Pentest-Report Mailvelope (RFA) 12.2012 - 02.2013

Cure53, Dr.-Ing. Mario Heiderich / Krzysztof Kotowicz, Proof-Read by Paula Pustulka

Index

Introduction

> “Mailvelope uses the OpenPGP encryption standard which makes it compatible to existing mail encryption solutions. Installation of Mailvelope from the Chrome Web Store ensures that the installation package is signed and therefore its origin and integrity can be verified. Mailvelope integrates directly into the Webmail user interface, its elements are unintrusive and easy to use in your normal workflow. It comes preconfigured for major web mail provider. Mailvelope can be customized to work with any Webmail.”[1]

Test Chronicle

- 2012/12/20 - XSS vectors in common input fields (Mailvelope options etc.)
- 2012/12/20 - XSS vectors hidden in exported public PGP keys (via seahorse)
- 2012/12/21 - HTML injections/XSS inside encrypted HTML mail body
- 2012/12/21 - Webkit CSP bypass probing via data URIs and SVG
- 2012/12/21 - Attempts to inject JavaScript via Kendo UI Templates
- 2012/12/22 - Source code analysis JS/HTML
- 2012/12/22 - XSS tests with H5SC payloads
- 2012/12/30 - Additional Tests with showModalDialog Branch
- 2012/12/31 - Tests with showModalDialog and x-domain Drag&Drop / Copy&Paste
- 2013/01/01 – Ongoing design discussion and threat modeling
- 2013/01/07 - showModalDialog tests with new payloads
- 2013/01/12 - Verified fixes, tested against new copy& paste attack vectors
- 2013/01/14 - Verified security for Xing HTML5 RTE
- 2013/01/16 - Finalized tests, discussions and report

1 http://www.mailvelope.com/about

Methodology

The test against Mailvelope was not a classic penetration test against a static target, but rather a very early evaluation of Mailvelope's security implementation and its security design's aspects. The bugs listed in this report are based on a test against an alpha version, thus they are mostly absent from currently deployed versions. Mailvelope does not embark on an easy mission. It attempts the complicated split between being secure enough in implementation, information flow and design to be able to safely "bring PGP to the browser" but as well be capable of providing a satisfying user experience to allow even non-technical users to benefit from its privacy assuring goals and features.

Note that this report by no means encourages usage of PGP and related libraries in the DOM of the enhanced *websites or web-mailers*. All cryptographic business logic as well as any form of secret data must never be available to the website's DOM, being strictly isolated in the browser's extension context. Naturally, dealing with encrypted and decrypted messages in the context of a browser extension, which resides on top of almost arbitrary web mailer interfaces, raises a set of novel challenges. To be able to enumerate those properly, a threat model has to be designed and agreed upon by development and pentest teams. The result of this agreement shall be outlined as follows:

- **A rogue sender** - attempting to abuse the lack of server-side XSS filtering dictated by an end-to-end encryption on top of a web-mailer interface for client-side attacks against the user and the web-mailer platform itself. Note that no server-side filtering can be applied to the encrypted mail body, so the extension must filter any rich-content after decryption or render the content in a safe way. Goal: All data must be safe, even if a rogue sender is part of the communication.

- **A rogue web-mailer** - attempting to gain information on the plaintext before encryption or after decryption. This includes side-channels, such as the height of a text container, keystroke intervals, focus events and other information being passed between the user and browser extension. Goal: Even if the mail provider has malicious intent, all data must be safe.

- **A rogue third party** - potentially attacking the web-mailer or any other website the user is active on; it is trying to find a lever to get a handle on the decrypted information or any other secret used during the process of creating, encrypting, receiving, decrypting and reading mails. Goal: All data must be safe, even if the mail provider was attacked or the user has a malicious tab opened.

The first version of Mailvelope was completely incapable of preventing attacks from a rogue or attacked web-mailer. The mail editor, the buttons to trigger encryption and

decryption, as well as the passphrase dialog for the key, were placed in the same window as the web-mailer UI (in our specific test-case Gmail). In order to attack the privacy of the conversation protected by Mailvelope, the adversary controlling the DOM of the web-mailer would have to create an element that overlaps the mail editor. This simple trick would make it possible for the adversary to record the keystroke for mail composition and password, obtaining a clearly unacceptable amount of data from the victim in result. The cryptographic protection would have simply been bypassed by a naive yet powerful UI redressing attack.

The only way to get around this problem across browser versions and families was to completely detach the critical dialogs from the UI, which means using *modal* dialogs. Non-modal dialogs would enable a small attack surface for focus stealing attacks. An attacker could simply request focus for a different view, and thereby redirect victim's keystrokes. Depending on the browser's capabilities, the attack could have been carried out so subtly that our victim would notice the redirected keystrokes far too late, yet again irreversibly leaking a significant portion of private information. Unfortunately, modal dialogs in Google Chrome are not modal and have never been such. This means that an attacker can place another "modal" on top of the "modal" and simply spoof its appearance, with sensitive data leakages as a consequence. For this reason, the loss of focus on Mailvelope dialog windows had to be dealt with differently and a trust token, clearly identifying the modal rather than a spoofed version is desirable. To obtain this goal, a feature seen in Cryptocat, namely the color coding for the OTR keys, was introduced. Along with that, several mechanisms capable of noticing and reacting to focus loss and focus changes were implemented. For these and several other components of Mailvelope, we needed to find solutions with a capability of providing the necessary level of security while still preserving productivity, which could, when hindered, deter the users from the tool. During the feature and design discussions, it turned out helpful to keep the most important security requirement in mind: Never must any of the users' secrets and sensitive data in any way leak to anyone other than the intended recipient(s). Neither an attacker, nor the mail provider (be it benign or not) or any other party must have any awareness of this data. This requirement includes simple XSS attacks, geometry-based side-channels, keystrokes, data storage, information flows, predictable URLs and other vectors and information sinks.

In its current stage, Mailvelope might not yet be ready for unresented global usage, but the majority of UI-based and web-security issues should have been addressed successfully in its current state of development. Whenever possible, Mailvelope currently employs several HTML5-based security features and additionally makes use of a considerably slim and hardened HTML Rich Text Editor (RTE) that includes a white-list based filter[2] for the purpose of markup sanitation.

