**Security and Privacy Assessment of Landmarks API**

Prepared by: Adam Plumer

**Introduction**

Security and privacy are paramount in an age where most people use, and provide sensitive information to online, publicly-accessible services. It’s almost impossible to go more than a week in the modern world without another major vulnerability discovery or security breach of a major firm. Within the two days prior to this report, there were breaches of the Mexican Voter Database, and the Society for Worldwide Interbank Financial Telecommunication (SWIFT) system.

Thus, this report aims to proactively report on the breaches and vulnerabilities of a low-scale, low-exposure product, in an effort to highlight the need for defensive programming, and secure systems in all facets of the technological world.

In this report, a *vulnerability* may be seen as any security flaw in the given product[[1]](#footnote-1), which is an API for retrieving historical landmarks listed with the United States National Parks Service (NPS) within one mile of your present location. This report will mostly focus on the server-side application, which is responsible for storing and display user data based on HTTP GET requests, as well as an HTTP POST API for inputting data, as well as retrieving landmark information.

**Methodology**

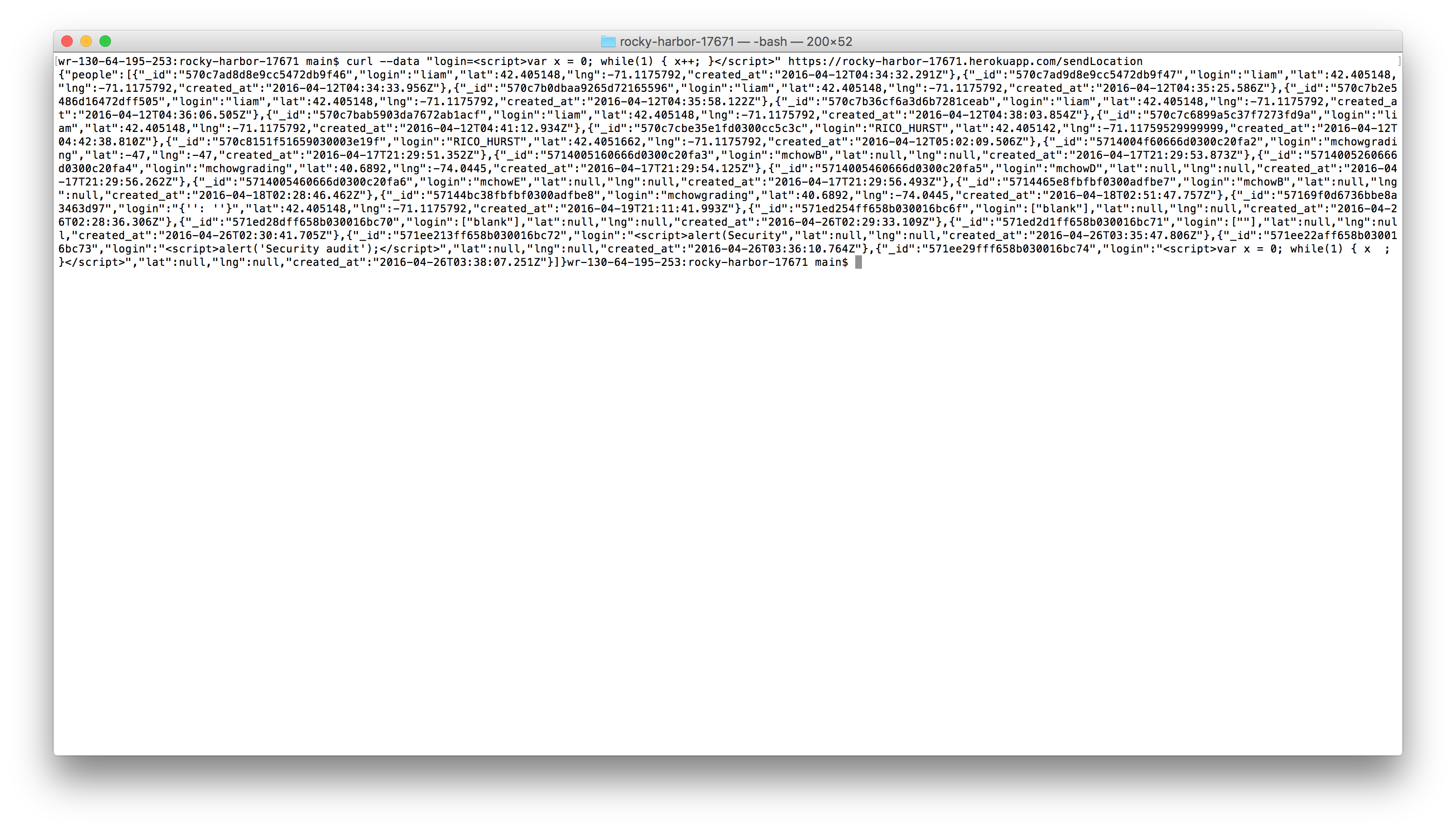
This section starts with a caveat, namely that many of the implementation details are known about this API. First and foremost, the execution paths for the web application were all known in advance. Second, it was known during black box testing that the web server employs a Node.js framework with a MongoDB database server. With that being said, query operators for Mongo were used to reveal vast amounts of user information from the database. The curl suite was also used to inject JavaScript data into the database, which was then rendered on the target website. Finally, privacy issues were observed based on the formatting of the website itself, and the services offered.

