Pentest-Report Whiteout.io 04.2015

Cure53, Dr.-Ing. Mario Heiderich, Dr. Jonas Magazinius, Franz Antesberger, Jann Horn

Index

Introduction

> *“Whiteout Mail is the first email solution with end-to-end encryption based on open standards that has a beautiful user interface and is easy to use. We support all major platforms and form factors (Windows and Firefox coming soon).“*
>
> From https://whiteout.io/#product

This penetration test and source code audit against several parts of the Whiteout.io software portfolio was carried out by four senior testers of the Cure53 team. The test was completed over the course of eight days. The audit was performed against a specific version of the Whiteout software, specifically tagged as ready for testing by the maintainers. The results of this test comprise eighteen security vulnerabilities of varying severity, as well as several general weaknesses.

Prior to this test, Cure53 has since late 2013 accompanied the Whiteout team in their work towards arriving at a secure yet user-friendly PGP-based mail client solution. This penetration test report describes the last of numerous tests that took place at different stages leading to the time when the Whiteout software reached the much anticipated stable version 1.0.

The reported high severity vulnerabilities have been discussed with the Whiteout team and fixed in accordance with the agreed discussions’ outcomes. This final report describes the identified findings and gives insight into attack methodology, impact, and fix recommendations. Some of the mentioned low-severity issues will be addressed in the later versions, due to the fact that the resulting attack surface and connected risks were deemed acceptable: they would require complex preconditions for successful exploitation and can therefore await holistic handling through “Defense in Depth”.

Scope

- **Concept-Review “PGP Key Management”**
  - https://blog.whiteout.io/2015/02/06/making-pgp-key-management-invisible-so-johnny-can-encrypt/
- **Penetration-Test against Whiteout.io 0.25**
  - https://github.com/whiteout-io/mail-html5/tree/v0.25.0

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact, which is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier *(*e.g. *WO-03-00X)* for the purpose of facilitating any future follow-up correspondence.

WO-03-002 Insecure Regexps usage on DOMPurify Sanitizer Output *(High)*

Upon reviewing the usage of the XSS filter library DOMPurify, a problem was spotted that results from attempting to manually filter out certain attributes from image elements for privacy's sake. The file *mail-html5/src/js/controller/app/read-sandbox.js* uses the following code to sanitize HTML mails:

```
// sanitize HTML content: https://github.com/cure53/DOMPurify
html = window.DOMPurify.sanitize(html);
// make links open in a new window
html = html.replace(/<a /g, '<a target="_blank" ');

// remove sources where necessary
if (e.data.removeImages) {
  html = html.replace(/(<img[^>]+\b)src=['"][^'">]+['"]/ig, function(match,
prefix) {
    return prefix;
  });
}
document.body.innerHTML = html;
```

While DOMPurify outputs HTML code that is safe to assign to `innerHTML`, it does not guarantee that the code is free of potentially problematic issues like single-quotes within attribute values. In addition, a regular expression is being used against the resulting HTML string, seeking to manipulate image elements - and the manipulation is done in an unsafe way. Therefore the following attack is possible.

The attacker sends an HTML mail containing the code:
`<img src="foo' onmouseover=alert(1)//" width="200" height="200">`

After going through DOMPurify, the code looks like this:
`<img height="200" width="200" src="foo' onmouseover=alert(1)//">`

Then, if `e.data.removeImages` is enabled, the regular expression turns the code into this:
`<img height="200" width="200"  onmouseover=alert(1)//">`

This is then parsed by the browser as the following HTML:
`<img height="200" width="200" onmouseover="alert(1)//&quot;">`

While it does not seem possible to use this for an attack in most browsers due to the `Content-Security-Policy` header, it is vital that Internet Explorer 11 ignores that header. Thus, in order to make CSP rules effective in this case, it is necessary to set the `X-Content-Security-Policy` header.[1] Combined with WO-03-008, the problem described here indicates that any website can run arbitrary JS code in the context of the user, just as long as the employed browser is Internet Explorer accompanied by code like this:

```
<iframe src="https://mail.whiteout.io/tpl/read-sandbox.html#alert('this is 
xss!\nlocation: '+location)" width="100%" height="100%"></iframe>
<script>
setTimeout(function() {
  frames[0].postMessage({html:'<img src="foo\' 
onmouseover=eval(location.hash.slice(1))//" width="2000" height="2000">',
removeImages: true}, '*');
}, 3000);
</script>
```

