




**Security Review – Phase 1
for Wire Swiss GmbH**

Final Report

2017-02-08

FOR PUBLIC RELEASE



<i>Revision</i>	<i>Date</i>	<i>Change</i>
1	2016-11-24	Start of review
2	2016-11-28	Initial report creation
3	2017-01-04	Final findings added
4	2017-01-09	Delivery to Wire
5	2017-02-08	Public version (formatting, severity ratings)



Contents

1	Executive Summary	4
2	Introduction	5
2.1	Findings overview	5
2.2	Scope	5
2.3	Methodology	6
3	Findings in Proteus (Rust)	7
3.1	WIRE-P1-000: DH accepts degenerate keys	7
3.2	WIRE-P1-001: Invalid public keys undetected	9
3.3	WIRE-P1-002: Thread-unsafety risk	11
3.4	WIRE-P1-003: Secret values not zeroized	14
3.5	Differences with the specifications	15
3.6	Possible improvements	17
4	Findings in Proteus (CoffeeScript)	18

4.1	WIRE-P1-004: DH accepts degenerate keys	18
4.2	WIRE-P1-005: Invalid public keys undetected	19
4.3	WIRE-P1-006: Secret values not zeroized	20
4.4	WIRE-P1-007: Thread-unsafety risk	21
5	Findings in Cryptobox and CBOR	22
5.1	WIRE-P1-008: Strings NULL termination	23
5.2	WIRE-P1-009: Null pointers check	23
5.3	WIRE-P1-010: Prekey length value	24
5.4	WIRE-P1-011: Cryptobox-c tests buffer overread	24
5.5	WIRE-P1-012: CBOR arithmetic overflow	25
5.6	WIRE-P1-013: CBOR Lax parsing of serialized data	27
6	About	29



1 Executive Summary

This security assessment focused on Wire's core cryptographic components: the Proteus protocol, and the Cryptobox API built over Proteus.

The components reviewed were found to have a high security, thanks to state-of-the-art cryptographic protocols and algorithms, and software engineering practices mitigating the risk of software bugs. Issues were nonetheless found, with some of them potentially leading to a degraded security level. None of the issues found is critical in terms of security.

We for example found that invalid public keys could be transmitted and processed without raising an error. As a consequence, the shared secret negotiated by communicating parties becomes predictable, which in turns weakens security guarantees in terms of "break-in recovery". The root cause of this issue is a bug in a third-party component (neglect to verify an error code).

We recommend that this issue be fixed, and that other security improvements be implemented to address thread-unsafety risks, sensitive data in memory, and other aspects as described in this report.

The work was performed between November 23rd, 2016 and January 9th, 2017, by Jean-Philippe Aumasson (Kudelski Security) and Markus Vervier (X41 D-Sec GmbH). A total of 14 person-days were spent, with most of the time spent reviewing code and investigating potential security issues. Wire provided the source code of all components to be reviewed, including approximately 3300 lines of Rust, 2400 lines of CoffeeScript, and 400 lines of C.



2 Introduction

2.1 FINDINGS OVERVIEW

DESCRIPTION	ID	SEVERITY	SECTION
Insecure DH ratchet keys (Rust)	WIRE-P1-000	LOW	3.1
Invalid public keys undetected	WIRE-P1-001	LOW	3.2
Thread-unsafety risk	WIRE-P1-002	MEDIUM	3.3
Secret values not zeroized	WIRE-P1-003	LOW	3.4
Insecure DH ratchet keys (JS)	WIRE-P1-004	LOW	4.1
Invalid public keys undetected	WIRE-P1-005	LOW	4.2
Secret values not zeroized	WIRE-P1-006	LOW	4.3
Thread-unsafety risk	WIRE-P1-007	MEDIUM	4.4
Strings NULL termination	WIRE-P1-008	MEDIUM	5.1
Null pointers check	WIRE-P1-009	LOW	5.2
Prekey length value	WIRE-P1-010	LOW	5.3
Cryptobox-c tests buffer overread	WIRE-P1-011	MEDIUM	5.4
CBOR arithmetic overflow in decoding	WIRE-P1-012	LOW	5.5
CBOR Lax parsing of serialized data	WIRE-P1-013	MEDIUM	5.6

Table 2.1: Security relevant findings.

2.2 SCOPE

We reviewed the Proteus messaging protocol as implemented in the `proteus` repository, as of November 28th in version `6b317191` (we consider projects as hosted on `https://github.com/wireapp`). We performed a mostly manual review of that code, as well as of its dependencies created by Wire (`cbor-codec` and `hkdf`).

