identity solutions. Our mission is to accelerate the adoption, development and knowledge of compliant and interoperable digital ID solutions.

We accomplish this mission by delivering education, assessment, advisory and sandbox services. When delivering those services, IDLab preserves its neutrality and independence by complying with its Policy on the Limitations of Commercial Activities.

Contributors to this report include:

- P. Roberge
- B. Daly
- N. Kettle
- M. Barker
- J. Gage
- L. Francis