2 http://xing.github.com/wysihtml5/

Vulnerabilities

The following sections list vulnerabilities and implementation issues spotted during our testing of the evolving Mailvelope extension. Let us highlight that all of these issues have been resolved. To facilitate reader's comprehensive experience, we decided to arrange the listed vulnerabilities in chronological order rather than by a degree of severity. In case the fix deserved a special explanation, a note was added to the paragraphs that described that particular vulnerability.

Insufficient Output Filtering enables Frame Hijacking Attacks *(High)*

Mailvelope allows decryption of the received mails and displays them instantly inside an Iframe container. This applies even to HTML mails, which can contain JavaScript that cannot be cleansed by the server-side sanitation features/output filters the mail provider offers and applies. That is probably not really news, nor an actual problem in itself. The content loads in the extension context via an Iframe that is applied with the *X-Webkit-CSP* restrictions[3]. Thus, even in case when an attacker composes a mail with arbitrary external JavaScript, none of it will actually execute in the victim's browser, and, as a result, no XSS is possible. Script content must be coming from the Iframe's head-element and be loaded from the extension folder. Consequently, since an attacker cannot XSS the extension or the Gmail windows around it, we are pretty much left with the non-scripting attacks.

One way to potentially get there and employ non-scripting attacks against Mailvelope is a technique known as frame hijacking. A link or form can be applied with a target attribute pointed at the hosting Iframe by using the value *_self*. Upon trying that, the attacker needs to realize that the target attribute is being filtered! The content after the "send, receive, decrypt" process will always be using *target="_blank"*. Therefore, it will load in a new window and avoid phishing-like spoofing attacks[4]. The Iframe hosting the Mailvelope RTE does not appear to be "hijackable". We tried SVG and embedded XLink targets, both unsuccessfully. The output data will always be applied with fresh target attributes pointing to *_blank* and even in SVG *target* has precedence over xlink:show and the often unsupported *xlink:target*[5]

During the ongoing tests, we discovered a bypass method for this protection technique and were able to inject arbitrary target attributes. This allows an attacker to replace the

3 https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html

4 http://www.whatwg.org/specs/web-apps/current-work/multipage/links.html

5 http://www.w3.org/TR/xlink/

content of the application window, the parent window and even the whole Gmail window hosting the Mailvelope Iframe. The trick we were able to use here is based on utilizing the meanwhile obsolete *<xmp>* element[6]. Once the malicious payload is being embedded inside an *<xmp>* container, the sanitation applied will remove the XMP, nevertheless leaving the embedded links containing malicious target attributes untouched. This way, an attacker can fully control the target window for the embedded links and spoof the UI of either the Mailvelope Iframe or the surrounding Gmail window.

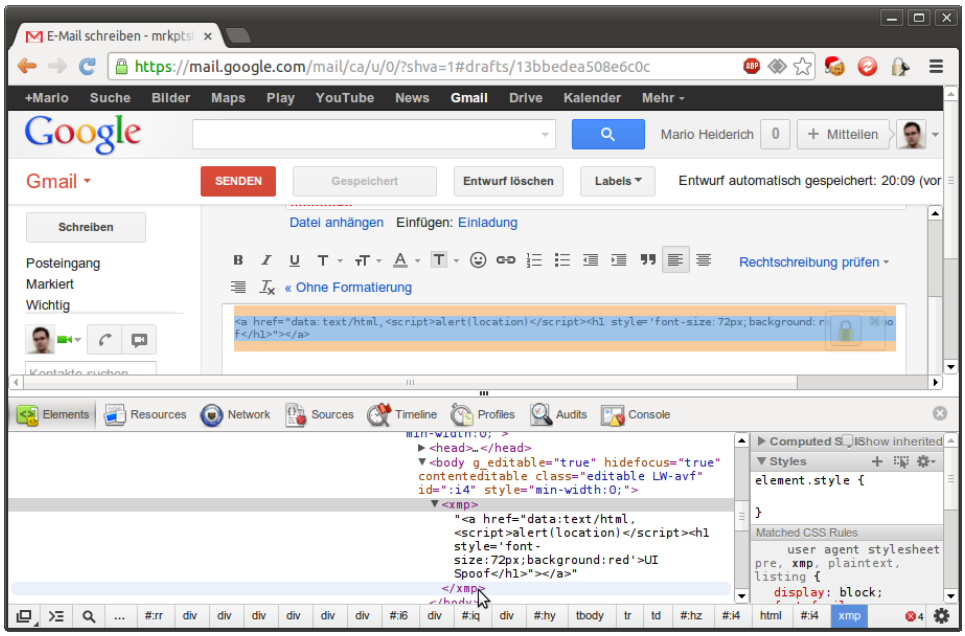

Figure A: Example injection in progress - attacker's point of view

Similar effects can be observed when one uses forms with malicious form-target or the declarative form-target override via HTML5 *formaction*[7].

Example of markup used for the attack is:

```
<xmp>
<a href="data:text/html,&lt;script&gt;alert(location)&lt;/script&gt;"
target="_self"><h1>CLICKME</h1></a>
<a href="data:text/html,&lt;script&gt;alert(location)&lt;/script&gt;"
target="_parent"><h1>CLICKME</h1></a>
```

6 http://developers.whatwg.org/obsolete.html

7 http://www.whatwg.org/specs/web-apps/current-work/multipage/forms.html

```
<a href="data:text/html,&lt;script&gt;alert(location)&lt;/script&gt;"
target="_top"><h1>CLICKME</h1></a>
<form id="foobar"></form>
<button
formaction="data:text/html,&lt;script&gt;alert(location)&lt;/script&gt;"
formtarget="_self" form="foobar">CLICKME</button>
</xmp>
```

The injected code can afterwards simply mimic either the Mailvelope UI or the whole Gmail UI, in effect tricking the user into submitting arbitrary data to an attacker-controlled resource.