Instead of manipulating the output of DOMPurify by means of regular expressions, it is recommended to use DOMPurify hooks or implement a proper configuration object to filter certain elements and attributes. The Hook API is predestined to be used with special element and attribute modifications, which signifies its high capability in handling this specific use case. Furthermore, it is recommended to set CSP rules for both the `Content-Security-Policy` and the `X-Content-Security-Policy` headers.

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

WO-03-003 Insecure File Download Method Fallbacks *(Low)*

In browsers failing to support the combination of the `download` attribute and `Blob`, as well as lacking support for `window.navigator.msSaveBlob()` (a setup currently appearing exclusively applicable to Safari), `Download.prototype.createDownload()` falls back to navigating to a Blob or a Data URI with attacker-specified MIME type and content.

Clearly, this is insecure because, since an attachment has a MIME type that causes the browser to treat it as an HTML document and the user attempts to download it in a browser like Safari, the HTML document will be opened with the Whiteout UI as origin. It is recommended to restrict the MIME types for the fall-back methods to a known-safe subset.

WO-03-009 Image Loading Opt-in Protection can be bypassed *(Low)*

Whiteout blocks the loading of images in HTML mails by filtering out the `src` attributes of the embedded images. However, this only happens if images are detected in the mail with the use of the following regex, which is applied to unsanitized HTML mail:

```
/<img[^>]+\bsrc=['"][^'">]+['"]/ig
```

[1] http://caniuse.com/#feat=contentsecuritypolicy

This means that the image loading opt-in can be bypassed through malformed HTML, such as the one shown in the following HTML string sample:

```
<img src=https://cure53.de/img/header.gif>
```

Analogically, the following string bypasses the same regular expression:

```
<img foo=">" src="https://cure53.de/img/header.gif">
```

In hopes of eradicating this problem, it is recommended to detect the use of `<img src="...">` in a DOMPurify hook instead and only then report back to the main document on whether the opt-in prompt should be shown. Alternatively, DOMPurify can be called twice on the same string and an image-less as well as an image-enriched version of the HTML string can be used on demand. It is also desirable to a have a second sandbox URI added. It should work like the normal one but with more restrictive CSP headers set to prevent leaks through images that are loaded via CSS, for instance. Without user opt-in, emails should only be rendered in the locked-down iframe.

Please note that this topic was taken offline and a solution was drafted using DOMPurify and a specially created configuration directive in combination with a hook. The implemented mechanism attempts to eliminate the problem of HTTP leakage via images and all other known HTML elements.

WO-03-011 No Reliable Sender Indication is implemented *(Medium)*

Whiteout neither has a reliable, usable way to identify the sender of a signed message, nor a telling mechanism for determining if a message has at all been signed. To illustrate, please note that a mail sent by “Foo Bar <foo@example.org>” looks exactly the same as a mail sent by “Foo Bar <evil@example.org>”, a mail sent by “foo@example.org” looks identical to a message sent by “foo@example.org <evil @example.org>”.

It is recommended to display status information pertaining to a received message having been signed. For signed messages, it is recommended to display the email address of the sender. It is crucial to ensure that the same email was used to perform the key lookup and verify it was stored together with the name in the key. It is additionally recommended to consider adding a way for users to mark certain keys as “known”, which would e.g. be reflected in the color of the sender’s identity and in parallel aid in preventing Homoglyph attacks.

Note: The issue has been addressed by the Whiteout-Team and was marked as resolved by Cure53.

WO-03-012 Broken postMessage Origin-Check in Iframe-Resizer *(Low)*

The iframe-resizer library uses the following origin check for cross-frame communication:

```
(''+origin !== 'null') && (origin !== remoteHost)
```

This allows any website to bypass the origin check by first navigating to a data URI or so. The impact on Whiteout is that websites opened through a link in the Whiteout client can open the "Reply" window inside the Whiteout UI with an arbitrary recipient.

