Pentest-Report AccessNow 05.2015

Cure53, Dr.-Ing. Mario Heiderich, Nikolai Krein, Dr. Jonas Magazinius

Index

Introduction

“The OpConsole is a web based stand-alone platform, it is valuable for the work of the helpline but not as essential as Request Tracker. It is designed to work on two separate screens, each one presenting a visual environment. Those two environments are graphically separated (there is no link leading from an env to another). You can access both environments using the same screen, but since the informational screen concerns real-time events it is recommended to have it always on for the operators.“

From OpConsole README

This test against the AccessNow OpConsole was carried out by three Cure53 testers and took an overall of seven days to complete. The team had access to the application sources, as well as a carefully set up test server accompanied by several test user accounts with different roles. This framework was excellently prepared by the AccessNow team prior to the penetration test and source code audit, effectively enabling a very productive, full-coverage workflow.

The test yielded an overall of thirteen vulnerabilities and five general weaknesses. Among the issues, one finds several that were classified as critical in terms of their severity. These potentially most harmful vulnerabilities not only allow an attacker to remotely gain administrative privileges, but also permit an acquisition of a read and write access to arbitrary files on the server. Abusing critical and other vulnerabilities can have tremendous consequences. Namely, it may result in administrative access through session files' reading, a compromise of the database via an exfiltration of the database password and, finally, it could eventually lead to a point where a remote code execution becomes possible. All issues spotted throughout the test and source code audits were directly filed into the AccessNow OpConsole bug tracker for review and discussion. A full range of the identified issues is listed and discussed here in this final penetration test report.

The tested OpConsole application is built on top of the Perl Web Application Framework Catalyst.[1] Consequently, a large amount of time within the test scope was dedicated to the Perl sources of the application and paired with checks against the core features of the Catalyst framework. Further tests were conducted with a focus on the application's JavaScript logic, as well as the business logic, which sought to check for authentication and authorization flaws, succeeding in their discovery of issues like AN-01-014 described below. The amount and severity of the identified vulnerabilities and weaknesses calls for a thorough addressal and a re-test after the vulnerabilities have been fixed. In addition, consideration should be given to conducting a short security audit (lasting 1-2 days), prior to any major release. On a positive note, it is noteworthy that the examined code is very well-structured and well-readable, hence making auditing tasks relatively easy and certainly productive. Following this brief description of the test's scope, all of the identified vulnerabilities will be listed and discussed below.

Scope

- **GitLab Source Access**
  - URL: https://git.accessnow.org/op-console-devs/opconsole/tree/master
- **Access via VPN / Client Certificate for Black-Box and Testing**
  - URL: http://opconsole.test.accessnow.org/

[1] http://www.catalystframework.org/

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. *AN-01-00X)* for the purpose of facilitating any future follow-up correspondence.

AN-01-002 Requests for a “Day Off” can be accepted or rejected via CSRF *(Low)*

It was discovered that the requests capable of managing several states of “day off” requests can be executed using CSRF.[2] This is due to the fact that the request body is missing an Anti-CSRF token. This allows an attacker to lure a logged-in user onto a maliciously prepared website and, from there, fire state-changing requests in the name of that logged-in user. The requests capable of validating or rejecting a “day off” request are unprotected and henceforth vulnerable against the CSRF.

Example:
http://opconsole.test.accessnow.org/shifts/off/show/3

PoC (Accept):
http://opconsole.test.accessnow.org/shifts/off/validate/3

PoC (Reject):
http://opconsole.test.accessnow.org/shifts/off/reject/3

It should be made sure that the aforementioned requests can only be triggered by a POST request including an Anti-CSRF token. Alternatively, all GET links leading to state-changing actions should be applied with both an Anti-CSRF token, and a corresponding server-side check.

This problem remains one of the most commonly spotted security issues on the platform, and as such relates to tickets AN-01-003, AN-01-004 and AN-01-008. It is expected that other actions are affected as well, which is why it is recommended to implement CSRF-safe GET links more generally or avoid using GET for requests that are capable of state-changing on the server and therefore follow what is specified in RFC 2616.[3]

AN-01-003 Users can be activated and deactivated via CSRF *(High)*

CSRF attack makes users prone to activation and deactivation. The links for altering their state in the system’s database are not secured by a token. This can be abused by an attacker to either deactivate other users, or to re-activate themselves after being deactivated by an admin.