**Abstract of Findings**

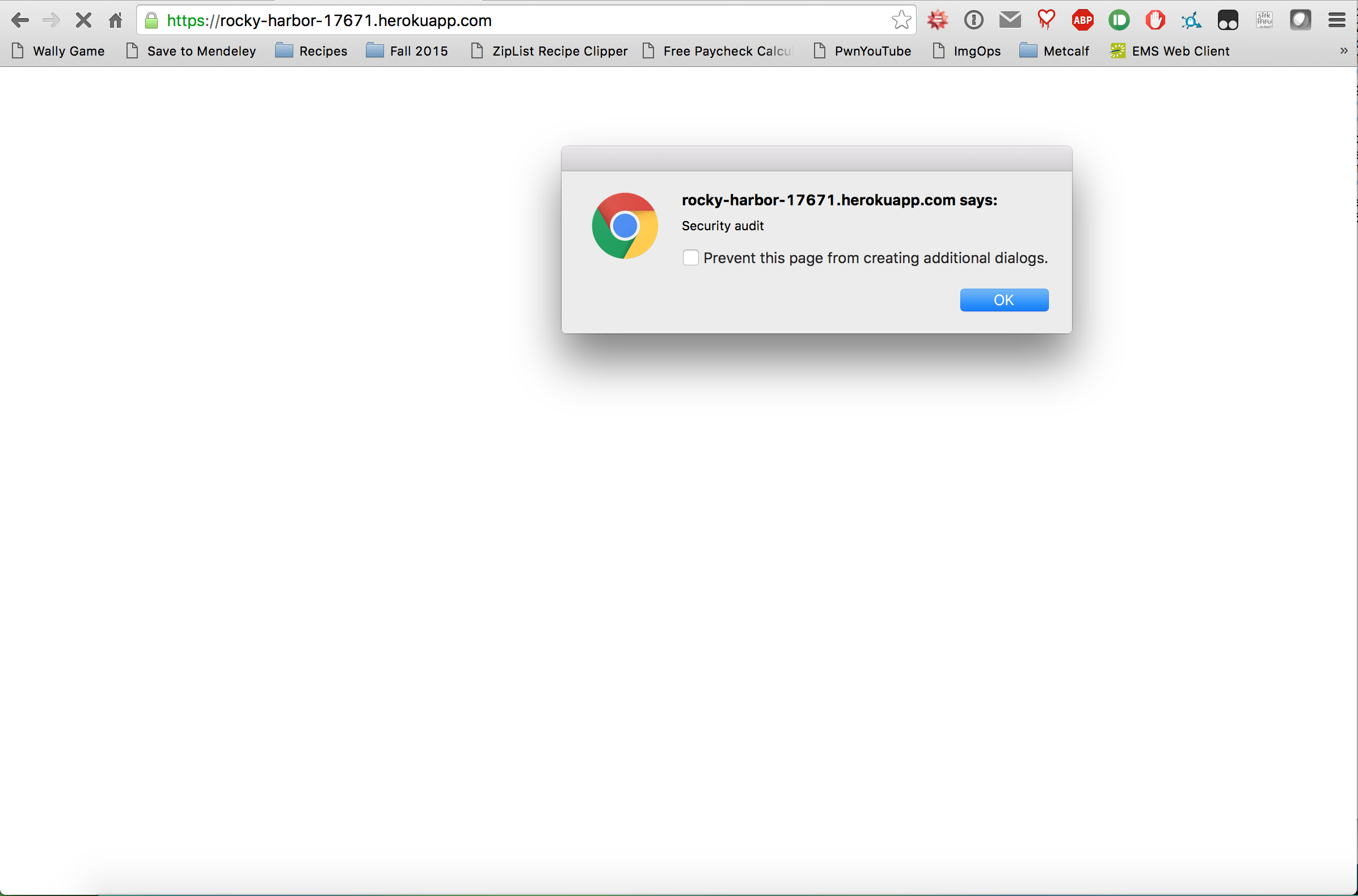
* User input is not parsed correctly, allowing injection of JavaScript into an HTTP POST request, thus being stored in a database, and rendered on an HTML page
* User input is not parsed correctly, allowing injection of HTML code into an HTTP POST request, thus being stored in a database, and rendered on an HTML page
* HTTP parameters are not parsed correctly, allowing an exploitation of the Mongo query operators to reveal the contents of the entire database
* Server details (namely the username and password for the Mongo database) are stored in plaintext in the Node.js server file