It is recommended to ascertain the integrity of messages from the sandboxed iframe to the parent by giving a random authentication token to the iframe when loading it (e.g. as part of the URI). Subsequently, it should be verified that this token is part of all messages from the client.

WO-03-013 Lack of X-Frame-Options Header on Whiteout Server *(Medium)*

The Whiteout webserver does not set X-Frame-Options, which makes it possible for other websites to place the Whiteout UI in an iframe and trick the user into clicking on elements of Whiteout's UI that the user does not intend to click on ("Clickjacking"). For example, a malicious website could, depending on whether it knows the UID of a message in the user's inbox, trick the user into deleting a mail with one or two clicks.

It is recommended to set the header "`X-Frame-Options: DENY`", meaning the most restrictive policy, for the main browser UI and all other resources. Similarly, "`X-Frame-Options: SAMEORIGIN`" should be set for *tpl/read-sandbox.html* to allow framing by the main UI.

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

WO-03-014 Spoofing of Signed Messages and general UI Concerns *(High)*

Normally, when a signed message is displayed, the signed text is extracted and shown to the user. During the process of message extraction, the armoring of the signature is removed by deletion of the PGP signature block following the text.

An attacker can send a specially crafted message that allows him to append the text after the signed text. This originates from a bug determining how the signed text is extracted from the message. The application will delete the last signature block of the message, which implies that if two signature blocks are present, the first one will be validated by OpenPGP.js, while the second will be deleted by the application.

The attacker can append HTML that will render as an overlay above the original text of the message. Thereby, although the signature will be interpreted as valid, the visible message will be something entirely different. Keep in mind that there is no indication whether a message was signed unless the signature was false. Therefore the impact of the vulnerability is unclear, since an attacker could simply send an unsigned spoofed email instead. Should an indicator confirming that an email was correctly signed transpire in the future, the issue would have instantly gained criticality potential.

Proof of Concept:
Observe the following message:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

afaf
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v2.0.22 (GNU/Linux)

iQEcBAEBAgAGBQJVLQ6cAAoJEJbmz9lpkvR2Nc4H/1IvB3YDCzCdT9rIqr5t018X
x3JABFbbhi4Oh4Z8WRAsuOHU9pdyeuuink+pWRhmsYo3e3MAKnUeJrQrV/jt5oX0
hnAhT5NolkkeZvCWYGbE9u36+2CtkT2GkiYh70bEEg5lu13KZ3PpHakYwhP+yK0h
kNiKw8qv+Xcv6jZdmJeKkJKmtR7Sdo2YPhTAJMxdQ93UKRbOd7Qi7cX4cSYRM1r7
v5xyEnO+f+Jg6q+v8cSmrBBEn1KgTwpS7GSMk7+T6kVjyetYIkOwLeSVpd3Ax58p
YA5Tb3WErPAKZV5rqblvkNjq7qqDOozRMiJD1DrB5eDaVPP4MDPdTxg2xYLDr78=
=yUYL
-----END PGP SIGNATURE-----

<div style="position:fixed;width:100%;height:100%;top:0;left:0;z-
index:1000;background-color:#ccc;">This text is (not) signed!</div>

-----BEGIN PGP SIGNATURE-----


-----END PGP SIGNATURE-----
```

Note that the above email will be interpreted as having a correct signature and rendered as:

```
This text is (not) signed!
```

It is recommend to discuss the possibility of indicating the correctness of a signature. In the process of doing so, it could turn out worthwhile to find a way that is safe from CSS positioning attacks. Indicating the signature information in an area which is not prone to overlapping is desirable, as it makes it out of reach for CSS rules applicable to HTML inside the mail body. This might potentially be realized by using separate Iframes to display the information in conjunction with the postMessage API.

Note: The issue has been addressed by the Whiteout-Team and was verified as successfully mitigated by Cure53.

WO-03-016 TOFU Behavior for Forge-based TLS *(Medium)*

When the Whiteout client connects to a server using Forge-based TLS for the first time, it silently accepts and stores the server's TLS certificate without indicating this to the user, even if the certificate was not signed by a known Certificate Authority. This behavior violates reasonable assumptions about the behavior of TLS clients.

Consequently, it is recommended to either check certificates when connections are established or, if that is not attainable at this time for one or other reason, to at the very least inform the users about this behavior and warn them about the potential implications.
It is also recommended to block the transition from a CA-signed certificate to one that is not CA-signed. From this follows that the user is either not permitted to override the error, or, alternatively, he or she is only allowed to perform an override after having accepted a warning message, which must be edited to be more direct, informative and deterring than the one currently in place. The difference between a normal certificate rotation and a MITM attack with a bogus certificate should be made obvious to the user.

WO-03-017 No Forward Secrecy for TLS Connection in Forge *(Low)*

When TLS connections are established using Forge, the client only offers the cipher suites `TLS_RSA_WITH_AES_{128,256}_CBC_SHA`. These do not provide forward secrecy, meaning that if an attacker manages to obtain the private key of the server in the future, he can decrypt encrypted traffic captured in the past. It is recommended to implement a cipher suite with Forward Secrecy.

WO-03-018 Weak Passwords & Misleading Passphrase Strength Check *(Low)*

In the file *src/js/controller/app/set-passphrase.js*, the method *checkPassphraseQuality()* gives the user instant feedback on the quality of the PGP passphrase he has chosen. Unfortunately, this password strength check is dangerously misleading, as, for example, the password "passw0rd" gets a "Good" rating, and "qwerty123" is classified as "Strong". Both of these are not suitable as passwords anywhere, let alone in a PGP passphrase context.

The password strength check assumes that a passphrase containing eight random characters in the set [a-z0-9] is secure ("Strong").

While the above rule may be true for login passwords that can typically be only cracked online (as in, the attacker has to send a new login request to the service for every guess he makes), it is clearly inadequate for a PGP passphrase. A password of the above construction offers $log_2(36^8)\simeq 41$ bits of security, to which the String-to-Key function used by OpenPGP.js adds about 16 bits. This results in a total of around 57 bits of security. To put this in perspective, one shall think about the existing Bitcoin mining hardware which advertises, among others, a speed of 2^{42}hashes per second for hardware costs of 2299 USD (estimated). This translates to breaking a passphrase deemed "Strong" above within around 4.5 hours on a device with this speed, and only as long as the passphrase is indeed completely random. Evidently, though Bitcoin mining specialized hardware cannot be used for password cracking in practice, its capabilities are probably a good indication of what an adversary with a skill-set to make his own ASICs could achieve.

Especially given the lack of ability to check for common passwords such as “passw0rd”, it is recommended to either replace or remove the password strength indicator. An alternative solution is a proper verifier of the password quality is available for JavaScript is the library *zxcvbn*[2].

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

WO-03-019 Personal Data appearing in Debug Logs *(Low)*

Whenever a task is unsuccessful, the Whiteout client suggests a submission of an auto-generated bug report, which can be already found pre-filled in a new email. The message that has been generated begins with the following statement:

> *“Below is the log. It includes your interactions with your email provider in an anonymized way from the point where you started the app for the last time. Any information provided by you will be used for the purpose of locating and fixing the bug you reported. It will be deleted subsequently. However, you can edit this log and/or remove log data in the event that something would show up.”*

