ENECUUM CRYPTOGRAPHIC LIBRARY AUDIT

July 26, 2021

ENQ
ENECUUM

MixBytes()

TABLE OF CONTENTS

01 INTRODUCTION TO THE AUDIT

General Provisions

Enecuum is a blockchain mobile network for decentralized applications. It was created as a decentralized ecosystem to be able to bring the blockchain and cryptocurrencies to the real mainstream, involving a crowd with regular mobile and desktop devices into the blockchain network. Enecuum allows each smartphone owner to be a part of our global network.

MixBytes team (the Contractor) asked team's Lead Auditor Mikhail Komarov (the Auditor) to audit Enecuum Blockchain's (the Client) cryptography library.

Scope of audit

https://github.com/Enecuum/lib-crypto
Commit: 52f764fda3cbba7deef9bdc7e6a8938135ae0cec

02 SECURITY ASSESSMENT PRINCIPLES

Classification of Issues

- CRITICAL: Bugs and vulnerabilities that enable remote code execution attack (leads to making the sensitive data accesible by remote users) or local some kind of priveledge-escalation (paves the road to remote users to become local ones), segmentation failures or non-zero code termination within the usecases considered by the Client as most common or most important ones.

- MAJOR: Bugs and vulnerabilities that enable local private memory contents leakage, making sensitive data available to the same physical machine users.

- WARNINGS: Bugs that can break the intended algorithm logic or enable a DoS attack.

- COMMENTS: Various issues and recommendations.

Security Assessment Methodology

The audit was performed with double redundancy by two auditors.
Stages of the audit were as follows:

- Initial check. Used for the time required estimation.
- Manual check by the cryptography implementation practitioner.
- Manual check by the theoretic research practitioner.
- Mutual results check.
- Discussion and merge of independent audit results.
- Report execution.

Since the reviewed library consists of two parts - C++ library and NodeJS wrapper, these parts were reviewed separately and as a whole in the end.

03 DETECTED ISSUES

CRITICAL

1. Returning nullptr instead of class instance. Possible remote-executed SEGFAULT.

Description

This issue is about incorrect OpenSSL error core handling.

OpenSSL's `BN_*` -family functions return error codes as integers (https://linux.die.net/man/3/bn_add). For all functions, 1 is returned for success, 0 on error. The return value should always be checked (e.g., `if (!BN_add(r,a,b)) goto err;`).

The particular BigNumber class (BigNumber.h#L10) uses these functions as the way to perform multiprecision arithmetics operations (crypto.cpp#L43 , crypto.cpp#L26 , crypto.cpp#L19 , crypto.cpp#L12) . But in case one of `BN_*` -family function fails, error-handling case returns `nullptr` inside the class' `operator+ () / operator*() / operator-()` functions, instead of returning the instance of `BigNumber` class. This means every `operator+() / operator*() / operator-()` usage will result in further `BigNumber` class instance undefined behavior (for example here: crypto.cpp#L146 and here: crypto.cpp#L145). This often leads to SEGFAULT.

The situation got even worse, when we realized this wrapper (with it's class operators) is being commonly used among the C++ to NodeJS interface (addon.cc#L210 , addon.cc#L200).

This means using this wrapper from NodeJS (which interface is available to remote user) with some incorrect (e.g. too big or incorrectly formatted) number would result in error in one of `BN_*` -family functions, which would result in undefined behavior because of incorrect error code handling, which would result in SEGFAULT.

Considering NodeJS architecture supposes same code usage at both sides (backend and frontend), such a crash can result in not only crashing locally-executed client, but a remote-executed server.

Recommendation

The very particular recommendation about fixing this issue is to change error-handling return values in here: crypto.cpp#L46 , crypto.cpp#L29 , crypto.cpp#L20 , crypto.cpp#L13 , crypto.cpp#L77 to default-initialized BigNumber instance. The overall (fix once and for all) recommendation about this (and one more following major issue) is to replace `BigNumber` wrapper usage with plain OpenSSL's `BIGNUM` pointer usage.

