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

Pentest-Report Ledgy Web UI, API & Backend 03.2021

Cure53, Dr.-Ing. M. Heiderich, Dipl.-Ing. A. Inführ, BSc. C. Kean, M. Rupp

Index

Introduction

“Digitize your cap table, automate equity plans and simplify compliance. Find out how leading companies use Ledgy to empower their employees and engage investors.”

From https://ledgy.com/

This report - entitled LED-01 - details the scope, results, and conclusory summaries of a penetration test and security assessment against the Ledgy web UI, backend, and API. The work was requested by Ledgy AG in December 2020 and enacted by Cure53 in early March 2021, namely in CW10. A total of eleven days were invested to reach the coverage expected for this project.

The testing conducted throughout *LED-01* was divided into two separate work packages (WPs) for execution efficiency, as follows:

- **WP1**: White-Box Tests & Source Code Audits against Ledgy Web UI & Frontend
- **WP2**: White-Box Tests & Source Code Audits against Ledgy Backend API

Cure53 was granted access to source codes, test user accounts, and a variety of supporting documentation. Given that all of these assets were necessarily required to procure the coverage levels expected by Ledgy, the methodology chosen here was white-box. A team consisting of four Cure53 senior testers were assigned to this project’s preparation, audit execution and finalization.

All preparations were completed in late February 2021, namely CW09, to ensure that the testing phase could proceed without hindrance. Communications were facilitated via a dedicated Ledgy Mattermost instance that was deployed to combine the workspaces of Ledgy and Cure53, thereby allowing an optimal collaborative working environment to flourish. All participatory personnel from both parties were invited to partake throughout the test preparations and discussions.

One can denote that communications proceeded smoothly on the whole. The scope was well prepared and clear, no noteworthy roadblocks were encountered throughout testing, and cross-team queries were kept to a minimum as a result. Ledgy delivered excellent test preparation and assisted the Cure53 team in every respect to procure maximum coverage and depth levels for this exercise. In regard to the findings, the Cure53 team instigated excellent coverage over the WP1 and WP2 scope items, identifying a total of nine findings. Three of these findings were classified as security vulnerabilities, and six classified as general weaknesses with lower exploitation potential.

Positively, the highest severity level reached in this assignment was *Medium* - an excellent outcome for Ledgy indeed. Despite the complex scope and the associated risk with feature exposure via mechanics such as PDF conversion, no *High* or *Critical* severity issues were unearthed. Additionally, the volume of findings is unusually low for a scope of this caliber, which can certainly be considered a positive sign in addition. The report will now shed more light on the scope and testing setup as well as provide a comprehensive breakdown of the available materials. Subsequently, the report will list all findings identified in chronological order. Each finding will be accompanied by a technical description and a Proof of Concept where applicable, plus any relevant mitigatory or preventative advice to action.

In summation, the report will finalize with a conclusion in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the Ledgy web UI, backend and API, giving high-level hardening advice where applicable.

***Note**: This report was updated with fix notes for each addressed ticket in late March 2021. All of those fixes have been inspected and verified successfully by the Cure53 team in March 2021.*

Scope

- **White-Box Penetration tests & code audits against Ledgy Web UI, Backend & API**
  - **WP1**: White-Box Tests & Source Code Audits against Ledgy Web UI & Frontend
    - Production environment:
      - https://app.ledgy.com
    - Staging environment:
      - https://beta.ledgy.com/
    - Test User
      - Account:
        - elon@must.com
      - Full Plan:
        - Ledgy Inc. company
    - Email Access:
      - URL:
        - https://mailtrap.io/inboxes/368483/messages
      - User:
        - timo@ledgy.com
  - **WP2**: White-Box Tests & Source Code Audits against Ledgy Backend API
    - See above
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Identified Vulnerabilities

The following sections list all vulnerabilities and implementation issues identified throughout the testing period. Please note that findings are listed in 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. Furthermore, each vulnerability is given a unique identifier (e.g. LED-01-001) for the purpose of facilitating any future follow-up correspondence.