Core cryptographic components were provided by the Sodium library via wrappers at <https://github.com/dnag/sodiumoxide>. We did not perform a comprehensive review of Sodium, but only of certain components critical to Proteus' security.

Third-party references used for the review include <https://whispersystems.org/docs/> (Revision 1 versions), as well as the report of a previous audit, as provided by Wire.

We also reviewed the Cryptobox API (`cryptobox` repository) as well as its C wrapper (`cryptobox-c`). Cryptobox defines a simple, high-level API to Proteus in order to hide the protocol's complexity to callers in Wire applications.

Finally, we reviewed the CoffeeScript counterparts of Proteus and cryptobox, as implemented in the `proteus.js` and `cryptobox.js`.

2.3 METHODOLOGY

We sought security issues in the Proteus protocol implementations both in terms of functionality (does it behave as intended, and in particular as specified?), of software security (are there software bugs exploitable by an attacker?), and of usage (will the Wire applications use Proteus securely via the cryptobox API?).

All discovered issues were classified using Common Weakness Enumeration (CWE)¹. Due to the nature of reviewing library components and protocols the usage of the Common Vulnerability Scoring System (CVSS)² was not applicable. Therefore all issues were rated using a severity rating of *low*, *medium*, *high* or *critical* based on the possible security impact in common use cases and secure coding and design best practices.

¹<https://cwe.mitre.org>

²<https://www.first.org/cvss>



3 Findings in Proteus (Rust)

As discussed with Wire, the following issues are already known and the associated risk is accepted:

- The version byte in a message envelope is not authenticated. This does no harm in the current version, since the `version` byte is not interpreted by the receiver.
- Prekeys are not signed, which leaves the protocol vulnerable to attackers that 1) serve forged prekeys and 2) later compromise the identity key belonging to the supposed creator of the prekeys.

In the following, we present security issues discovered during the review. We conclude by discussing minor differences with the documented protocols, and by proposing improvements to the protocol and code. Well-known inherent properties of the protocol such as unknown key-share attacks will not be discussed.

3.1 WIRE-P1-000: DH ACCEPTS DEGENERATE KEYS

Severity: **LOW**

CWE: 358

If an all-zero ratchet public key is received, then the resulting shared secret will be zero regardless of the value of the secret key—whereas such a degenerate key should be rejected as invalid. Therefore, if Bob sends all-zero ratchet public keys, subsequent

message keys will only depend on the root key and not on Alice's ephemeral private keys. A dishonest peer may therefore keep sending degenerate keys in order to reduce break-in recovery guarantees, or force all sessions initiated by a third party to use a same message key, while a network attacker may force the first message keys to a public constant value, for example.

The root cause is that sodiumoxide's Rust bindings to libsodium do not check the return value of `crypto_scalarmult_curve25519`, whereas the corresponding C function does check whether it computes an all-zero shared secret (indicating a degenerate, insecure behavior) and returns a non-zero value in this case.

In the proof-of-concept program below, a shared secret is computed using a random private key `sk` and an all zero public key `pk`:

```
extern crate sodiumoxide;

fn main() {
    sodiumoxide::init();
    let pk = [0u8; 32];
    let mut sk = [0u8; 32];
    sodiumoxide::randombytes::randombytes_into(&mut sk);
    let p = sodiumoxide::crypto::scalarmult::GroupElement(pk);
    let s = sodiumoxide::crypto::scalarmult::Scalar(sk);
    let h = sodiumoxide::crypto::scalarmult::scalarmult(&s, &p);
    println!("{:?}", sk)
    println!("{:?}", h)
}
```