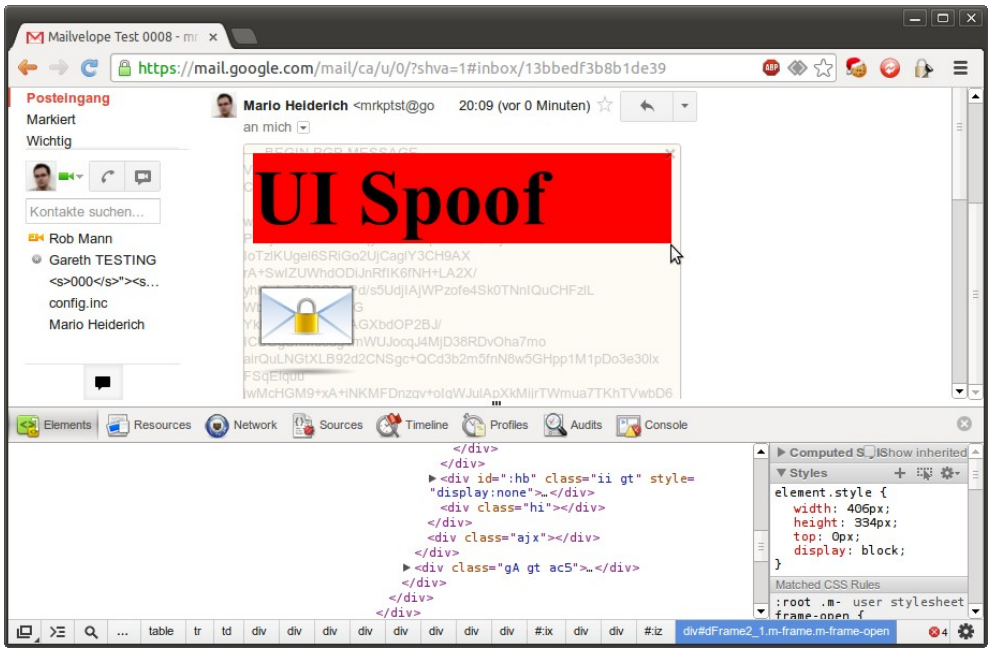

Figure B: Example injection seen by victim after a click on link

Note that we **do not** have an actual XSS vulnerability but a HTML/JavaScript injection present. So far it seemed impossible to cross domain boundaries and execute JavaScript in Gmail or extension context, rather than just the Data-URI scheme host / *about:blank*. Note though that users have no visible indicator for a spoofed Mailvelope window being present, making the likelihood of a successful attack considerably high.

Update (2013/01/07): A much simpler UI spoof payload has been discovered. It is enough to just send an HTML with arbitrary iframe prepended with e.g. *<p>*:

```
<p><iframe seamless style="border:none;width:100%:" src='data:text/html,
```

```
<img src="chrome-
extension://medhphpfknkfaaldoeefmpffekdcnibh/common/img/mail_locked_96.png">
UI SPOOF<a target=_top href="http://evil.com">mailagent spoof</a>'></iframe>
```

Upon decrypting such payload, victim is presented with attacker-supplied content rendered within *decryptDialog* <*iframe*>. Attacker can also easily replace the whole webmail interface via e.g. <a target=_top>; this would not be stopped by the CSP provided by the extension. Content can also be delivered from a third-party server, allowing for drive-by-download attacks that could not be ceased by the webmail provider.

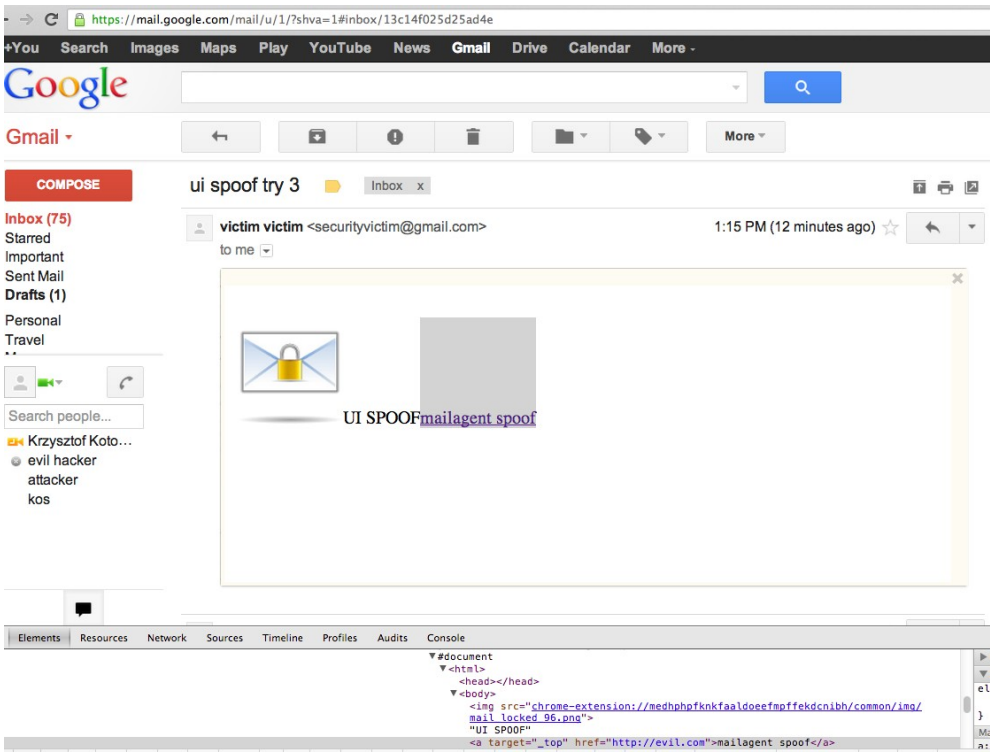

Figure B2: Simpler UI spoof payload example

To prevent webmail UI spoofing and drive-by download attacks, it is advisable to put the decrypted mail contents within decryptDialog in an HTML5 iframe sandbox[8]. This will disallow top window location replacing and all scripts/plugins within the decrypted mail.