LED-01-001 WP2: Remote resource feature enabled in Aspose library *(Low)*

Note: *This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

The Ledgy application offers users the ability to upload PDF or DOCX files, which can be digitally signed by specified recipients. To properly support DOCX files, the user-controlled office file is converted by an internal service to a PDF file. The discovery was made that not only is the Aspose library utilized, but that remote resources like images are fetched by the backend. This could allow attackers to send HTTP *GET* requests to internal systems, leak the content of local images or cause a denial of service via so-called decompression bombs. Pertinent to note here that the impact of this issue was diminished; despite the fact that the Aspose library supports a multitude of different file formats, the backend code correctly verifies the content-type of the uploaded document to ensure no unintended format is parsed by Aspose.

The issue was verified by uploading a DOCX document, which includes a remote image in its document. Additionally, it was tested and verified that the *jar:* protocol handler is available as well. This allows it to specify resources inside a remote ZIP file, which in turn facilitates the potential for triggering a denial of service via ZIP compression bomb.

Steps to reproduce:

1. Open office suite (e.g. LibreOffice).
2. Create a word file and include a local image “linked”.
3. Save the file as a DOCX file.
4. Unzip the file created in step 3.
5. Look for the following file. It contains the *file:///* URI pointing to the local image.
   word/_rels/document.xml.rels
6. Change the URL to, for instance, http://example.com/favicon.cio.
7. Re-zip the structure and assign a DOCX extension.
8. Upload the file and assign recipients to sign it, triggering the conversion of DOCX to PDF.
9. The remote image fill can then be fetched by the Aspose library.

The recommendation can be made to disable the support of remote resources during the conversion of DOCX files to PDF in *Aspose*. Aspose's documentation has a web application security section, which contains code snippets pertaining to how this can be achieved by implementing the *IResourceLoadingCallback* interface[1].

LED-01-007 WP2: Denial-of-Service in HTML to PDF conversion *(Medium)*

Note: *This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

Company owners are able to create HTML report templates, which are converted to PDF files and sent to stakeholders, to inform them about the current development of the company. To ensure a malicious attacker cannot specify arbitrary HTML, a markdown library is utilized on the client side as well as the backend to limit the available HTML tags and prohibit the execution of malicious JavaScript. Nevertheless, testing corroborated that it is possible to cause a denial of service within the application. Not only are remote images supported by the markdown library, but the server side utilizes a Chromium browser instance to convert the HTML structure into PDF.

The issue was verified by including a remote SVG image, which consumes an excessive amount of RAM when it is rendered in Chromium.

Steps to reproduce using a non-blink-based browser (e.g., Firefox):

1. Login with a company-owner user.
2. Go to the specific company > *Investor Relations* > *Reports.*
3. Click on *"Template".*
4. Add the following markdown to insert the remote image SVG content displayed below:

   ![text](**http://yourdomain/pov.svg**)

5. Ensure the changes are saved. Close the editor.
6. Click on *"Create Report".*
7. Click on the newly-created report > *Preview* > *Publish.*
8. The remote Chromium will consume all available RAM until it crashes, thereby rendering the web application unavailable.

[1] https://docs.aspose.com/words/java/web-applications-security-when-loading-external-resources/

Filename:
poc.svg

File content:
```
<?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/>
 <node/>
 <node/>
 <node/>
 <node/>
 <node/>
 <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">
<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:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
</xsl:for-each>
 </xsl:template>
 </xsl:stylesheet>
</root>
```

It is highly recommended to transfer this feature to a dedicated server or docker instance. This would allow administrators to configure a maximum level of system resources that the feature would be able to consume without influencing the stability of the application itself. Additionally, in the eventuality a vulnerability occurs allowing the execution of arbitrary JavaScript code in the Chromium browser, one would be able to guarantee that the attacker must pass additional security boundaries before attempting to access customer- or system-relevant resources.

LED-01-008 WP2: HTML injection in HTML to PDF conversion via name *(Low)*

***Note:** This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

During the discovery of the issue reported via LED-01-007, the code was assessed for additional injection vulnerabilities during the conversion of the HTML report structure to PDF. This testing confirmed that the *footerTemplate* HTML property includes the current user-name without any HTML encoding. By specifying HTML elements in the user-name, an attacker is able to render arbitrary HTML on the backend.