While login credentials are indeed scrubbed from the logs, folder names, message subjects and attachment filenames remain visible there. It is recommended to reconsider whether logging this information - especially subjects and filenames - is at all necessary. If the answer is no, then it is recommended to scrub that information in the client.

Note: The issue has been addressed by the Whiteout-Team and was marked as resolved by Cure53.

WO-03-020 Insecure Default in Implementation of BCC Feature *(Low)*

Whenever one wishes to send emails with hidden recipients (“BCC”), the Whiteout client is not functional, as it currently disables encryption completely. This alteration is signaled to the user by the changing the UI but it should be considered whether another mechanism might be more suitable.

More specifically, wild-card Key IDs could be used. These all-zero Key IDs instruct implementations which support the feature to attempt decryption with all available secret keys, even when the Key IDs do not match. It should be noted that even with wild-card Key IDs, the encrypted session key still leaks a small amount of information about the hidden recipient, and it obviously reveals the presence and number of BCC recipients. Therefore, this probably should not be done without informing the user. Enigmail seems

[2] https://github.com/dropbox/zxcvbn

to be using this method with a warning about compatibility. Another option would be to simply send two differently encrypted copies of the otherwise same message.

WO-03-021 No Caching happening for Keyserver Responses *(Medium)*

When a mail is opened for viewing, the Whiteout client looks up the key of the sender using the keyserver at https://keys.whiteout.io/, regardless of whether the key might have already been known. When the "Reply all" button is pressed, it even looks up the keys of all recipients at once. Even though the usability motivation behind this decision is recognizable, it might be a good idea to avoid leaking the recipient lists of mails to the keyserver.

It is recommended to cache keyserver replies (positive and negative ones) locally for some time, perhaps a day, while allowing the user to manually clear that cache as necessary. It could also be made possible to refresh random sets of keys with random sizes in random intervals. This will still reveal the identities of people the user communicates with, but will nonetheless provide some degree of protection for the timing and grouping of those people.

WO-03-022 Mail Server Settings are not displayed by default *(Low)*

During setup, the Whiteout client determines the mail server configuration to be used based on the email address entered by the user employed for querying the URL `https://settings.whiteout.io/autodiscovery/{email}`. Just like the Whiteout server, an attacker with the ability to impersonate it, can potentially configure the client to connect to, and therefore send the password to an arbitrary server and with arbitrary security settings. By default, the server settings are only shown to the user if the Whiteout server signals that it is unsure about the configuration - a state a rogue server would never enter of course.

While it is already possible to inspect the mail server configuration before it is used by clicking "Show Options", it is recommended to always show it to the user and display a security warning for cases when the encryption method used is not TLS (but opportunistic STARTTLS or None). Moreover, it is recommended to verify that the hostnames only contain ASCII characters that are respectively valid in hostnames (and no Unicode homoglyphs).

WO-03-023 STARTTLS Setting leads to opportunistic STARTSSL *(High)*

If the user uses the STARTTLS setting for connection security, the Whiteout client interprets that as a recommendation to "opportunistically use STARTTLS if available". This means that an active Man-in-the-Middle attacker between the user and the mail server can perform a downgrade attack by modifying the server's response to indicate that it is not capable of using STARTTLS. The Whiteout client will then send the user's login credentials over an insecure connection.

The current behavior essentially means that the Whiteout client renegotiates whether to use encryption or not for every new connection. It is recommended to instead let the user configure whether STARTTLS should be used or not. Only then, as the user actively opts for using this setup, connections without it should be disallowed.

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

WO-03-024 Links can be opened in the message frame in MSIE11 *(High)*

The Whiteout client attempts to enforce a policy that links can only be opened in new tabs. There are two mechanisms important for discussion here. The first one is that using a regular expression, the attribute `target="_blank"` is added to `<a>` tags. This can be circumvented by using an image map, which only requires the tags `<img>`, `<map>` and `<area>` for the purpose of creating a link. Together with the image display opt-in bypass WO-03-009, an image map looks like this:

```
<img width="972" height="93" src=https://cure53.de/img/header.gif usemap="#map">
<map name="map">
    <area href="https://var.thejh.net/wofo_VugIbeffeuv4.html" shape="rect"
coords="0,0,972,93">
</map>
```

The second mechanism that would prevent this is Content Security Policy. Whiteout sets the *default-src* directive, which implicitly sets *frame-src*, and in turn prevents loading arbitrary sites into the frame. However, as described in WO-03-002, this protection does not work in IE11.

Clicking on the image in IE11 will cause the link target to be opened within the message frame. If the user then opens another message, the decrypted message body is sent to the existing iframe without an origin check (with the origin for `postMessage()` set to `'*'`), meaning that the iframe can set a `message` event handler on its window to gain access to the user's mail:

```
<script>
  window.onmessage = function(event) {
    alert('got your message!\n'+JSON.stringify(event.data));
  }
</script>
```

Because the user can't see the location of the iframe, it would be possible to perform this attack without making it obvious to the user that something is actually happening.

It is recommended to filter links using a DOMPurify hook, as shown in the demo folder of the tool, where this exact problem has already been covered[3]. Like for WO-03-002, it is recommended to add the `X-Content-Security-Policy` header. Because loading

[3] https://github.com/cure53/DOMPurify/blob/master/demos/hooks-target-blank-demo.html

multiple messages into the same sandboxed iframe somewhat reduces the security benefits of sandboxing, it is recommended to, whenever a new message is opened, destroy the old iframe and create a new one prior to the loading of the message body.

WO-03-027 Public-Key Verifier approves of unknown public Keys *(Low)*

During the signup process, the Whiteout server sends a verification mail with a link containing a UUID to the user. This is automatically sent back to the server by the client to confirm that the key really belongs to an account in question. However, this mail does not contain any information about the public key, meaning that the client has no way to differentiate between legitimate verification requests and confirmation mails that were sent because of the actions devised and executed by an attacker. Furthermore, the message contains no text that would discourage a user from manually clicking the link containing the UUID.

It is recommended to add the fingerprint of the user's public key to the verification mail and let the client check it. Moreover, it is recommended to put the UUID into the mail in a form that ensures that the user will not accidentally click it (by adding an explanation and not sending the UUID as part of a link).

WO-03-028 Spoofing of Return Address using malformed Reply-To Header *(High)*

When conceiving a response to an email, the "Reply-To" email header, if specified, is naturally used to determine the recipient of the message. A problem here emerges from the way in which the "Reply-To" address is parsed. Essentially, the email might be sent to a different address than the one visible in the address field. An attacker can send an email that has a "Reply-To" set to **`"evil@attacker.com"@victim.com`**, which would display as such in the address field, but cause the response to be sent to **`evil@attacker.com`** instead of the expected address *@victim.com*.

The issue seems to be related to how email addresses are handled when sending emails. To mitigate the issue it is recommended that reply-to addresses are constrained to a more strict format, which would ensure no ambiguity or discrepancies in terms of what the user sees and what is actually being used.

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid attackers in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

WO-03-001 Loss of Entropy in *randomString()* Method of crypto-lib *(Low)*

The function *randomString()* in the file *util.js*, part of the crypto-lib that is being developed by Whiteout as well, is used for encrypting a private key before uploading it. However, a bug was found in the following line of code:

```
result += chars[Math.round(binaryString.charCodeAt(i)
       / 255 * (chars.length - 1))];
```

What happens here is a conversion of one byte to one character of the alphabet string '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ' (an alternative alphabet string could have been passed in, but within the *mail-html5* context this default was used). That means that the origin value that was once *[0..255]* is now matched to a value from the range *[0..35]*. The rest of the entropy is wasted, in spite of it being a precious resource.

Let's now have a look on the usage within the code published by *mail-html5*. The *randomString()* method is used in the file *./controller/login/login-privatekey-upload.js* at line 15 within the *LoginPrivateKey- UploadCtrl* constructor function:

```
$scope.code = util.randomString(24);
```

This `code` property is used to encrypt the private key before uploading it to the server. The *$scope.code* property does not have 24 * 8 = 192 bits of entropy but only *Math.log(36) / Math.LN2 / 8 * 24 *8* which amounts to about 124 bits. This is fair enough but definitely less than it could be. So the interface of *randomString(resultLength)* is misleading, because it makes a statement about the length of the resulting length rather than about the quality of the outcome. It is recommended to preferentially use the generated *binaryString* value and base64-encode it. In that scenario the length would be ambiguous but the quality of randomness predefined and reliable.

Note: The issue has been addressed by the Whiteout-Team and was marked as resolved by Cure53.

WO-03-004 Off-by-one Error in *randomString()* Method of crypto-lib *(Low)*

The same line as in WO-03-001 distributes the incoming bytes from *binaryString* uniformly to the characters in chars. Still, due to inaccuracy, the distribution is not as uniform as it could be:

```
result += chars[Math.round(binaryString.charCodeAt(i) / 255 * (chars.length -
1))];
```

The first and the last character of the chars string has only half of the frequency than the other character. This can be correctly by using the following code:

```
result += chars[Math.floor(binaryString.charCodeAt(i) / 256 * chars.length)];
```

Note however that if the issue WO-03-001 is fixed properly via the use of base64-encoding instead of the currently used conversion, the fix suggested above becomes obsolete.

Note: The issue has been addressed by the Whiteout-Team and verified as fixed by Cure53.

WO-03-005 Off-by-one Error in Prime Worker Code of Forge library *(Low)*

In the file *forge/js/jsbn.js* the method *bnGetPrng()* is present, tasked with generating a pseudo-random big number. This method is used during RSA key generation in the Miller Rabin pseudo-prime test. The following line gets a random number and converts it as uniformly distributed into a byte (*[0..255]*):

```
x[i] = Math.floor(Math.random() * 0xFF);
```

Unfortunately the value of 255 will never be reached because *Math.random()* generates a value between 0 inclusive and 1 exclusive. This results in a subtle weakness in the Miller-Rabin pseudoprime test. It should be corrected to the following code:

```
x[i] = Math.floor(Math.random() * 0x0100);
```

Note that here the astonishingly often despised call to *Math.random()* is acceptable because *bnGetPrng()* is only called to create the "witness" in the Miller Rabin test. If *Math.random()* was used for key creation, it would constitute a critical issue instead.

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

WO-03-008 No Origin Checks for postMessage Communication *(High)*

The JS code running under `/tpl/read-sandbox.html` does not verify the origin of message events. When Whiteout is accessed over an HTTPS website, this allows any website to frame the sandbox page (without setting the *sandbox* attribute) and pass arbitrary JSON objects to `DOMPurify.sanitize()`.

Under the slightly altered conditions of Whiteout not using the most recent DOMPurify version and the `html = html.replace(/<a /g, '<a target="_blank" ');` line (which causes an error if the value of `html` is not a string), this vulnerability would lead to XSS, allowing any website to run arbitrary JS code in the context of the Whiteout webinterface. A standing consequence of this issue, however, is that any webpage that

was opened by clicking on a link can overwrite the contents of the sandbox iframe - not just for the current message, but also for future ones - using `window.opener.postMessage()`.

It is recommended to verify the origin of messages in the `message` event handler in *read-sandbox.js* to prevent other windows from loading arbitrary content into the sandbox and avoid sending data with unexpected types.

Note: The issue has been addressed by the Whiteout-Team and was marked as resolved by Cure53.

WO-03-015 Regex-based Certificate Verification prone to Bypasses *(Medium)*

In the *tcp-socket* project, more specifically in the file *src/tcp-socket-tls.js*, the function *verifyCertificate()* is used to check the server certificate. This takes place when the user connects to the server for the first time in an environment where the connection has to be proxied through the Whiteout server. This function attempts to transform the “Common Name” and the “Subject Alternative Names” into regular expressions, against which the hostnames are then matched. This is performed as follows:

```
cnRegex = new RegExp(cn.value.replace(/\./g, '\\.')
    .replace(/\*/g, '.*'), 'i');

if (cnRegex.test(host)) {
    return true;
}
```

However, this does not take into account that the `test()` method also returns `true` if only a substring matches the regex. This means that if someone can obtain a valid certificate for the domain “example.co”, he can impersonate the domain “example.com” - and both “co” and “com” are existing TLDs. This issue could be fixed by adding a prefix `^` and a suffix `$` to the regex, but given the risk that someone might obtain a valid certificate for a name like `foo|bar.example.org`, it is recommended not to use regular expressions for this purpose.

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

WO-03-025 Unsafe Extraction of clearsigned Text *(Low)*

This issue is being flagged with very low severity rating because Whiteout currently does not have any indicators for whether a message is signed. When a clearsigned email is viewed, the method `PGP.prototype.verifyClearSignedMessage()` in *js/crypto/pgp.js* calls into `verifyClearSignedMessage()` in OpenPGP to verify the integrity of the clearsigned message. However, only the `signatures` property of the returned result object is used, the `text` property is discarded. This means that if a message contains dash-escaped text[4], an attacker can insert dashes that will be shown to the user in front of arbitrary lines without affecting the validity of the signature. In the following message the dash was inserted after the signature was created without affecting the validity of the clearsigned OpenPGP message:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

This is
- a test
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcBAEBAgAGBQJVMT9CAAoJEPasYGZHTO8I89EH/2UiGzlswqvFI4BX2NBphOzY
P7PlVa+PLBrZlL94E5ZkWPB7ts0OjPb9N9CFbUnazpk/cSwStEK5ucSxgSxPgfxO
ZN9/6XxzWGeI347QLwZsOquolCR+Fx4kAdN5gNR9eiYRN9E6IlUy/KGe0Ituyb3T
G6iFMyY/1gp+37cs+rzMHFruwsH1eTbxML4oOM41bezh6C5sV5zCh1zrN9hUCUVN
wieCv1t5KZGodm7FsO0sfCh71wxlxHFW0W+fUh9LhE1zjbRRy/z+zZivNHNToCj+
NX3LxLYll1KzIGHHamfZHPyxNIlbWL4xKr0b3VmWRt2P0GjAoFPztmSmc9E7T7U=
=YXx6
-----END PGP SIGNATURE-----
```