8 http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox

Resolution: (2013/02/07) UI spoofing risks have been mitigated in the new version by means of introducing text & color watermarks randomly chosen during extension installation and user-changeable in the extension preferences. Extension dialogs are always displayed together with the watermark which is never reachable from webmail provider's DOM. They are also isolated from user-controllable inputs by the Same Origin Policy and/or HTML5 Iframe sandboxing. Similar anti-spoofing mechanism is introduced when displaying decrypted mail contents takes place.

Arbitrary JavaScript execution in decrypted mail contents *(High)*

Having similar effects to those described in paragraphs above, different markup injection can be employed, e.g. using *embed*,*object* or *iframe* elements. Contrary to the aforementioned attack, user interaction is not necessarily required to execute arbitrary JavaScript in the content of the Data URI host/*about:blank*. The following vectors' examples demonstrate an injection that will execute code directly upon the malicious mail body being decrypted:

```
<p><embed
src="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg==></embed>
<p><object
data="data:text/html;base64,PHNjcmlwdD5hbGVydCgxKTwvc2NyaXB0Pg"></object>
<p><iframe seamless style="border:none;width:100%:" src='data:text/html,
<script>alert(1)</script>'></iframe>
```

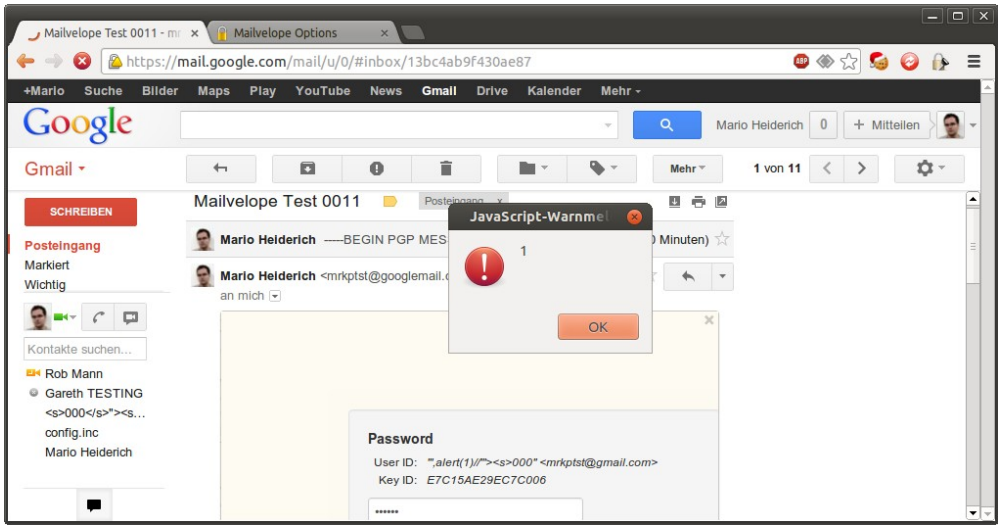

Figure C: Executing arbitrary JavaScript on about:blank via object / embed

It is recommended for remediation purposes, that no trust is being put in jQuery's *parseHTML()* method to sanitize untrusted markup. Judging by a discovered code comment, this method is being used to avoid occurrence of script element in the decrypted mails, combined with the assumption that the Google Chrome CSP mechanism will prevent JavaScript from executing:

```
/common/ui/inline/dialogs/decryptDialog.js:101
// parseHTML to filter out <script>, inline js will not be filtered out but
// execution is prevented by Content Security Policy directive: "script-src
'self' chrome-extension-resource:"
$('#decryptmail').html($.parseHTML(msg.message))
```

Instead of mitigating attacks by simply avoiding script elements and similarly active code via *parseHTML()* and trust in CSP's protective features, Mailvelope extension should employ a whitelist of allowed tags and attributes in attempts of hardening security. As an additional protection, decrypted mail should be placed within HTML5 sandboxed[9] <*iframe*> element.

Our tests have nevertheless showed that injections using Iframe elements, as well as those utilizing SVG injection, did not succeed. It remains unclear though why the Webkit CSP directive *script-src 'self' chrome-extension-resource:* permits usage of Data URIs to execute JavaScript in otherwise protected environments.

Resolution: Decrypted mail is now rendered in HTML5 sandboxed Iframe with script execution disabled.

Usage of external CSS loaded via HTTP in privileged context *(Medium)*

In some of the used HTML files, the extension is instructed to load CSS from a Google server, namely, the Google Font API.

```
/common/ui/keyRing.html:25
<link href='http://fonts.googleapis.com/css?family=Courgette'
rel='stylesheet' type='text/css'>
```

This should be avoided for numerous reasons, such as user privacy and the aforementioned consideration of perceiving the mail provider as an adversary. The font CSS should be stored in extension folders, just as the necessary WOFF files, and it shall not be transferred over the wire.

An attacker can further interfere with the loaded CSS and inject arbitrary styles in a MitM attack scenario. This issue is determined to be of medium severity. The injected CSS

9 http://www.whatwg.org/specs/web-apps/current-work/multipage/the-iframe-element.html#attr-iframe-sandbox

can be used by an adversary to introduce style-sheets with extended capabilities of sniffing sensitive information or conducting spoofing attacks.

Resolution: The insecurely referenced resource has been moved to the extension package and is therefore properly isolated.

UI Spoof via z-indexed positioned DOM elements on provider site *(Medium)*

It is possible for any of the supported mail providers (or an attacker abusing an XSS vulnerability affecting these mail providers) to create a DOM element that is positioned transparently, right on top of the encryption key's password field. An adversary can thereby sniff the encryption key's password of the user without them really being able to notice any rogue activity.