**Issues Found**

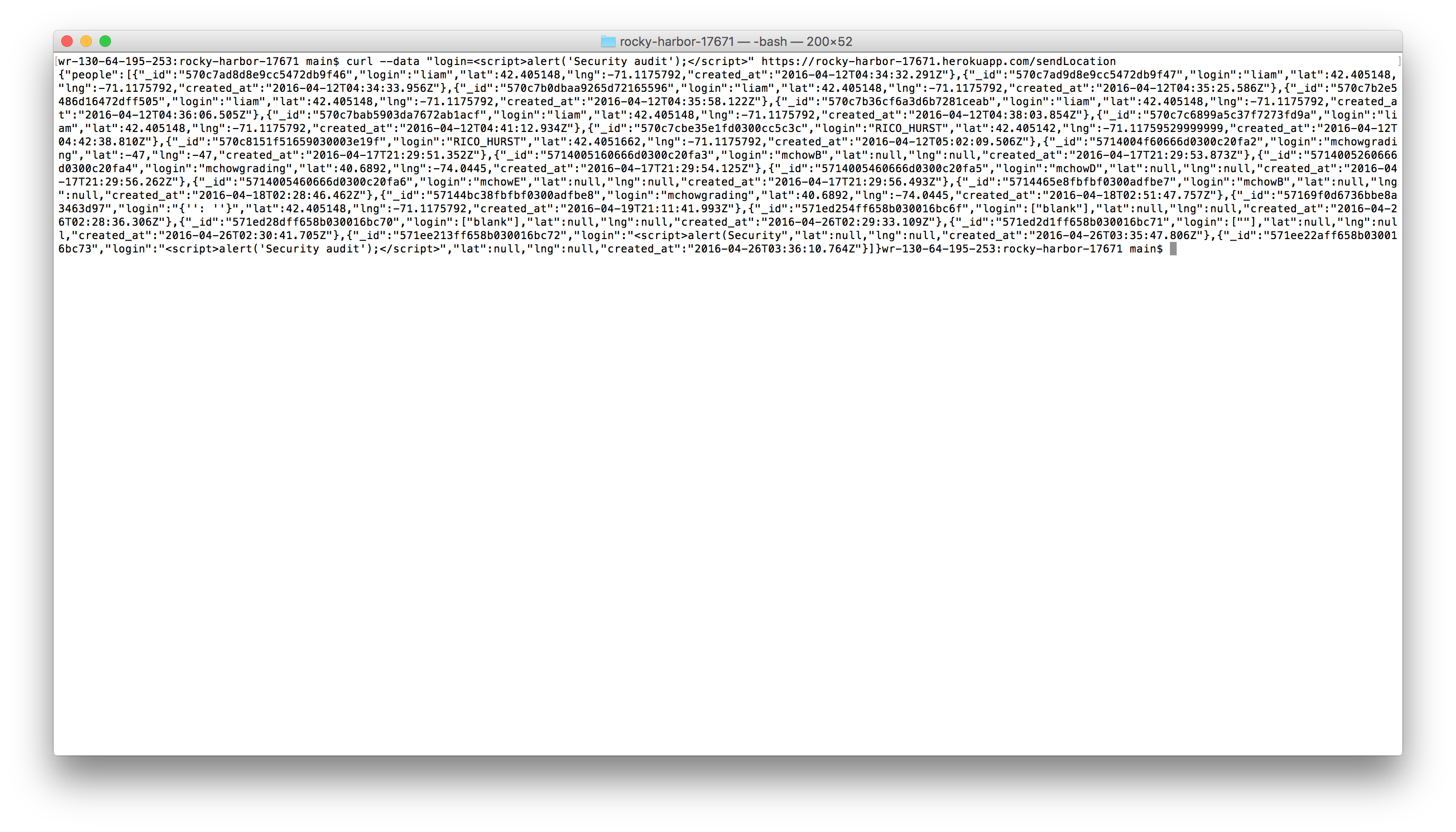
1. Cross-Site Scripting (XSS) via HTTP POST – ***high severity***
   * Location of issue: ***sendLocation*** execution path
   * Using the curl suite to submit an HTTP POST request, one field of the request’s parameters can be altered to include JavaScript text. This text then gets stored verbatim in the server-side database, and then rendered when retrieved
   * This is a serious breach, because arbitrary scripts could track user movements, or act maliciously towards the user without the owner’s knowledge. This could lead to the user entering information they otherwise wouldn’t, or redirect the user to a website that could compromise a user’s system further
   * This attack can also be used to render the website useless as a form of Denial of Service (DoS). This attack was inspired by Gabriel Joseph’s security report from 2013
   * Images that show the attack and proof of concept can be seen in Figures 1 through 4 on the following pages
   * **Recommended Resolution:** parse user input using the sanitize() function:
     1. e.g. 🡪 login: sanitize(req.body.login)