It is recommended to use the text returned by `verifyClearSignedMessage()` instead of keeping the original one.

Note: The issue has been addressed by the Whiteout-Team and was verified as fixed by Cure53.

WO-03-026 Key ID Collisions can prevent Key Download from working *(Low)*

If an attacker sends Alice a message from his email address, for which a crafted key is stored on the keyserver that has the same key ID as Bob's key, and then Bob sends Alice a message for the first time, the Whiteout client will not download Bob's key. Generating a key with a given key ID is relatively easy if the targeted client accepts V3 keys (and OpenPGP.js does).

However, Alice can fix this rather easily by searching for the key with the colliding Key ID and removing it. Thereby, allowing colliding Key IDs to exist in an OpenPGP client can become rather complicated. It is recommended to add an error message to `Keychain.prototype.getUserKeyPair()`, informing the user about any Key ID collisions. Alternatively, the code could be altered to identify keys by a fingerprint implemented in place of Key ID, but such a change might introduce additional complexity.

[4] https://tools.ietf.org/html/rfc4880#section-7.1

Conclusion

There is a certain stance that Whiteout takes as a powerful yet user friendly tool. Importantly, it is standards-compliant and fully PGP-compatible, thus enabling secure and convenient mail encryption. Unlike in many other approaches in the field of mail and communications encryption, Whiteout does not attempt to establish a novel cryptographic scheme, but rather aims for maximum compatibility with the existing PGP-based mail clients. When this is taken into account, one must note that Whiteout offers good integration patterns into the existing communication architectures, and, by doing so, it presents its significant real world value. Whiteout further approaches key synchronization and management to leverage disadvantages of having to perform many tedious tasks that a PGP user faces. Previously, a rather high expertise required could be blamed for hindering the technically less savvy users' involvement with encryption standards in their daily life communications.

Cure53 has followed the Whiteout team over the years, providing thorough security advice and in-depth audit. Keep in mind that the first tests and discussion started in late 2013 and continued fruitfully until this day. Cure53 is therefore in a position to have watched the software grow into the current shape of the long-anticipated version 1.0. In the current state of development, Whiteout has mastered mitigation of a large range of security issues. It has matured to a level, where an attacker can only (if at all) revert to social engineering attacks in hopes of meeting his or her goals. Only minor issues remain capable of causing limited damage to the user-base and their encrypted communications.

Providing a secure yet usable mail encryption software is not an easy task and requires consideration of a multitude of attacks, as well as a well-balanced security defense deployed across many levels. It begins on the lower layers of mail reception, mail body parsing and key management, and then moves all the way to the high up handling of the application stack, where a safe, secure and comprehensible UI is required to ensure that the user makes the right decisions. Over the mentioned time period a close collaboration of the Whiteout team with Cure53 was marked by several code audits, concept reviews, discussions and design considerations. Eventually, the software in scope reached the necessary level of maturity, allowing for it to be labeled a 1.0 version without false pretense or an overly quick jump to the first stable major release.

Cure53 would like to thank Tankred Hase, Felix Hammerl, Oliver Gajek and the entire Whiteout.io Team for this exciting project, as well as the outstanding support and assistance during this assignment. We hereby praise your perseverance in handling our often overly paranoid requests and remarks about safety and security on various levels.