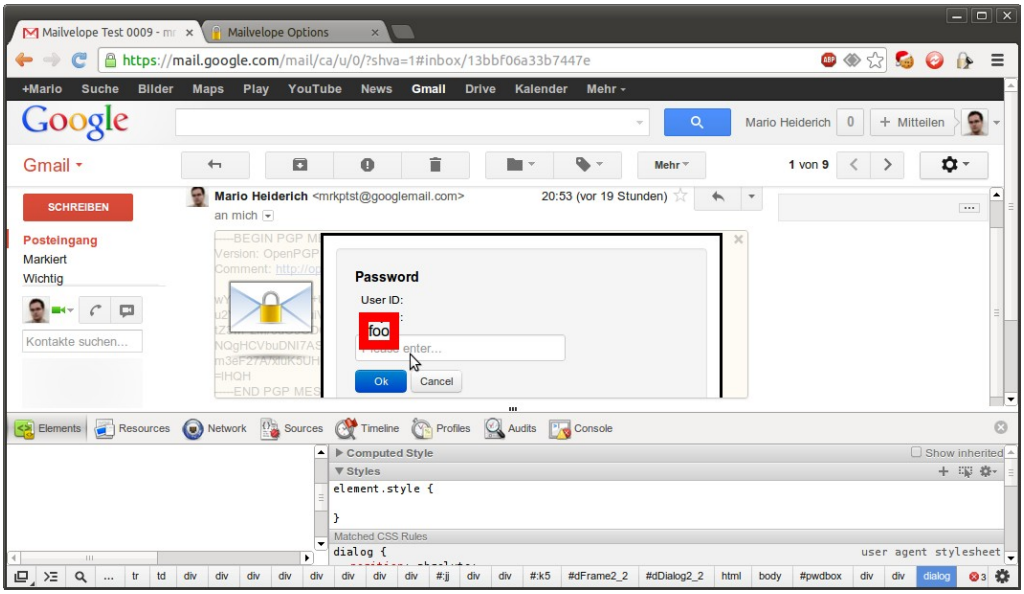

Figure D: Example: injection overlapping the encryption key's password field

No effective protection from these kinds of overlap attacks can be guaranteed when only regular HTML is employed. This holds neither for absolute positionings and very large *z-index* values nor the HTML5 *dialog* element[10]. It is recommended to display the password in a detached view created by the DOM method *showModalDialog()*[11]. In order

10 http://www.whatwg.org/specs/web-apps/current-work/multipage/commands.html

11 https://github.com/toberndo/mailvelope/tree/dom_isolation

to avoid critical data leakage, it needs to be ensured that the provider's site cannot overlap this element with other DOM elements.

Resolution: Critical Mailvelope UI elements are now displayed in modal dialogs and cannot be overlayed with content from the webmail provider's window.

Predictable GET parameters usage for connection identifiers *(Medium)*

Extension's web-accessible resources, namely *decryptDialog.html*, *encryptDialog.html* and *richText.html,* are initialized with GET *id* parameter that is later on being used to bind the resources with other components. For example, rich text editor loaded with *richText.html?id=215_1* URL will send the editor content back to a frame with the ID *215_1*. Numbers in component IDs are based on Chrome tab IDs and can be predicted or read by any Chrome extension having the *tabs* permission.

Aforementioned resources can be created with arbitrary IDs by any webpage e.g. via *<iframe src=chrome-extension://.....?id=215_1>* element. Extension allows repeated re-usage of the same ID. It is therefore possible to hijack some of the extension activities to launch UI redressing attacks on manually loaded resources.

For example, once the webmail provider (e.g. Gmail) window is currently loaded with a tab ID of *215*, and the composition frame is visible, visiting (in a separate tab) a document with the following code will enable the attack:

```
<iframe style="opacity:0.4; width: 100%; height: 500px;position:absolute;"
src="chrome-
extension://medhphpfknkfaaldoeefmpffekdcnibh/common/ui/inline/dialogs/richText.
html?id=215_1">
</iframe>
```

The act of modifying the textarea contents and clicking "Transfer" button will replace the text in mail provider's console window. This technique has been successfully combined into an UI redressing scenario where Mailvelope extension is used for triggering XSS in Gmail document context; this attack is described in 'Rich Text Editor transfers HTML content' vulnerability section.

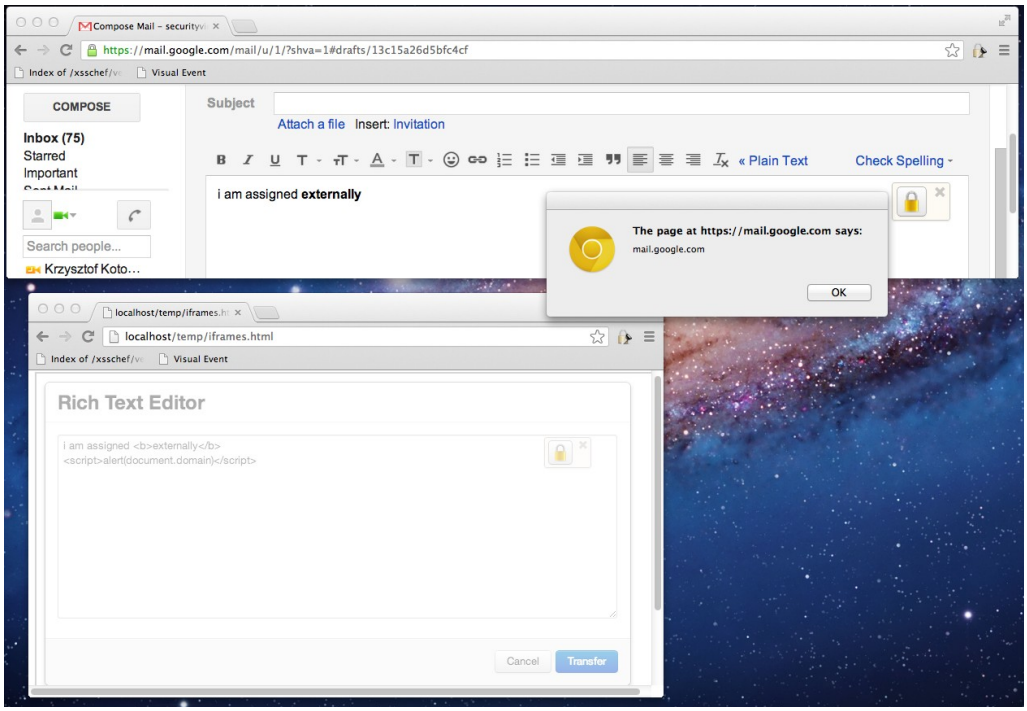

Figure: Externally embedded RTE used to trigger XSS in webmail provider document

It would be recommended to stop using GET *id* parameter for carrying binding identifiers. Instead, upon loading the resource, the calling component should send the appropriate ID via native Chrome extension message passing mechanism. Alternatively, background component should manage separate ID => random hash store, where only random hashes can be observed in URLs. Multiple usage of the same ID (e.g. by launching a few RTEs all bound to the same *encryptFrame*) should be disallowed.