[2] http://en.wikipedia.org/wiki/Cross-site_request_forgery
[3] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

Example:
http://opconsole.test.accessnow.org/user/

PoC (Deactivate):
http://opconsole.test.accessnow.org/user/delete/18/0

Poc (Activate):
http://opconsole.test.accessnow.org/user/delete/18/1

It should be ensured that those requests can only be triggered by a POST request containing an Anti-CSRF token. Alternatively, all GET links leading to state-changing actions should be applied with an Anti-CSRF token and a corresponding server-side check as well.

This issue is one of the most commonly spotted security problems on the platform and relates to tickets AN-01-001, AN-01-004 and AN-01-008. It is likely that other actions are affected as well, so it is recommended to implement CSRF-safe GET links in general or avoid using GET for requests that are capable of state-changing on the server.

AN-01-004 Permissions can be enabled and disabled via CSRF *(High)*

In addition to a possibility for modifying a user's activation state via CSRF, altering the permission system (activating and deactivating permission rules) are also achievable through the CSRF attacks.

Example:
http://opconsole.test.accessnow.org/user/permission/

PoC (Disable):
http://opconsole.test.accessnow.org/user/permission/delete/231/0

PoC (Enable):
http://opconsole.test.accessnow.org/user/permission/delete/231/1

Again, it needs to be guaranteed that the aforementioned requests can only be triggered by a POST request including an Anti-CSRF token. Alternatively, t all GET links leading to state-changing actions should be applied with both an Anti-CSRF token, and a corresponding server-side check.

This problem remains one of the most commonly spotted security issues on the platform, and as such relates to tickets AN-01-003, AN-01-004 and AN-01-008. It is expected that other actions are affected as well, which is why it is recommended to implement CSRF-safe GET links more generally or avoid using GET for requests that are capable of state-changing on the server.

AN-01-005 Reflected XSS in Module View of POD::Browser *(High)*

A reflected XSS[4] vulnerability was spotted in the documentation section, namely the POD::Browser[5]. The problem allows gaining insights into the class structure of the AccessNow OpConsole application. The attack lets an attacker inject arbitrary HTML and JavaScript into the website's HTML and, by luring a victim to visit a maliciously prepared link, execute that JavaScript in the context of the logged-in victim.

PoC:
http://opconsole.test.accessnow.org/document/pod/module/%22%3E%3Csvg%20onload=alert%281%29%3E

A clear recommendation is to have all user-controlled data properly filtered or escaped. Characters with semantic meaning in HTML should be converted into entities. This particularly includes quotes, 'lesser-than' as well as 'greater-than' characters. Furthermore, the POD::Browser should not be exposed to the public but rather used as an internal development and debugging tool.

AN-01-006 No Security Headers used for POD::Browser *(Medium)*

Contrary to the core application, the *POD::Browser* does not deploy any HTTP security headers[6] to protect against clickjacking and related attack-classes. It is highly recommended to deploy the following headers for each and every request that is available on the target domain:

```
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Strict-Transport-Security: max-age=31536000; includeSubdomains
```

The settings can be issued platform-wide with the use of a server module that allows to manage the headers at a central place. In addition, deploying any headers that leak software versions or platform details (such as the version of the webserver or the like) should be consistently avoided.

AN-01-008 RSS Feeds in Admin Area can be removed via CSRF *(Low)*

It is possible to remove RSS feeds from the awareness-area by using a CSRF attack targeted at the admin user. While the impact of this attack is rather low, it is being mentioned for the sake of complete coverage.

PoC:
http://opconsole.test.accessnow.org/awareness/bot/rss/remove/7

[4] http://en.wikipedia.org/wiki/Cross-site_scripting
[5] http://search.cpan.org/~perler/Pod-Browser-1.0.1/lib/Pod/Browser.pm
[6] https://www.owasp.org/index.php/List_of_useful_HTTP_headers

It should be made sure that the aforementioned requests can only be triggered by a POST request including an Anti-CSRF token. Alternatively, t all GET links leading to state-changing actions should be applied with both an Anti-CSRF token, and a corresponding server-side check.