Status

***Fixed** with error handlers append.*

MAJOR

1. Uncleaned private key memory.

Description

This vulnerability is quite a complex one.

BigNumber.cpp#L62
This particular line defines the commented-out OpenSSL's `BN_free(bn);` function commonly used for cleaning out memory used for the particular OpenSSL's multiprecision number. It is a crucial thing to leave as less crucial data in the RAM as possible to avoid side-channel or timing attacks. `BigNumber` class is a wrapper built ontop of OpenSSL's `BIGNUM` and it contains a pointer to the actual data-holding structure in here: BigNumber.h#L23. This pointer gets initialized with multiprecision number data storage (or copied) in several places (BigNumber.cpp#L13 , BigNumber.cpp#L17 , BigNumber.cpp#L8), but never gets cleaned out. This results in memory leakage, leaving the private key available in RAM even after the binary using the library being reviewed was terminated. This particular RAM page piece becomes available for allocation for every particular OS user or (in case the attacker really want to mess with this) every particular physical machine user.

Meanwhile such a wrapper class is being used among the library as well as in critical place for managing private keys.

crypto.cpp#L129
This particular line begins definition of a function intended to perform Shamir scheme keys generation. It's arguments include initial secret value to be shared with the scheme (`secretM`). This argument gets passed into the function by value. This means every content of the structure representing

this argument gets copied into the function argument stack. This includes the `BIGNUM` pointer defined in here: BigNumber.h#L23. That one, which never gets cleaned out.

Every usage of this wrapper (addon.cc#L20 , addon.cc#L24 , addon.cc#L31 , crypto.cpp#L27 , crypto.cpp#L44 , crypto.cpp#L70 , crypto.cpp#L78 , crypto.cpp#L85) leaves the uncleaned data in the RAM available to any willing user of the particular physical machine to retrieve it.

Recommendation

Since `BigNumber` wrapper is being used all over the library, the actual issue becomes quite a complex one. Every particular usage of this wrapper (in case is allocates more memory) in every function induces memory leak.

The attempt and intention of a developer to make things right (correctly deallocate and zeroize the memory `BIGNUM` pointer points to every time the `BigNumber` wrapper instance gets destroyed) was spotted in here: BigNumber.cpp#L62, but according to what the `BigNumber` wrapper is (architecturally speaking, it is a pointerholder), deallocating the memory every time BigNumber object gets destroyed would destroy all the other `BigNumber` wrapper instances functionality (several wrapper instances are using the same OpenSSL's `BIGNUM` pointer, pointing to the same memory piece). Such an architectural decision induced the necessity to track every particular `BIGNUM` pointer allocations and clean them out only in case there are no `BigNumber` wrappers using that. It is hard. So it was easier for the developer just to comment-out `BN_free` function and leave the data uncleaned.

The general recommendation about such a complex issue is not to fix every particular place `BIGNUM` pointer stays uncleaned, but to get rid of `BigNumber` wrapper and simply use the raw `BIGNUM` pointer. This would require library refactoring, but this is the right way to avoid multiple memory leaks.

Status

***Fixed* with duplicating `BIGNUM` instances every time `BigNumber` is copied (e.g. BigNumber.cpp#L10) along with freeing them every time an instance gets destructed.**

2. Potential temporary variable memory leak (suggestion provided by the Client).

Description

This issue is about potential memory leak containing sensitive data induced by the `buff` temporary variable within the `keyRecovery` function in `crypto.cpp` .

According to the client's suggestion it leads to sensitive-contents memory leak with `buff` variable never being properly freed and zeroized.

Recommendation

Contractor agress that this particular memory leak was present, and, no matter its exploitaton would be very hard, it should be elliminated.
The fix suggested by the client (introducing `EC_POINT_free(buff)`) is correct.