As a side note, webmail provider knows appropriate IDs because of shared DOM usage present in the Mailvelope extension. It is possible for a malicious webmail provider to further interact with the extension to e.g. silently switch the plaintext before encryption (although no code path allowing this has been found yet).

Resolution: Aforementioned HTML pages are not web-accessible resources now and are being created through `chrome.windows.create()` call. Additionally, *id* parameter is now a random hash and a connection to the same *id* happening twice is no longer possible.

Rich Text Editor transfers unsanitized HTML content *(High)*

The Rich Text Editor introduced in the *dom_isolation* (formerly *showModalDialog)*[12] branch in its tested form uses *<textarea>* element that accepts and renders plain text only. The text can be encrypted (this step is optional though) and transferred back into webmail provided compose element.

However, text that is typed into RTE will upon transferring be posted into the webmail provided container (e.g. *<div contenteditable>*) and can therefore be interpreted as HTML, making XSS attacks on webmail provider through Mailvelope possible. The risk is greater as RTE can be included on any third-party page and abused through an UI redressing attack. Exemplary attack code (relying on GET parameters predictability) is demonstrated below:

```
<span style="background:red;position:absolute;top:100px;left:50px;">HERE</span>
<iframe style="opacity:0.4; width: 100%; height: 500px;position:absolute;"
src="chrome-
extension://medhphpfknkfaaldoeefmpffekdcnibh/common/ui/inline/dialogs/richText.
html?id=54_1">
</iframe>
<span style="background:red;position:absolute;top:360px;left:660px;">WIN</span>
<a href="#" style="position:absolute;left:300px;"
onclick="window.open(&quot;data:text/html,DRAGME<textarea
style=width:30px;height:20px;opacity:0.5;position:absolute;left:0 id=t>
\n\n\n\nnnn  &lt;p>&lt;img src=x onerror=alert(document.domain)
>&lt;p>bbbba<br>b<br>c<br>d</textarea><script>document.getElementById('t').sele
ct();&lt;/script>&quot;,null,'width=500')">Start game</a>
```

When visiting a page with the above code, victim user is presented with a drag&drop-based game. In the first step, user drags HTML code with XSS payload into RTE area, while in the second step, user clicks Transfer button that copies the payload back into webmail provider compose area and starts XSS/UI spoofing attack. In the Gmail interface, the code will run in *mail.google.com* context.

12 https://github.com/toberndo/mailvelope/tree/showModalDialog

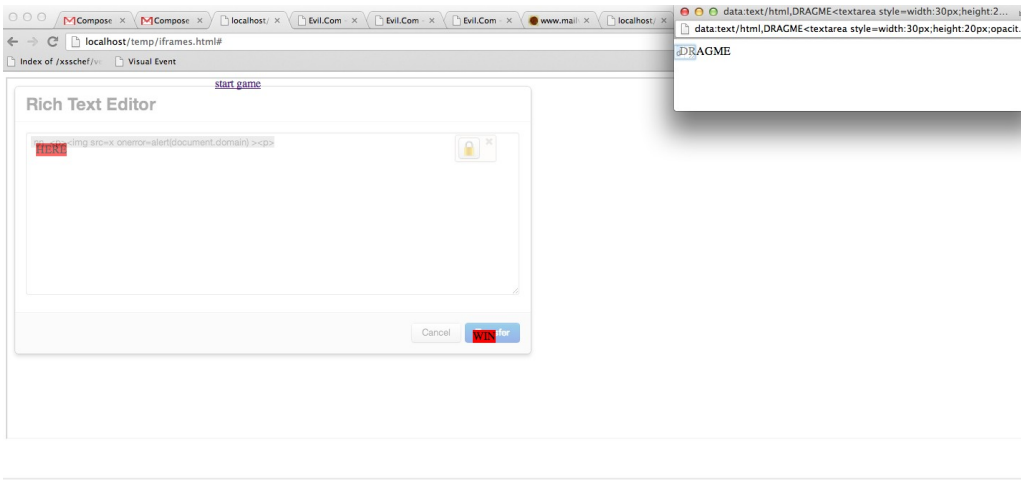

Figure: UI redressing Rich Text Editor

The Rich Text Editor should encode all HTML entities before copying the content back into webmail provider's window, with an appropriate newline to *
* conversion. Alternatively, a whitelist-based HTML sanitizer should be used.

Resolution: Mailvelope now uses Xing HTML5 WYSIWYG editor that is based on a whitelist-based sanitization filter for preventing XSS attacks. Additionally, there is a warning message if the user wants to transfer unencrypted editor contents back to the webmail provider's window.

Features in showModalDialog Branch exposing Provider to XSS Attacks *(Medium)*

Based on the architecture used in the meanwhile deprecated *showModalDialog* branch[13] tested on the 30th and 31st of December 2012, the mail provider's compose window is no longer owned by the Mailvelope extension. Since all composition logic was moved to a different window, it is now left untouched. This enables cross-domain copy&paste attacks against (in our tests) Gmail and other providers, given the Mailvelope window shows a maliciously prepared HTML mail body. Google Chrome restricts the content that is being moved between domains and filters several possibly dangerous markup elements and attributes. This protection is not reliable though and was broken during our tests, as shown by the following PoC code:

```
<xmp><div contenteditable>
<embed src="https://heideri.ch/jso/vulnerable.swf?a=1:0;alert(location)//"
allowscriptaccess="always"
></div></xmp>
```