This problem remains one of the most commonly spotted security issues on the platform, and as such relates to tickets AN-01-001, AN-01-003 and AN-01-008. . It is expected that other actions are affected as well, which is why it is recommended to implement CSRF-safe GET links more generally or avoid using GET for requests that are capable of state-changing on the serve

Note: At this point of the test it was decided to remove CSRF vulnerabilities from the scope, instead having this wider problem treated in a more systematic and targeted manner.

AN-01-009 RSS Feeds allow to cause persistent XSS for all Users *(**Critical**)*

The content that is fetched from the RSS feeds[7] and displayed in the news-area at the page bottom is neither escaped nor filtered properly. This allows anyone with control over the feed content to cause a remotely-administrable, persistent XSS that affects all users that are logged into the application, including the administrators.

As a consequence, the attacker can take over sessions, impersonate the affected users, and do additional damage by deploying browser exploits. The latter presents a rationale for classifying this issue as critical.
Note that all site owners delivering feed content will notice the AccessNow application requesting the content via log.

Example RSS:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
<channel>
    <title>Cure53</title>
    <item>
      <title><![CDATA[XSS]]></title>
      <link>javascript:alert(2)//"onmouseover="alert(3)"style="position:fixed;
            top:1px;left:1px;height:99em;width:99em;">
      </link>
      <description><![CDATA[XSS]]></description>
    </item>
  </channel>
</rss>
```

Example RSS URL:
http://cure53.de/test-rss

[7] http://en.wikipedia.org/wiki/RSS

Resulting HTML:

```
<div id="features">
   <ul>
      <li>
           <a href="javascript:alert(2)//"onmouseover="alert(3)"
              style="position:fixed;top:1px;left:1px;height:99em;width:99em;">
              " title="" target="_blank">
              javascript:alert(2)//"onmouseover="alert(3)"
              style="position:fixed;top:1px;left:1px;height:99em;width:99em;">
           </a>
       </li>
    </ul>
</div>
```

It is strongly recommended to filter and encode all data that can be controlled by users and external entities while being later (potentially or predictably) used or displayed in the AccessNow application. The recommendations given in AN-01-005 also apply here, since it is of great importance to encode characters such as quotes, lesser-than and greater-than in hopes of prohibiting HTML and JavaScript injections attacks that lead to XSS issues like this one.

Note: The RSS parser was also tested against XML-based attacks but proven immune. Following further investigation, it turned out that the parser is purely built upon regular expressions. [8]

AN-01-010 Directory Traversal in Controller::Document::Markdown *(High)*

The feature designed to allow attaching code repositories and offer included MD[9] files for reading on the platform is plagued by a directory traversal[10] vulnerability. It lets an attacker navigate around the server and learn about the directory structure. This happens because the feature receives a parameter responsible for specifying the directory in which the cached MD files reside. Unfortunately, this parameter in is not validated properly: as it is permitted to contain dots and slashes, it allows directory traversal.

Poc:
http://opconsole.test.accessnow.org/document/markdown/show?r=../../../../../../

Affected Code:

```
my $repository = $c->req->param('r');
my @paths = split ("/", $repository);
my $root = OpConsole->path_to('root', 'src', 'markdowns', $repository);
opendir my($dh), $root or die "Couldn't open dir '$root': $!";
my @dirs = readdir $dh;
foreach my $dir (@dirs){
    ...
```

[8] http://search.cpan.org/~tima/XML-RSS-Parser-4.0/lib/XML/RSS/Parser.pm
[9] https://help.github.com/articles/github-flavored-markdown/
[10] http://en.wikipedia.org/wiki/Directory_traversal_attack