The impact of this vulnerability is highly reduced as Skia - the component in Chromium handling PDF rendering - does not only prohibit the loading of local files in the footer, but also disables the support for JavaScript.

Steps to reproduce:

1. Login with a company-owner user.
2. Change the name of the account to the following text:

    1. *Max <b>BOLD</b> name.*
3. Go to the specific company > *Investor Relations* > *Reports.*
4. Click on *"Create Report".*
5. Click on the newly-created *Report* > *Preview* > *Publish.*
6. Open the *"data room"* > *"Investor Reports"*.
7. This should contain a PDF with the name set in *Step 4.*
8. View the PDF.
9. The footer of the PDF should display a bold "BOLD" text.

Affected file:
ledgy-app-master/modules/server/startup/reports/lib/htmlToPdf.ts

Affected code:
```
const footerText = `Created ${date} by ${userName} and sealed by Ledgy.com`;
[..]
const watermark = `[...]<div style="text-align: center; color: ${muted}">$
{footerText}</div>`;

const footerTemplate = `
<div style="${footerStyle}">
${hideWatermark ? '' : watermark}
<div style="${
hideWatermark ? pageNumberStyle : 'white-space: nowrap;'
}">Page ${pageNumber} of ${totalPages}</div>
</div>`;

const page = await browser.newPage();
await page.setContent(html);
const pdf = await page.pdf({
printBackground: true,
displayHeaderFooter: true,
format: 'A4',
headerTemplate: ' ',
footerTemplate,
});
```

It is highly recommended to HTML encode any user-controlled variables when they are included in HTML structures. This ensures that no arbitrary HTML tags are rendered in the backend, which could potentially harm the security of the application itself.

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

Miscellaneous Issues

This section covers any and all noteworthy findings that did not lead to an exploit but might assist an attacker in successfully achieving malicious objectives 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.

LED-01-002 WP1: Phishing via modified *Host* header (*Info*)

Note: *This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

Testing confirmed that a selection of application endpoints can be influenced during a redirection by providing a *Host* header that directs to an attacker-controlled website. An adversary can gain control over this header by luring users of an outdated Adobe Reader version into visiting a maliciously-prepared website, for instance.

This might be leveraged by malicious attackers for phishing purposes, whereby a victim is manipulated into visiting an attacker-controlled website that redirects to a crafted *login* page. On the one hand, the overall severity of this issue can only be considered *Info* due to a greatly-limited range of exploitability. On the other hand, the attack does remain possible and is, therefore, worth fixing. This issue's PoC request, related to one of the affected endpoints detected during this assessment, can be consulted below:

Request:
```
curl -i -X 'POST' -H 'Host: cure53.de' 'https://app.ledgy.com/sockjs/\.'
```

Response:
```
HTTP/2 302
cache-control: private
content-type: text/html; charset=UTF-8
referrer-policy: no-referrer
location: https://cure53.de/sockjs//
content-length: 223
date: Tue, 09 Mar 2021 13:47:29 GMT
alt-svc: clear

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="https://cure53.de/sockjs//">here</A>.
</BODY></HTML>
```

It is recommended to validate the *Host* header more strictly or ensure that it is completely free from user-input. It is advisable to conduct additional investigations toward the cause of the reflected header value by the backend. This behavior can also be caused by some kind of weakness in the application stack's implementation.