13 https://github.com/toberndo/mailvelope/tree/showModalDialog

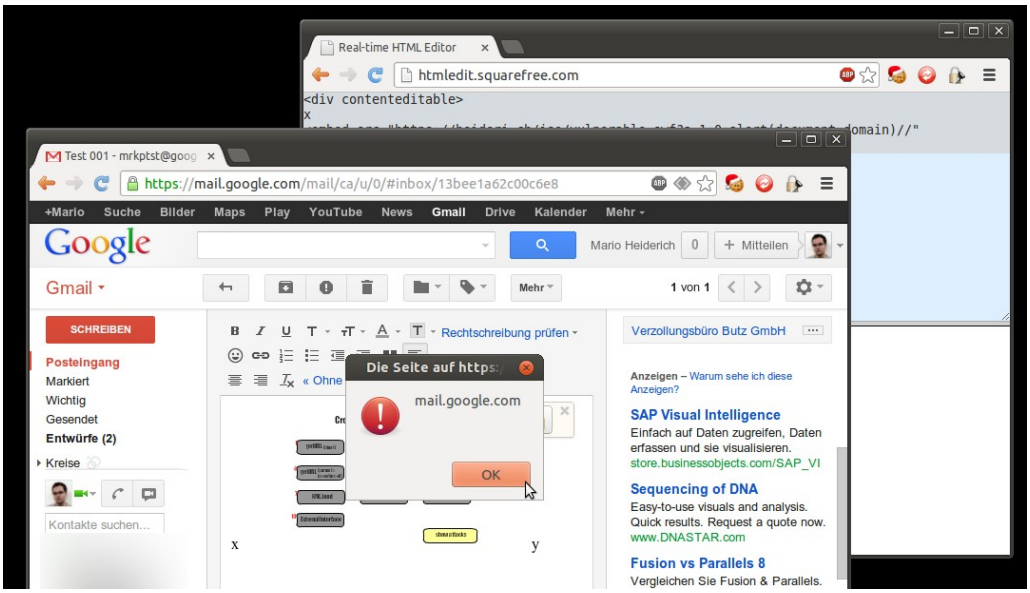

Figure E: Example of a cross-domain drag&drop XSS based on insufficient RTE protection

It is recommended to keep both mail editor of the provider, as well as the rich-text editor of the Mailvelope extension itself in Chrome extension context, in order to avoid bypasses of the otherwise fairly sufficient CSP protection. Note that user interaction is required to have the attack succeed. However, a simple social engineering mail is likely to convince less security-verse users rather easily (for example “Drag a basketball into a basket to win a free iPad”). Note that user agents mostly comply to a currently insufficient blacklist regarding cross-origin rich-text transfers[14]. Given the nature of blacklists, any new HTML feature might compromise formerly existing security - thus cross-origin views in the Mailvelope application should be avoided.

The publicly available Mailvelope version (at the time of writing: 0.5.4.2) is not affected by this problem. Here, the extension protects the rich-text editor and the installed CSP policy makes XSS attacks almost impossible. Yet *another* variation of this vector was discovered in early 2013 - pointing to a security bug in Google Chrome that was reported on 22nd of January (Issue 171392). The vector used in this bug utilizes the HTML5 srcdoc attribute to bypass the cross-domain copy&paste blacklist:

```
<body contenteditable>copy <iframe style="height:0;width:0;opacity:0"
srcdoc="<img src=x onerror=alert(domain)>"></iframe>me into a x-domain window
```

14 http://www.w3.org/TR/clipboard-apis/#cross-origin-html-paste-sanitization-algorithm

Arbitrary file download with RTE editor filter bypass *(Low)*

A newer branch created in the Mailvelope repository introduced a WYSIWYG editor with a whitelist-based filter to protect from injecting malicious scripting (or active content in general) via copy-paste and/or drag&drop operations. The editor frame, being inside Chrome Extension context, is protected via its Content Security Policy setting: `"script-src 'self' chrome-extension:"`, so attacks using inline handlers cannot function.

However, it is still possible to trigger browser actions (e.g. arbitrary content download) with pasting e.g. the following vector into the editor:

```
<p>select and copypaste me<div><iframe height=1 src="data:image/svg-xml,yay, im
downloaded"></iframe><p>select and copypaste me</div>
```

Immediately after pasting into editor frame, the file with arbitrary content is downloaded by the browser. However, the DOM contents will be sanitized afterwards by the filter. It is possible to abuse this vulnerability to form a Social Engineering attack.

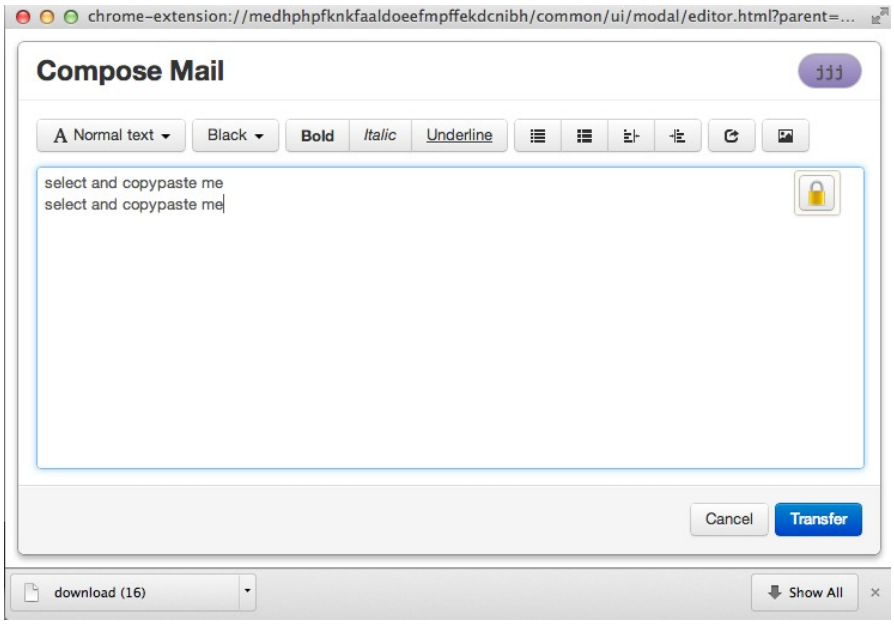

Figure F: A download triggered by an Iframe in the Mailvelope RTE

Lack of HTML sanitization when using a plaintext editor *(Medium)*

In latest version of the tool, user might optionally use a plaintext editor based on <*textarea*> instead of Xing RTE with its whitelist-based sanitation routines. This setting is not a default one, so user must manually set it at some point.