When executed, this program will always print the same all-zero shared secret value, whatever the value of the private key:

```
$ ./target/debug/poc_zerokey
[25, 128, 231, 205, 226, 155, 42, 69, 58, 63, 117, 20, 231, 1, 8, 62,
20, 233, 74, 113, 51, 253, 162, 201, 120, 178, 63, 120, 155, 25, 17,
141]
GroupElement([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])  
$ ./target/debug/poc_zerokey  
[39, 124, 213, 115, 23, 77, 104, 78, 102, 188, 91, 233, 242, 114, 63,  
173, 111, 182, 177, 122, 248, 40, 255, 132, 194, 121, 41, 168, 10, 97,  
255, 147]  
GroupElement([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

To fix this issue, Proteus must check the value of the shared secret and throw an error if the all-zero value is found. We believe that the issue should also be reported to, and fixed by sodiumoxide (in which case the above check may be replaced by a check of sodiumoxide function's return value).

Note that the same behavior occurs if the public key is 010000...00, that is, the result of converting an all-zero Ed25519 key to Curve25519 format.

3.2 WIRE-P1-001: INVALID PUBLIC KEYS UNDETECTED

Severity: **LOW**

CWE: 325

When `PreKeyBundle::deserialise()` is called when processing a `PreKeyBundle`, if an invalid public key is included (that is, a point not on the curve), be it as prekey public key or identity, then the deserialized public key received will be the all-zero point.

An attacker may therefore serve prekey bundles with invalid public keys, which will lead to a triple Diffie-Hellman handshake using all-zero public-keys in place of Bob's prekey and identity key. As a result, the handshake's result will always be the same value, and it will be independent from Alice's private keys, by exploiting the issue in §3.1. This may in turn lead to predictable message keys.

The root cause is that when `libsodium's ffi::crypto_sign_ed25519_pk_to_curve25519` receives an invalid Ed25519 public key, it leaves the receiving buffer unchanged and

returns -1, as the following code will demonstrate for the invalid key `ffffff...ff00`:

```
// gcc -lsodium poc_keyconversion.c

#include <sodium.h>
#include <string.h>

#define KEYLEN 32

int main() {
    unsigned char cpk[KEYLEN];
    unsigned char epk[KEYLEN];
    int ret, i;

    memset(epk, 0xff, KEYLEN);
    epk[31] = 0x00;
    memset(cpk, 0xff, KEYLEN);

    ret = crypto_sign_ed25519_pk_to_curve25519(cpk, epk);
    printf("%d\n", ret);

    for(i=0; i<KEYLEN; ++i)
        printf("%02x", cpk[i]);
    printf("\n");

    return 0;
}
```

In Proteus' `from_ed25519_pk()` however, called by `PublicKey::decode()`, the return value is not checked and therefore the all-zero array `ep` remains all-zero when an invalid `k` is received:

```
pub fn from_ed25519_pk(k: &sign::PublicKey) -> [u8; ecdh::GROUPELEMENTBYTES] {
    let mut ep = [0u8; ecdh::GROUPELEMENTBYTES];
    unsafe {
        ffi::crypto_sign_ed25519_pk_to_curve25519(ep.as_mut_ptr(), (&k.0).as_ptr());
    }
    ep
}
```

The call path to the vulnerable code is `PreKeyBundle::deserialise() → PreKeyBundle::decode() → PublicKey::decode() → from_ed25519_pk → ffi::crypto_sign_ed25519_pk_to_curve25519`.

`PreKeyBundle::deserialise()` is called in cryptobox' `session_from_prekey()` and in cryptobox-c's `cbox_is_prekey()`.

A solution is to verify the return value of `crypto_sign_ed25519_pk_to_curve25519` and refuse to proceed if an invalid key is received.

3.3 WIRE-P1-002: THREAD-UNSAFETY RISK

Severity: **MEDIUM**

CWE: 252

Because Proteus does not check correct initialization of libsodium, it may be affected by an incorrect initialization.

The sodiumoxide Rust wrapper over libsodium requires proper initialization in order to behave securely, as noted in the file `sodiumoxide-0.0.12/src/randombytes.rs`, used by Proteus:

```
/// 'randombytes()' randomly generates size bytes of data.  
///  
/// THREAD SAFETY: 'randombytes()' is thread-safe provided that you have  
/// called 'sodiumoxide::init()' once before using any other function  
/// from sodiumoxide.  
pub fn randombytes(size: usize) -> Vec {  
...  
}
```

The said `sodiumoxide::init()` function is called in Proteus, but its success not checked: Proteus' 'lib.rs' indeed includes

```
pub fn init() {
    sodiumoxide::init();
}
```

In contrast, sodiumoxide's `lib.rs` correctly includes the check for success:

```
/// 'init()' initializes the sodium library and chooses faster versions of
/// the primitives if possible. 'init()' also makes the random number generation
/// functions ('gen_key', 'gen_keypair', 'gen_nonce', 'randombytes', 'randombytes_into')
/// thread-safe
///
/// 'init()' returns 'false' if initialization failed.
pub fn init() -> bool {
    unsafe {
        ffi::sodium_init() != -1
    }
}
```

Here `sodium_init()` is the FFI-imported C function from Sodium, defined as follows in `libsodium/src/libsodium/sodium/core.c`:

```
int
sodium_init(void)
{
    if (sodium_crit_enter() != 0) {
        return -1;
    }
    if (initialized != 0) {
        if (sodium_crit_leave() != 0) {
            return -1;
        }
        return 1;
    }
    _sodium_runtime_get_cpu_features();
    randombytes_stir();
    _sodium_alloc_init();
    _crypto_pwhash_argon2i_pick_best_implementation();
    _crypto_generichash_blake2b_pick_best_implementation();
}
```

```
_crypto_onetimeauth_poly1305_pick_best_implementation();
_crypto_scalarmult_curve25519_pick_best_implementation();
_crypto_stream_chacha20_pick_best_implementation();
initialized = 1;
if (sodium_crit_leave() != 0) {
    return -1;
}
return 0;
}
```

`sodium_init()` will fail when lock or unlock of the `_sodium_lock` mutex fails. This may happen for example if calling `sodium_init()` from two concurrent threads.

Consequences of a misinitialized context can be the following:

- Sodium may not pick the best implementations of crypto algorithms (it will default to the reference ones).
- The pseudorandom number generator (PRNG) may not be initialized correctly. But this should only be a problem when a custom PRNG is used, as opposed to the default one (such as `/dev/urandom`).
- The heap canary will not be random, and therefore becomes less useful to mitigate memory corruption vulnerabilities.

Fixing the issue just consists in recording the return value of `sodiumoxide::init()` in Proteus' `init`, as follows (note the absence of semicolon):

```
pub fn init() -> bool {
    sodiumoxide::init()
}
```

3.4 WIRE-P1-003: SECRET VALUES NOT ZEROIZED

Severity: **LOW**

CWE: 316

To avoid leaking secrets through core dumps or memory reuse by other processes, it's recommended to zeroize the memory holding secret values after usage. For this, sodiumoxide uses Rust's `Drop` trait in order to zeroize keys when they go out of scope. This is implemented in the `new_type` macro of sodiumoxide:

```
macro_rules! new_type {
    ( #[meta:meta] )*
    secret name : ident(bytes : expr);
    ) => (
        #[meta] *
        #[must_use]
        pub struct name (pub [u8; bytes]);
        newtype_clone!(name);
        newtype_traits!(name, bytes);
        impl name {
            newtype_from_slice!(name, bytes);
        }
        impl Drop for name {
            fn drop(&mut self) {
                use utils::memzero;
                let &mut name (ref mut v) = self;
                memzero(v);
            }
        }
    );
}
(...)
```

Proteus relies on this zeroized type for the types `CipherKey`, `MacKey`, `SecretKey` as used to hold chain keys, message keys, ratchet keys' secret, and so on.

However, when Proteus' `kdf` calls `hkdf : : hkdf ()`, the result is a `Vec<u8>` type (`okm`), which is not zeroized after usage.

To fix this, `okm` may be directly zeroized in `hkdf : :hkdf()`. The output of `hkdf : :hkdf()` may also be redefined as (say) a `stream : :Key` type in order to benefit from the `Drop` zeroizing method.

For an stronger protection of secret values, the following library may be used: <https://github.com/stouset/secrets>.

3.5 DIFFERENCES WITH THE SPECIFICATIONS

We highlight minor discrepancies between Proteus and the documented x3DH and double-ratchet protocols:

3.5.1 Chain key generation

Instead of using only a key derivation function (KDF) as the double-ratchet documentation recommends, Proteus uses

1. A MAC (HMAC-SHA-256) to generate a new chain key from a previous chain key,
2. A KDF (HKDF...) to generate message keys from a chain key.

This is acceptable because HMAC-SHA-256 behaves as a pseudorandom function (PRF), and therefore gives keys of similar strengths as if they were generated by a proper KDF.

3.5.2 Session initialization

The double-ratchet specification summarizes the session states initialization as follow:

```
def RatchetInitAlice(state, SK, bob_dh_public_key):
    state.DHs = GENERATE_DH()
    state.DHr = bob_dh_public_key
```