LED-01-003 WP2: NodeJS backend supports *Content-Encoding* header (*Info*)

Note: *This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

During the assessment phase, the discovery was made that the Ledgy application supports the *Content-Encoding* header in HTTP requests, despite the fact that it is not currently utilized by the application. This header allows compressed HTTP bodies to be sent, which the backend automatically deflates before it is processed. The impact of this issue is reduced as the default behavior applies a limit on the amount of resources that can be consumed during decompression. This in effect prevents a denial of service. Nevertheless, an attacker could abuse this feature to send multiple minor HTTP request bodies, which would strain the backend.

cURL example:
```
curl -v -H 'Content-Type: application/pdf' -H "Content-Encoding: deflate" -H
'Authorization: aZTwKrAcQsL1Z3_7MXyPGMeEv0wWa3K0Vc7VIMbCr6N' --data-binary
@deflate_compressed.pdf 'https://app.ledgy.com/internal/api/upload/doc?
companyId=j5PeDdwSW5eymYbMj&name=formc333312312233c_2.0'
```

Create *zlib* compressed file:
```
cat example.pdf | zlib-flate -compress > deflate_compressed.pdf
```

In the eventuality this feature is not utilized, one can recommend disabling the support for compressed HTTP bodies. This can be achieved by setting the *inflate* option, which is available in the Express body parser middleware[2]. By disabling this feature, one can guarantee that an attacker would not be able to instigate a compression bomb attack against the backend through repeated sending of compressed HTTP bodies.

[2] http://expressjs.com/en/resources/middleware/body-parser.html

LED-01-004 WP1: Client-side input-checks not enforced by backend *(Info)*

***Note:** This issue only affects the staging system used in this test. The production system appears not to be affected by this problem.*

***Note:** This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

The potential to insert arbitrary input into a selection of application fields was discovered. For instance, one can assign a negative value in the *numberOfDays* property for the *collaborators.grantHelp* method. Arbitrary input can then be accomplished by deploying an intercepting proxy and altering the parameters once the client-side checks have been applied. It is worth noting that new application components could trust these forms and use the information in an insecure manner, thereby elevating the risk toward a more severe vulnerability. An example of the current behavior can be observed below:

Request:
```
POST /sockjs/459/_xuuap75/xhr_send HTTP/1.1
Host: beta.ledgy.com
Accept: */*
Content-Type: text/plain;charset=UTF-8

["{\"msg\":\"method\",\"method\":\"collaborators.grantHelp\",\"params\":
[{\"companyId\":\"th9TiSmu7RyvE44j2\",\"numberOfDays\":-5}],\"id\":\"107\"}"]
```

Note that the backend still processes the request with negative values, despite the fact that the schema has thrown an error.

The application will correctly accept a request with a negative value for the "*numberOfDays*" parameter; pertinently, the application's web interface does not allow the user to set a value of this nature.

A client-side input validation should only serve to assist the user in following the predefined rules. A server-side input validation should enforce the rules and reply with an error message when said rules are violated. In addition, one should corroborate if any other forms within the web application employ input validation that exists on the client-side exclusively.

LED-01-005 WP1: Improper output for "support access" functionality *(Info)*

***Note:** This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

During the assessment, the discovery was made that the "*Give Ledgy support access to ...*" functionality of the application on the "*General*" page can accept more than one user value. This is possible by calling the "*collaborators.grantHelp*" method directly - bypassing the client-side checks in the web interface - but nevertheless the web interface will only display the first entry added. In this way, the existence of potential long-term access (which this feature provides) may be hidden from the user.

To mitigate this, one can recommend rejecting repeated user requests for the "*collaborators.grantHelp*" method if it is already in an active state - and also validating the user input on the server side as described in LED-01-004 - in order to eliminate negative number entry.

LED-01-006 WP1: Absent "session" regeneration mechanism *(Low)*

***Note:** This issue has been flagged as a false alert after a debriefing meeting.*

The application implements a stateless session design that does not fetch user information from the database on every request. Whilst this method helps to reduce resources, unfortunately an erroneous instance occurs when critical permissions are changed – for example, when the user role is removed or when a user's privileges are disabled. In these instances, the "session" does not get refreshed, and the user can continue to use the WebSocket connection to access privileged endpoints, even though one should no longer have access.

The recommendation can be made to renew or regenerate used sockets after any privilege-level amendment within the associated user account.

LED-01-009 WP1: HTTP security headers not set for error pages *(Info)*

***Note:** This issue was verified as properly fixed in March 2021 by the Cure53 team, the problem no longer exists. The fix was verified by inspecting a pull request.*

The Ledgy web application correctly utilizes common HTTP security headers in its responses. However, the discovery was made that these headers are not set for HTTP error pages. Although this cannot be considered an immediate security risk, a malicious attacker could abuse such behavior - in combination with alternative vulnerabilities - to achieve JavaScript execution on the platform.