Status

Fixed

WARNINGS

1. Redundant (or potentially dangerous) random number generator seeding.

Description

This warning is about seeding the unsecure random (luckily redundant for now) with the same value every time.

The particular seeding is being performed in here: crypto.cpp#L117 . `srand()` function seeds the embedded libc unsecure (https://linux.die.net/man/3/srand) random generator which is never should be used for cryptographically secure random numbers generation.

This particular line, it seems, was used for seeding random number generator for Shamir scheme polynomial coefficients generation.

Luckily, the developer moved to the OpenSSL's random generator implementation (`BN_rand_range`) in here: crypto.cpp#L218. So, the problem was mostly solved.

Recommendation

Since this issue is mostly a warning, we recommend the developer pay more attention to random generation across the C++ library, recheck every usage case, and remove redundant random seed generator to remove any newcoming reviewer doubts about the developer understanding.

Status

Fixed ***with additional random generation correctness checks implemented (e.g. crypto.cpp#L333).***

2. Wrong subgroup order.

Description

This warning is about G1 group suborder being hardcoded with one bit missing. The particular group suborder is being hardcoded in here: crypto.cpp#L270 as binary string "011011". Actual group suborder equals "91" which is "1011011" in binary.

Recommendation

This particular issue gets handled in here: crypto.cpp#L248, but the group suborder is still being defined wrong.

Status

Fixed ***in crypto.cpp#L410***

3. Redundant self-assignment.

Description

This warning is about unnecessary assignment of the class instance object to itself in here: ellipticCurve.cpp#L261.
We see no real necessity for this line.

Recommendation

To remove this redundant assignment?

Status

Fixed ***with the redundant assignment removed.***

4. Elliptic curve wrapper generator element memory leak.

Description

This one is a minor one in here: Curve.h#L22.

Pointer to the `EC_POINT` gets allocated, but never gets released properly. This induces memory leak.

Recommendation

This particular issue does not lead to any particular damage, but it increases memory consumption and leaves it polluted. Deallocation with `EC_POINT_free(G)` in `Curve` class destructor could make things better.

Status

***Fixed** with Curve.cpp#L61*

5. Elliptic Curve Data Memory Leak.

Description

This one is pretty much similar to the issue about non-deallocated `BIGNUM` pointer in `BigNumber` wrapper. Mostly an architectural one in here: Curve.h#L21. But, in spite of the issue with non-deallocated `BIGNUM` pointer, this one is pretty much harmless. Elliptic curve params are not that sensitive in terms of security to be concerned about them being left in the RAM. But, still, this is a memory leak. Pointer to the `EC_GROUP` gets allocated, but never gets released properly.

Recommendation

This particular issue does not lead to any particular damage, but it increases memory consumption and leaves it polluted. Deallocation with `EC_GROUP_free(curve)` in `Curve` class destructor could make things better.

Status

***Fixed** with Curve.cpp#L62*

6. String output memory leak (suggestion provided by the Client).

Description

This warning is about potential memory leak induced by `BigNumber` class string output formatting functions `toHexString` , `toDecString` and by decimal conversion function `decimal` in `BigNumber.cpp` .

According to the client's suggestion it leads to the memory leak (according to these functions' purpose, preparing contents for the output cannot lead to unintentional private value reveal), so this has to be handled with `OPENSSL_free` call.

Recommendation

Contractor agress that the less memory leaks are present - the better it is, but for these particular cases using regular `free` from `stdlib.h` with `char *` -typed arguments would be enough.

Status

Fixed

7. Tests memory leak (suggestion provided by the Client).

Description

This warning is about potential memory leak induced by the absence of `free` or `OPENSSL_free` function calls for variables `ec` , `proj` , `secret` in file `main.cpp` . According to the client's suggestion it leads to non-critical memory leak (according to this file purpose of being a test one).