```
state.RK, state.CKs = KDF_RK(SK, DH(state.DHs, state.DHr))
state.CKr = None
state.Ns = 0
state.Nr = 0
state.PN = 0
state.MKSKIPPED = {}

def RatchetInitBob(state, SK, bob_dh_key_pair):
    state.DHs = bob_dh_key_pair
    state.DHr = None
    state.RK = SK
    state.CKs = None
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}
```

and the document notes: “This assumes Alice begins sending messages first, and Bob doesn’t send messages until he has received one of Alice’s messages. To allow Bob to send messages immediately after initialization Bob’s sending chain key and Alice’s receiving chain key could be initialized to a shared secret. For the sake of simplicity we won’t consider this further.”

Proteus uses such a trick and initializes Alice’s receiving chain key (that is, Bob’s sending chain key) to a key derived from the initial shared secret (an extension of (SK) using the OWS notations, and `dsecs.mac_key` in `SessionState` initializers.

This chain key is derived from the triple DH key agreement but is otherwise unrelated to the value used as a root key. According to our analysis, this construction is sound and secure.

3.6 POSSIBLE IMPROVEMENTS

3.6.1 Use an authenticated cipher instead of a cipher and a MAC

Currently encryption and MAC authentication are realized with two separate algorithms (and thus using two distinct keys): ChaCha and HMAC-SHA-256. Using a single authenticated cipher would simplify things (a single key instead of two keys, a single call to the crypto library, etc.) and provide a slight speed-up (one pass over the data, less computation).

For this, sodiumoxide provides the `xsa1sa20poly1305` “secret box” function.

3.6.2 Counters randomization

Chain key counters are assumed to be non-secret values, however they could leak the number of messages exchanged with a given party, since they are initialized to zero. To avoid leaking the number of messages, the counters may be initialized to a random value and incremented by ensuring that they’ll wrap around the limits of the `u32` type.

3.6.3 Code warnings

The static analyzer `rust-clippy` (<https://github.com/Manishearth/rust-clippy>) reports a number of harmless warnings for Proteus (absence of `Default` implementations, redundant closure, clone on a `Copy` type, needless borrows, etc.).



4 Findings in Proteus (CoffeeScript)

After reviewing the Rust implementation of Proteus, we reviewed its CoffeeScript counterpart. We first checked if issues found in the Rust implementations occurred in the CoffeeScript version as well, and then looked for specific issues.

Possible improvements to the protocol proposed in §§3.6.2 and §§3.6.1 apply as well to the CoffeeScript implementation.

4.1 WIRE-P1-004: DH ACCEPTS DEGENERATE KEYS

Severity: **LOW**

CWE: 358

This is a similar issue as the one reported for the Rust version in §3.1: public keys leading to an all-zero shared secret will be accepted as valid public keys and will be used within the Diffie-Hellman operation `SecretKey.shared_secret`, defined as follows in `SecretKey.coffee`:

```
shared_secret: (public_key) ->
  TypeUtil.assert_is_instance PublicKey, public_key

  return sodium.crypto_scalarmult @sec_curve, public_key.pub_curve
```

Neither this function nor its callers verify that the DH result is zero.

We can verify the issue as follows:

```
> b.public_key.pub_curve
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0]
> a.secret_key.shared_secret(b.public_key)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0]
```

We recommend to check that a public key is not the point at infinity at the earliest stage, namely when deserializing the point received.

4.2 WIRE-P1-005: INVALID PUBLIC KEYS UNDETECTED

Severity: **LOW**

CWE: 325

This is a similar issue as the one reported for the Rust version in §3.2, but with different implications: the following function from `ed2curve` does not check the validity of Ed25519 public keys, therefore `PreKeyBundle.deserialize()` will accept invalid keys:

```
function convertPublicKey(pk) {
  var z = new Uint8Array(32),
      y = gf(), a = gf(), b = gf();

  unpack25519(y, pk);