PoC request:
```
GET /%*0 HTTP/1.1
Host: app.ledgy.com
[...]

HTTP/1.1 400 Bad Request
Server: nginx
Date: Sun, 14 Mar 2021 18:53:23 GMT
Content-Type: text/html
Content-Length: 150
Via: 1.1 google
Alt-Svc: clear
Connection: close

<html>
```

It is recommended to deploy the omnipresent HTTP headers for error pages additionally. This can be deployed on a reverse proxy server to ensure every HTTP response utilizes the headers properly.

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

Conclusions

The impressions gained during this report - which details and extrapolates on all findings identified during the CW10 testing against the Ledgy web UI, backend and API by the Cure53 team - will now be discussed at length. To summarize, the confirmation can be made that the applications under scrutiny have left a good impression, mostly owing to the low volume of issues detected.

The audit conducted highlights that the application consists of a limited volume of technology stacks; their subsequent implementation naturally restricts any attack surface on the whole.

Despite extensive efforts and exemplary coverage from the Cure53 testers, no noteworthy findings were documented in Meteor, which contributes to the positive impression gained on the whole. With regard to input validation, existing endpoints have been identified and verified for the handling of various function and method calls in order to unearth any logical issues or input-validation issues. Similarly, heightened scrutiny was placed toward common injection vulnerabilities. The user-registration and password-recovery flow were tested for potential issues that could weaken or bypass the authentication flow. Additionally, ACL separation was deemed exceptionally secure. No erroneous instances were discovered in this regard despite extensive testing. The deployed schema validation of user-controlled JSON structures and their associated properties, in particular, prevent common attack vectors related to type confusions via parameters.

With considerable focus toward standard issues surrounding modern web applications, Cure53 investigated the client-side code and the application’s functionality for the presence of XSS attacks and similar input-manipulation issues. No issues of this nature were detected. The application makes a stable impression on the client-side, which is a very positive indication rarely found during audits of this nature. The client-side benefits from proper implementation of the React framework while avoiding the usage of the *dangerouslySetInnerHTML* property. Additionally, the client-side barely utilizes the *postMessage* functionality, which often introduces DOM XSS vulnerabilities in otherwise secure applications. Lastly, the possibility of causing script execution via maliciously uploaded documents was completely avoided by utilizing a third-party domain to host these files.

One of the largest attack surfaces present in the application owes to the parsing of user-controlled files or structures. One can deduce here that the developers had considered the threat of malicious files, as certain microservices are deployed which handle the conversion process. However, as the majority of impactful issues discovered during the

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

assessment are related to malicious files and the methods by which they are processed, additional separation and hardening should be deployed to isolate these components from the primary Ledgy application. This would also subsequently ensure that internal files and systems remain safeguarded from malicious attacks.

The fact that only one single *Medium*-security risk was unearthed indicates that the tested application scope items have a reasonably-stable security posture. One can safely assume that secure development was a key component of Ledgy's architectural and design paradigms.

Nevertheless, the handful of minor flaws identified indicates that there is indeed leeway for improvement and targeted hardening. One should also note here that even seemingly-minor issues tend to accumulate, which could potentially lead to new attack chains being formed. Therefore, even minor flaws should be mitigated to further harden the security of the application.

In summation, one can argue that the outcome of this report confirms the development team's commitment to maintaining security features with due diligence and adherence to best practice. Once again, despite extensive deep-dives and exemplary coverage toward a plethora of application features by the Cure53 testers, no serious issues were detected. To conclude, the application makes a stable impression in relation to its core security constitution. Cure53 believes that the project maintains a strong stance towards the primary objective of delivering a secure foundation to customers.

Cure53 would like to thank Timo Horstschäfer, Jules Henze, and Marius Colacioiu from the Ledgy team for their excellent project coordination, support and assistance, both before and during this assignment.