Recommendation

Contractor agress that the memory leak was present and the suggested fix (with applying `EC_POINT_free(secret)` and `delete ec; ec = nullptr;`) is correct.

Status

Fixed

8. Low-memory and high-loaded environments potential memory allocation issue (suggestion provided by the Client).

Description

This warning is about potentially incorrect memory allocation done within the `operator%` and other functions in `crypto.cpp` . In case the execution is being performed within extremely low free RAM environments (less than `sizeof(int)` bytes) or within extremely high-loaded environments, the absence of explicit `handleError(NO_MEMORY)` checks after `BN_CTX_new()` and `BN_new()` were called could lead to the crash.

According to the client's suggestion it could lead to the execution failure in case the environment's free virtual memory was exausted.

Recommendation

Contractor agress that the issue is present in case the execution is being done within extremely restricted environments. But, since execution within the environment restricted that much is not possible (according to massive heap allocations are present, so embedded environments execution is not practically possible), this issue is classified as a "Warning of an extremely low probability".

The fix suggested by the client, though, is correct.

Status

Fixed

COMMENT

1. Passing function arguments by value.

Description

Most of C++ library part is implemented with `std::string` and `std::vector` -typed function arguments being passed by value. This has a performance and security-related influence because every argument passed by value is being copied into the function argument stack, which populates sensitive data copies in the RAM.

Recommendation

The C++ way of passing arguments without populating data copies is to pass them by const reference. So to avoid populating sensitive data copies in RAM we recommend using this way.

Status

***Fixed** with l-value references and pointers usage (e.g. in here: BigNumber.cpp#L32).*

2. Comments revealing sensitive data.

Description

This comment is about `index.js` , which explicitly logs literally everything happening. Including secret keys. Is it really required in production?

Status

***Not an issue** (just comments anyway)*

3. Function arguments const specifier (suggestion provided by the Client).

Description

This comment is about potential further development issues related to the absence of `const` specifier within the function arguments. According to the client suggestion it is supposed to protect from unintentional pointer management issues.

Status

Contractor agrees on this might help to avoid the unintentional developer-issued pointer modifications, but such a suggestion however cannot be qualified in any other way than a "Comment".

4. Explicit pointer null-ification (suggestion provided by the Client).

Description

This comment is about necessity to explicitly set all the pointer-alike variables to `nullptr` (or `NULL`) to maintain `nullptr` comparisons correct (using those as unintialized value comparisons).

According to the client suggestion it is supposed to protect from memory management issues (like calling `OPENSSL_free` with an already free-d pointer as an argument).

Status

Contractor agrees on this might help with developer-induced memory mismanagement. However such a suggestion cannot be qualified in any other way than a "Comment".

04 CONCLUSION AND RESULTS

The system reviewed contains two parts: C++ library and NodeJS wrapper. They were reviewed separately and as a whole.

The security and reliability of the C++ part was rated "High". One critical and one major flaws were found and fixed. Several warnings were given about particular pieces of code and architecture decisions. All of the warnings given were fixed.

The security and reliability of the NodeJS part (excluding NAPI wrapper - it is related to C++ part) was rated "High". No major flaws were spotted.

Findings list:

Level	Amount
CRITICAL	1
MAJOR	2
WARNING	8
COMMENT	4

About MixBytes

MixBytes is a team of blockchain developers, auditors and analysts keen on decentralized systems. We build open-source solutions, smart contracts and blockchain protocols, perform security audits, work on benchmarking and software testing solutions, do research and tech consultancy.

Contacts

https://github.com/mixbytes/audits_public

https://mixbytes.io/

hello@mixbytes.io

https://t.me/MixBytes

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only. The information presented in this report is confidential and privileged. If you are reading this report, you agree to keep it confidential, not to copy, disclose or disseminate without the agreement of Aave. If you are not the intended recipient(s) of this document, please note that any disclosure, copying or dissemination of its content is strictly forbidden.