  A(a, gf1, y);
  Z(b, gf1, y);
  inv25519(b, b);
  M(a, a, b);
```

```
    pack25519(z, a);  
    return z;  
}
```

Unlike the Rust bug in §3.2, however, the result will be an incorrect Curve25519 key rather than the all-zero point, which reduces the security impact.

This issue has been reported to the maintainer of ed2curve, who fixed it in <https://github.com/dchest/ed2curve-js/releases/tag/v0.2.0>. A solution is therefore to update to the latest version of ed2curve.

4.3 WIRE-P1-006: SECRET VALUES NOT ZEROIZED

Severity: **LOW**

CWE: 316

Secret values that run out of scope will have their memory freed by the garbage collector, however the values will remain in memory until overwritten. Objects such as `ChainKey`, `SecretKey`, `MessageKey` may therefore be exposed to other processes or through a core dump.

Whereas the Rust implementation zeroizes most secret values via sodiumoxide's use of the `Drop` trick, the Coffeescript doesn't erase the secret values before releasing the memory. The `memzero()` function from `libsodium.js` can be used in order to wipe an `Uint8Array` of its secret values.

4.4 WIRE-P1-007: THREAD-UNSAFETY RISK

Severity: **MEDIUM**

CWE: 252

This is a similar issue as the one reported in §3.3: when libsodium is initialized in proteus.js (`libsodium._sodium_init();`), the return value is not checked.

Proteus should therefore verify that the said return value is zero, and return an error otherwise.



5 Findings in Cryptobox and CBOR

We reviewed the Cryptobox API, as used to expose the Proteus functionality to Wire applications. Since cryptobox does not add logical complexity, but instead simplifies Proteus through a thin abstraction layer, it has a limited attack surface and does not appear as a major source of bugs.

Additionally we reviewed the CBOR codec (from <https://gitlab.com/twittner/cbor-codec>) using automated fuzz-testing and manual inspection. CBOR is one of the core components of the Wire stack, and is used by the Proteus implementation.

The Cryptobox C API (`cryptobox-c` repository) over the Rust cryptobox code as well as the Rust implementation of CBOR could be made safer by fixing the following minor issues:

5.1 WIRE-P1-008: STRINGS NULL TERMINATION

Severity: **MEDIUM**

CWE: 170

If character arrays are not NULL-terminated, the Rust code may convert data out of the original bounds to a string. For example, `cbox_session_init_from_message()` takes a `char const* sid` passed to the `to_str()` function from `lib.rs`, which calls the unsafe `Cstr::from_ptr()` on the pointer received.

Since `Cstr::from_ptr()` will recalculate the string length, looking for a null character, the program will read outside the original buffer if no such null character is found as expected.

A malicious caller may then provide malformed strings that will crash the library or even try to avoid a crash and incorporate data into the session id that lies behind the original buffer in memory. For example, an attacker may modify the initialization message sent from Alice to Bob so that Bob receives a malformed string that will crash his system (we haven't verified such exploitability, however).

Depending on the context an attacker might be able to subsequently exploit this condition similarly to a use after free.

To improve safe usage of the exposed API and functions, it is recommended to change the API in order to avoid unintentional unsafe usage.

5.2 WIRE-P1-009: NULL POINTERS CHECK

Severity: **LOW**

CWE: 476

Pointers are not checked for the NULL value. For example, `cbox_encrypt()` will segfault

if any of the three pointers received is null; `cbox_session_init_from_message()` with a null SID will segfault because of the unsafe `CStr::from_ptr()`; and so on.

This makes the library less resilient against misuse by caller applications, and may result in unsafe behavior.

The issue can be fixed by having a library exposing C functions that first check the pointers and then calls the functions implemented in Rust.

5.3 WIRE-P1-010: PREKEY LENGTH VALUE

Severity: **LOW**

CWE: 20

The value of `c_prekey_len` in `cbox_session_init_from_prekey()` is not checked. In the usage in the `test_basic()` test, CBOR will return an error if `c_prekey_len` is 82 or less, but won't return an error for any other size provided (even `0xffffffffffffffff`).

It would be safer to verify that the length receives belongs to the range of authorized values.

5.4 WIRE-P1-011: CRYPTOBOX-C TESTS BUFFER OVERREAD

Severity: **MEDIUM**

CWE: 125

The test suite of `cryptobox-c` includes a minor buffer overread, when comparing a plain-text and the corresponding ciphertext:

```
assert(strncmp((char const *) hello_bob, (char const *) cbox_vec_data(cipher), \
              cbox_vec_len(cipher)) != 0);
```

Here the `assert()` aims to check that the ciphertext does differ from the plaintext. But it assumes that the plaintext has the same length as the ciphertext, namely `cbox_vec_len(cipher)`, which is wrong. For example, in `test_basics()` the plaintext is 11-byte long whereas the ciphertext is 197-byte.

5.5 WIRE-P1-012: CBOR ARITHMETIC OVERFLOW

Severity: **LOW**

CWE: 190

The following issue relates to the decoding of data serialized using `cbor-codec`: when values are skipped, the length field of an object might be too large and cause a length calculation to wrap around.

The method `cbor::Decoder::skip()` in turn calls the method `cbor::Decoder::skip_value()`:

```
pub fn skip(&mut self) -> DecodeResult<> {
    let start = self.config.max_nesting;
    self.skip_value(start).and(Ok(()))
}

fn skip_value(&mut self, level: usize) -> DecodeResult<bool> {
```

Inside this method the serialized fields are skipped over. In case of a field of type `Object`, the amount to skip over is calculated like this:

```
(Type::Object, a) => {
    let n = 2 * try!(self.kernel.unsigned(a));
```

```
    for _ in 0 .. n {
        try!(self.skip_value(level - 1));
    }
    Ok(true)
}
```

The calculation `let n = 2 * try!(self.kernel.unsigned(a))` might cause an arithmetic overflow. This was discovered during fuzz testing where invalid object size values led to a panic (in debug mode):

```
test_basics ... thread '<unnamed>' panicked at 'attempt to multiply with
overflow',
.cargo/registry/src/github.com-1ecc6299db9ec823/cbor-codec-0.6.0/src/decoder.rs:890
```

While in debug mode rust will panic due to arithmetic overflows, this check is omitted in release mode compilations.

An attacker might cause a panic if the code was compiled in debug mode. If it was compiled in release mode this behaviour might lead to different results in different parsers. When the overflow occurs a very large value might be truncated to a very small value. Since the return value of `self.kernel.unsigned(a)` is of type `U64` this would mean a value of $(\text{maximum size } U64 / 2) + 1$ would result in a size of zero after multiplication. So the current implementation would not discard such an insanely large value but continue parsing, whereas another parser would discard it. Therefore the decryption result would be different which could be exploited in the right context.

It is recommended to define a safe maximum length value for an object. Additionally all arithmetic operations not expected to overflow should use the checked rust arithmetic operations such as `checked_mul`¹ in this case.

It is recommended to define a safe maximum length value for an object. Additionally all arithmetic operations not expected to overflow should use the checked rust arithmetic operations such as `checked_mul`² in this case.

¹https://doc.rust-lang.org/std/primitive.i32.html#method.checked_mul

²https://doc.rust-lang.org/std/primitive.i32.html#method.checked_mul

5.6 WIRE-P1-013: CBOR LAX PARSING OF SERIALIZED DATA

Severity: **MEDIUM**

CWE: 707

The parsing and handling of values that are deserialized and decoded using CBOR may lead to unintended behaviour depending on the context it is used in.

When decoding messages and other data structures in Proteus, a field that occurs multiple times will be accepted as valid (recording only the latest entry) and fields with unknown tags are often ignored.

This was for example observed in the following code from `src/internal/message.rs`:

```
fn decode<'s, R: Read + Skip>(d: &mut Decoder<R>) -> DecodeResult<PreKeyMessage<'s>> {
    let n = try!(d.object());
    let mut prekey_id    = None;
    let mut base_key    = None;
    let mut identity_key = None;
    let mut message     = None;
    for _ in 0 .. n {
        match try!(d.u8()) {
            0 => prekey_id    = Some(try!(PreKeyId::decode(d))),
            1 => base_key    = Some(try!(PublicKey::decode(d))),
            2 => identity_key = Some(try!(IdentityKey::decode(d))),
            3 => message     = Some(try!(CipherMessage::decode(d))),
            _ => try!(d.skip())
        }
    }
}
```

While this does not result in any immediate vulnerability, lax parsing might lead to unforeseen consequences in situations where other parsing implementation might be used.

For example in the code shown above, a field might occur multiple times (e.g., the field `identity_key`). While in this implementation only the last decoded `identity_key` field

will be the one used, other implementations might for example use the first one. In cryptographic contexts this has been used³ to bypass signatures in the past.

Similar behavior was observed in multiple locations in the Proteus library as well as in the Cryptobox rust implementation.

It is recommended to enforce strict parsing and disallow multiple fields with the same tag. Processing should be terminated in case unknown tag values are encountered. Since the behavior was observed in multiple locations, all code that decodes serialized data should be checked.

³<http://theprivacyblog.com/android-2/easy-bypass-to-android-app-signing-discovered/>



6 About

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

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

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

X41 D-Sec is an expert provider for application security services. Founded in 2015 by Markus Vervier, X41 D-Sec relies on 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 to perform premium security services in the area of code review, binary reverse engineering and vulnerability discovery.

For more information, please visit <https://www.x41-dsec.de>.

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