```
my $current = (grep {-d} $root.'/'.$dir)[0];
```

It is highly recommended to validate the parameter properly and make sure that a user can only traverse into directories that are legitimate and belonging to the cached Git repository. It must be assured that no traversal outside those directories can happen. Problematic characters should be forbidden and the path should be normalized and checked against the expected path prior to being accessed.

AN-01-011 Persistent XSS in Markdown Docs via MD-Injection *(High)*

The feature tasked with connecting Git repositories and offering included MD documents for reading on the platform is highly vulnerable against persistent XSS. Anyone with access to the documents can connect a repository with malicious documents that will later be rendered by the application and contain arbitrary HTML and JavaScript. In the current state, there are two pertinent problems.

First issue is that the markdown to HTML converter can easily be tricked into creating malicious HTML by a simple mixing of markdown and HTML syntax. Secondly, the tool does not stop the actual HTML from being shown, simply passing it through to be displayed without modification instead.

Example MD:

```
![GitHub Logo](/images/logo.png?"onerror=alert&lpar;1&rpar;//)
[GitHub]
(javascript:alert&2par;1&rpar;//http://github.com"onclick=alert&lpar;3&rpar;//)
<svg onload=alert(4)>
```

Resulting HTML:

```
<div id="md"><p><img title="" alt="GitHub Logo"
onerror="alert(1)//&quot;" src="/images/logo.png?"></p>
<p><a onclick="alert(3)//&quot;"
href="javascript:alert(2)//http://github.com">GitHub</a></p>
<p><svg onload="alert(4)"/></p></div>
```

It is strongly recommended to use a proper HTML sanitizer to filter the HTML after it has been processed by the markdown parser but before it is to about be rendered. Given, that in the current state the HTML is fetched via XHR[11] and then rendered via JavaScript, usage of a sanitizer library such as DOMPurify[12] is a recommendable and highly possible solution.

Should this usage pattern ever change, the use of a server-side HTML sanitizer, such as HTML::Scrubber, [13] might be appropriate. In addition, it should be considered to display

[11] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest
[12] https://github.com/cure53/DOMPurify
[13] http://search.cpan.org/~nigelm/HTML-Scrubber-0.14/lib/HTML/Scrubber.pm

the user-controlled document content inside a sandboxed Iframe to guarantee that no malicious script can get access to sensitive data and user credentials.

AN-01-012 File Creation and Parameter Injection via Git::Repository *(**Critical**)*

The Markdown-Editing feature not only supplies a possibility of creating Git Repositories in the document/markdown directory in the webroot, but basically allows an attacker to clone repositories into arbitrary locations on the entire server. This is shown in the file *Markdown.pm* in the following code:

Affected Code:

```
sub add :Local :Args(0) FormConfig('conf/document/markdown/add.yml'){
    ...
        Git::Repository->run( clone => $form->{_processed_params}->{url},
            $root.'/'.$form->{_processed_params}->{name});
    ...
}
```

Here the attacker-controlled data from the “url” and “name” parameters are directly inserted into the `Git::Repository->run()` statement[14] without making sure that the parameters do not traverse the directories. A request like the one shown below demonstrates that Git will then try to clone into the `/tmp`-Directory.:

Example Request:

```
POST /document/markdown/add HTTP/1.1
Host: opconsole.test.accessnow.org
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded

name=../../../../../../tmp/cure53test&url=git@git.accessnow.org:bechir/test.git&
submit=&_token=bp1hokd4ix79vpd1
```

This can also be verified with another directory traversal vulnerability described in AN-01-010. If an attacker is able to find a writable directory within the webroot, he might be able to drop shells and thus reach command execution. Of course this feature is usually limited to admin users (or users with the ability to add new Git repositories), however, in combination with AN-01-013 which signifies the risk of every user’s ability to gain admin privileges, this poses a substantial risk.

In general, it is a bad idea to simply feed user-input into Git commands as an attacker is then also able to issue arbitrary parameters to the Git binary. A simple test-script shows that this is in fact possible:

Test Code:

```
#!/usr/bin/perl
use Git::Repository
Git::Repository->run( clone => '--help', 'command');
```

[14] http://search.cpan.org/~book/Git-Repository-1.313/lib/Git/Repository.pm

This will result in the following command awaiting execution:

```
execve("/usr/bin/git", ["git", "clone", "--help", "command"], [/* 18 vars */]) =
0
```

It is recommended to make sure that parameters passed to `Git::Repository->run` are carefully treated and do not contain special characters that result in directory traversals.

AN-01-013 Information Disclosure via Error Reporting *(Medium)*

The error reporting mechanism that the platform currently employs is far too verbose. Consequently, it should be deactivated completely for the production version of OpConsole. As it stands, some examples of its behavior are that it echoes the cookie values that are otherwise protected by HTTPOnly[15], leaks sensitive user data and, in addition to that, shows the user's password hashes and other sensitive debug data.

PoC:
http://opconsole.test.accessnow.org/document/markdown/md/XXX

Example Output:

```
email       => "axxxx\@accessnow.org",
first_name  => "AXXXXX",
id          => 9,
is_active   => 1,
is_rescue   => 0,
is_super_admin => 1,
last_login  => "2015-05-15 08:50:49",
last_name   => "SXXXXX",
load        => 3,
office_id   => undef,
passphrase  => undef,
password    =>
"{SSHA}Ls6OpTcoJFAxlScDRrsEMOzc4EyJG4KTjRb8zxA+GAMwNhaGIzRi5g==",
```

The debug output was provoked by "uploading" of a file using the issue described in AN-01-012. That is why it dramatically differs from the usual error pages which report server errors. The functionality of displaying debug data should be disabled on production machines. A simple deactivation of the error reporting mechanism invariably fixes this issue.

[15] https://www.owasp.org/index.php/HttpOnly

AN-01-014 Overwrite super_admin with modified POST Request *(**Critical**)*

The source code audit of the application unveiled a serious flaw in the administrative area of the tested OpConsole application. It showed that by using a slightly modified POST request, any active user can become a super-admin, while also acquiring a capacity to remove the super admin privileges from any other person holding them. Crucially, it is the sole requirement for the attack to work that one is able to log-in as an active user, equipped with any arbitrary role. Then a simple POST request needs to be executed.

The affected code shown below illustrates the problem. In essence, the code checks the user ID from the POST body against the user ID of the logged-in user. Then, if those IDs match, the logged in user becomes a super-admin and this privilege can be effectively revoked for all former super-admins. In the example, the super admin takeover was performed by the test-user “anivia”, who held the user ID 9.

Affected Code:

```
if ($form->submitted_and_valid){
      my $supper_admin_id = $form->param_value('user');
      my $users = $c->model('DB')->resultset('User')->search(
            { is_active => 1});

      while (my $user = $users->next) {
            if($user->id == $supper_admin_id){
                  $user->update({is_super_admin => 1});
            }else{
                  $user->update({is_super_admin => 0});
            }
      }
      $form->model->update($config);

      $c->session->{success} = "Your changes have been saved";
}
```

Example Request:

```
Host: opconsole.test.accessnow.org
Referer: http://opconsole.test.accessnow.org/config
Cookie: opconsole_session=d7484124f247f3f7f8561658fb93b568fb7c488c
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 583

domain_name=accessnow.org&user=9&email=opconsole
%40accessnow.org&password=&passphrase=&gpg_homedir=%2Fvar%2Fwww
%2F.gnupg&gpg_config_file=%2Fvar%2Fwww%2F.gnupg
%2Fgpg.conf&gpg_gnupg_path=%2Fusr%2Fbin
%2Fgpg&rt_vetted_customfieldid=5&rt_vettingconversation_customfieldid=15
&rt_listclient_customfieldid=17&rt_rank_customfieldid=16&rt_unprivileged
```

```
groups_id=5&rt_gpg_homedir=%2Fhome
%2Fgpg&shift_cycle=4&submit=&_token=hunkfuz4k17pyw7k
```

The code needs to be fixed urgently. The ID that the super-admin is checked against should by no means be controllable via POST. Instead, it shall be fetched from the database independently of the user-controlled input. Additionally, a privilege check is required to make ensure that the current super-admin is exclusively able to reach this feature.

AN-01-016 Arbitrary File Download via md-Handling in Markdown *(**Critical**)*

It was discovered that an attacker can download almost arbitrary files from the webroot thanks to a bug found in the Markdown controller. In combination with Catalyst's URL and path handling, *Markdown.pm* contains yet another directory traversal vulnerability that offers a possibility of downloading completely arbitrary files. The responsible code is located in `sub md`. It is shown below:

Affected Code:

```
sub md :Local {
    my ( $self, $c, @paths) = @_;
    my $path = join ("/", @paths);
         $c->serve_static_file(OpConsole->path_to('root',  'src',  'markdowns',
$path));
}
```

As it turns out Catalyst does not filter "../" directives from the URL itself, leaving the user in charge of handling such paths carefully. OpConsole, however, makes no use of any validation mechanisms here and combines the path with Catalyst's `serve_static_file`. As a result an attacker is able to view the database configuration:

Example Request:

```
GET /document/markdown/md/../../../lib/OpConsole/Model/DB.pm HTTP/1.1
Host: opconsole.test.accessnow.org
Cookie: opconsole_session=aa4a3413ee93f3dc03d7f2047af4d74ab23790d6
```

It is also possible to display the private SSH-Key found in the *.ssh* directory:

Example Request:

```
GET /document/markdown/md/../../../script/.ssh/id_rsa HTTP/1.1
Host: opconsole.test.accessnow.org
Cookie: opconsole_session=aa4a3413ee93f3dc03d7f2047af4d74ab23790d6
```

Note that in order to test this issue one is required to send 'raw' HTTP Requests, since normal browsers usually will not interpret "../" as a traversal that goes beyond the webroot. Like in similar issues, OpConsole needs to guarantee that user-supplied paths do not traverse outside the appropriate directory.

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.

AN-01-001 Limited inline CSS Injection for Operator Colors *(Info)*

A very limited and seemingly harmless CSS injection was spotted in the area where a user label's background color can be influenced with a Hex-Color value. The backend application does not properly validate the color string and therefore allows for almost arbitrary injection into the style attribute of the

Affected URL:
http://opconsole.test.accessnow.org/shifts/operator/show/2

Example vector:
Set background color to: `1;color:red;`

It was however not possible to conduct a real CSS-based attack as the data submitted by the user is heavily length-restricted. In that sense, there is not enough characters available to carry out a CSS-based XSS attack or alike. Thus, this issue is not considered a vulnerability but rather marked as a general weakness to be kept in mind.

AN-01-007 Passive Self-XSS in Admin Area for RSS Feed URLs *(Medium)*

A passive Self-XSS attack vector was spotted in the RSS administration area. The content for RSS feed URLs is not properly validated to use either the http: or https: protocol. Thus, it is possible to use JavaScript URIs (and other dangerous protocols such as *data:*, *vbscript:* and others[16]) for causing passive XSS.

Affected URL:
http://opconsole.test.accessnow.org/awareness/bot/rss

Example vector:
Use the value `javascript:alert(1)` as feed URL

The only reasonable attack scenario here would be one administrator trying to attack another administrator. Nevertheless, administrators have enough power to conduct other attacks with higher severity if targeting each other. This is a rationale behind this issue being a general weakness rather than a vulnerability calling for an immediate action. In hopes of mitigating this issue properly, only URLs using *http:* or *https:* as protocols should be allowed as feed URLs.

[16] http://en.wikipedia.org/wiki/URI_scheme

AN-01-015 Directory Traversal via Referer in Markdown Controller *(Low)*

Alongside the already described issue AN-01-010, the file *Markdown.pm* contains another directory traversal weakness. In this particular scenario an attacker is able to download arbitrary files that end with ".png", ".gif" and so on. This behaviour can be observed in the following snippet:

Affected Code:

```
sub index :Path  {
    my ( $self, $c, @paths ) = @_;
    if($paths[-1] =~ /\.(png|gif|jpg|jpeg)$/i){
        my $ref = $c->request->headers->referer;
        $ref =~ /r=(.*)$/;
        my $path = join ("/", @paths);
        $path = $1 . '/' . $path;
        $c->serve_static_file(OpConsole->path_to('root', 'src', 'markdowns',
        $path));
```

While this poses no real risk at this point, the ability to traverse directories via attacker-controlled referrers that contain the substring "r=../../" remains an option that should be prevented.

AN-01-017 Invalid RSS Feed DOS of Informational Dashboard *(Medium)*

An attacker can cause a DOS of the entire Informational Dashboard for all users by specifying a valid URL that returns an invalid RSS XML document. Every page that includes *OpConsole::Model::Awareness::Rss* will be unavailable after exploiting the vulnerability. The vulnerability seems to stem from *XML::RSS::Parser::Lite* throwing an error when parsing an invalid document.

Affected Code (Rss.pm):

```
28: my $rp = new XML::RSS::Parser::Lite;
29: $rp->parse($xml);
```

This vulnerability can easily be triggered using a data-URI[17] as follows:

```
data:text/xml,<a></a>
```

To eradicate this problem it is recommended to ensure that the parser does not fail to actually parse the document. If this condition is met, it is also crucial to ascertain that the application degrades gracefully and does not deny service of all connected pages.

[17] http://en.wikipedia.org/wiki/Data_URI_scheme

AN-01-018 Forgotten Password Feature allows Email Enumeration *(Low)*

When offering a recovery facility like the “Forgotten Password” function described in *User.pm,* the web application has to make sure that it does not reveal whether a certain registered email address or username are in fact in operation. However, OpConsole appears to reveal this information as its output differs between the case when a valid email is provided, and an attempt with an invalid one. The information gained from this can be used by an attacker to come into possession of a list of emails found in the system, which can later be used for targeted brute-force attacks.

Affected Code (User.pm):

```
sub forget_password ... {
...
    $c->session->{success} = "You might receive an email soon,
                Please check your ".$user->email."'s inbox.";
...
    $c->session->{error} = "Email address not found or not active anymore!";
...
}
```

It is recommended to remove the error message and just inform the user that he is going to receive a new password, regardless of whether the entered email was correct or not.

Conclusion

The AccessNow OpConsole application impresses through its well-written and peroperly structured Perl code. Using the Catalyst framework allows for a clean structure and lightweight business logic. Together they result in a small footprint as far as testing efforts are concerned. Most of the core application logic passed the security audit. No severe HTML injections or XSS problems were found in the core applications since a majority of the forms were created cleanly with the use of the respected form builder. In addition, database queries were properly parametrized and parameters were found to be correctly validated.

The majority of issues identified by the audit and described in this report were stemming from the usage of external libraries and tools. Some prominent examples include issues with the RSS parser and the Git integration. Through those somewhat foreign elements an attacker could exploit the unvalidated reception of parameters and data from RSS, Git and other entities. Eventually, those problems amounted to gaining an exploitation leverage. Ultimately, conducting a persistent XSS attack, writing arbitrary files on the server, navigating the application root, as well as reading configuration files and private keys all became possible. Other issues, mostly of logical nature, were caused by trusting the user and their input just a bit too much. This was the primary origin of the super-admin flag's exposure to being overwritten in a user's database entry. Thereby an unprivileged user was granted administrative rights and simultaneously became powerful enough to logout the actual admin.

It is recommended to put highest priority on fixing and re-structuring the ways in which external applications' input is processed. The path traversal issues with the GIT integration and the complete lack of XSS protection of the entirety of the RSS feature need to be addressed urgently. Similarly, the way markdown documents are being handled needs to be changed - namely a strong XSS filter needs to be added. Once those mitigation approaches have been installed, a fix verification process is highly recommended, optionally followed by a brief re-test of the potentially newly-added code. In addition, it should be considered to conduct regular code audits and ascertain that every major release with significant changes is being tested before going into production. The tool processes personal data in critically important scenarios, and must therefore be able to deliver a maximum of security, and keep the data protection promises. Given the well-structured code and cleanly implemented business logic, this should be attainable without major effort. Choosing a MVC framework that handles a lot of interactivity on its own was a wise choice and helps the application to scale features and security in a proportional way.

Cure53 would like to thank Bechir Nemlaghi, Al Walid Chennoufi and the entire AccessNow Team for their outstanding support and assistance during this assignment.