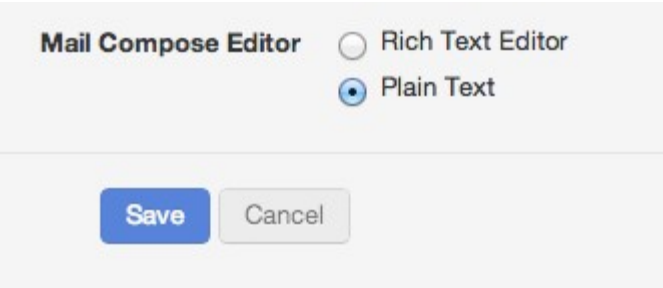

Figure G: Mailvelope 'General settings' options

In that case no HTML sanitation is taking place when the text is transferred back to webmail's editor window (similar to "Rich Text Editor transfers unsanitized HTML content" issue). Those using Plain Text editor might be socially engineered to paste malicious HTML content into textarea and transfer that content back to the webmail provider's editor (based on <*div contenteditable*>). Once transferred into webmail editor window, XSS payload will run in webmail provider's context. However, it is worth noting that attempts of transferring unencrypted mail contents displays a "*You are trying to transfer unencrypted content back to the mail provider*" confirmation dialog, yet this message is focused on plaintext disclosure and not on code injection issues.

Because the edited content is transferred back to webmail provider *as HTML*, it should be sanitized as HTML independently of the Mailvelope editor used. Sanitation should be based on a restrictive white list and a recommended approach is to reuse sanitation routines from Xing editor or HTMLReg[15] project.

15 http://code.google.com/p/htmlreg/

Miscellaneous Issues

Several unexploitable issues were detected during our tests. Those are identified and listed in the following paragraphs for the sake of the report giving a full picture.

The file *encryptFrame.js* is vulnerable against XSS attacks, with the cause being the code located in line 247:

```
text = text.replace(/(<br>)/g,'\n'); // replace <br> with new line
text = text.replace(/<\/(div|p)>/g,'\n'); // replace </div> or </p> tags ...
text = text.replace(/<(.+?)>/g,''); // remove tags
text = text.replace(/\n{3,}/g, '\n\n'); // compress new line
text = $('<div/>').html(text).text(); // decode
```

It appears though that the file is currently not being used by the Mailvelope extension. Nevertheless, even HTML code that is being processed by the *jQuery.html()* method without being applied to the DOM afterwards will be able to execute arbitrary JavaScript.

Example: $('<`div/>').html('<img src=x onerror=alert(1) \n>').text()`

Note: Precautions were taken successfully with the implementation of CSP headers for the HTML and JavaScript run in the extension context. An attack using this vector would in most cases die silently, revealing a console message informing about the CSP rule violation.

Further, unlike tools such as Thunderbird and some of the tested webmail providers themselves, Mailvelope does not yet provide a fail-safe opt-in feature for external image resources and other files the browser would load from external resources. It is to be expected, that upcoming versions will address and tackle that issue. Our recommendation to use the JavaScript library HTMLReg[16] would provide a lever to re-anonymize users receiving mails containing external resources by initially prefixing any external data source with the URI fragment `about:blank#`.

Note: Later versions were applied with a sandboxed Iframe and a Rich-Text Editor that allows usage of a HTML whitelist for filtering potentially bad markup[17]. The whitelist was thoroughly tested and no issues were discovered. The RTE has some race-condition-based XSS problems we observed, but given the fact that the editor window is loaded in a partially sandboxed Iframe and is additionally secured by CSP, no working exploit could be created. It needs to be tested whether the upcoming Firefox version of the Mailvelope extension might suffer from this issue.

16 http://code.google.com/p/htmlreg/

17 http://xing.github.com/wysihtml5/

Conclusion

Since the first emails pertaining to our tests have been exchanged, Mailvelope has undergone a lot of changes. This majority of alterations have successfully covered the risks defined by the specified and here-mentioned threat model, adversary roles. Ultimately, the goal of not allowing any form of leak of sensitive user data or similar secrets has been met.

The threat model of even considering the mail provider to be untrusted posed quite a challenge. Given the original nature of the extension - a set of isolated elements floating on top of the mail provider's UI elements - we needed to specify a better, more distinguished display of private data with the matching editor and dialogues. Yet, usability and workflow features needed to be left intact, sometimes even receiving enhancements. The lack of "real" modal windows in the Google Chrome extension scope was not exactly helpful but blur-events and other helpers managed to "do the trick" and emulate what was required.

Mailvelope is a considerably young tool and needs continuous attention from the security community - be it cryptographers peeking at the library code or UI experts making sure no overlapping, focus stealing or other spoofing attacks are possible without users clearly recognizing their presence. Further, XSS should not be left from the equation and always considered a possibility. Luckily, browser features, such as the Iframe sandbox, CSP and others, allowed us to minimize the attack window dramatically. But new and upcoming features will in no doubt further probe HTML5's capability of allowing secure browser extensions and web applications. For instance, once the encrypted uploads are added to the list of features, new challenges and risks will emerge, resulting in the HTML5 filesystem[18] and other APIs having to be used extensively. Comprehensive care should be taken with securing the upcoming Firefox version of Mailvelope. Here, many of the Chrome extension security features do not exist and a high-privilege XSS can quickly lead to a full-blown code execution.

As many other privacy tools residing in the browser and extension scope, Mailvelope is an experiment aimed to extend the reach of already well-known protective tools and libraries. The following months and years will probably be accompanied with a bumpy yet exciting ride for this family of tools. Provided they will become successful, those tools may fulfill the promise of convenient and usable privacy solutions for a target audience with a reach far greater than ever before.

18 http://www.w3.org/TR/file-system-api/