****

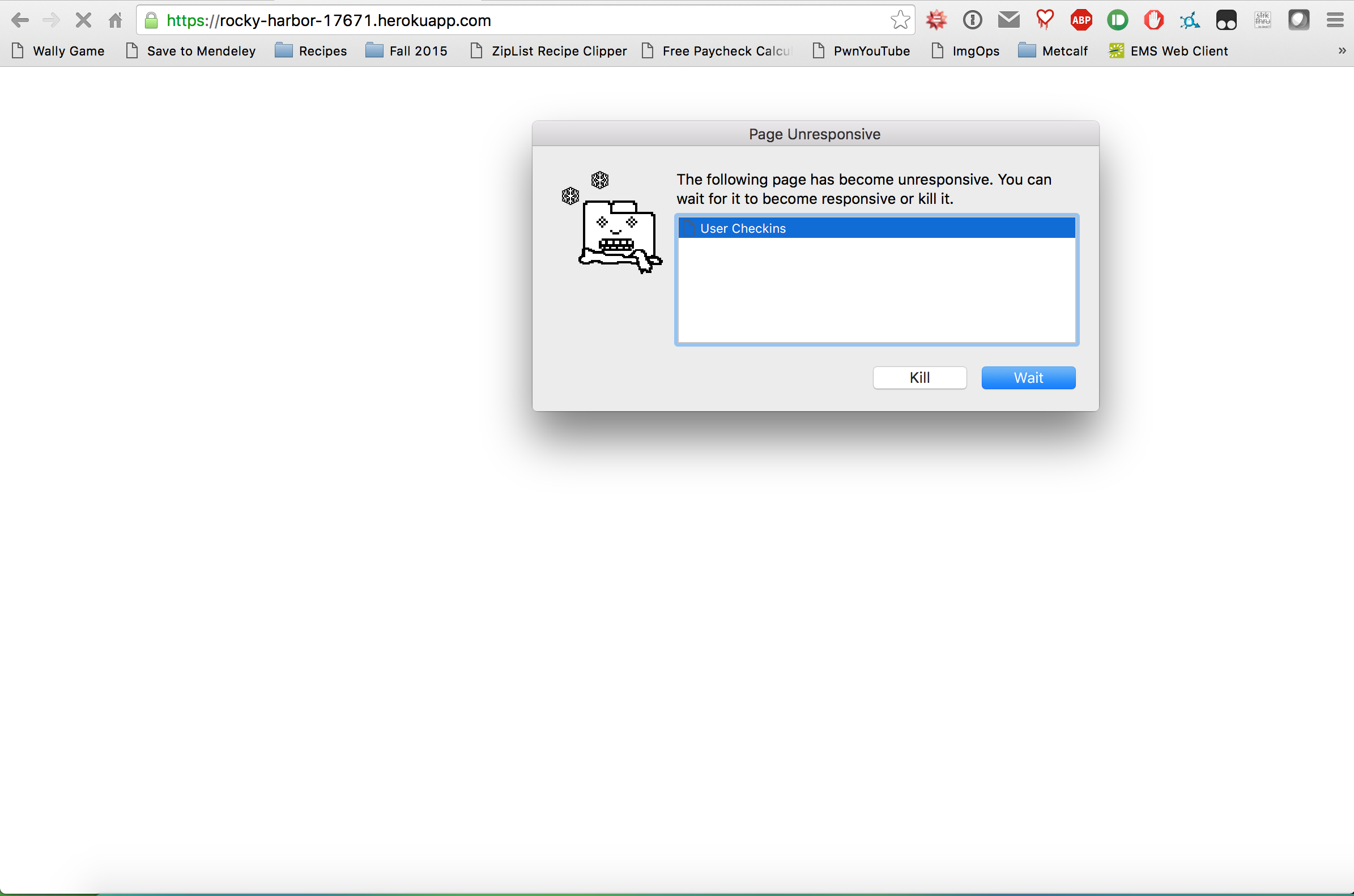
***Fig 1. Curl request to submit arbitrary JavaScript to the website***

****

***Fig 2. Proof of concept for attack shown in Fig. 1***

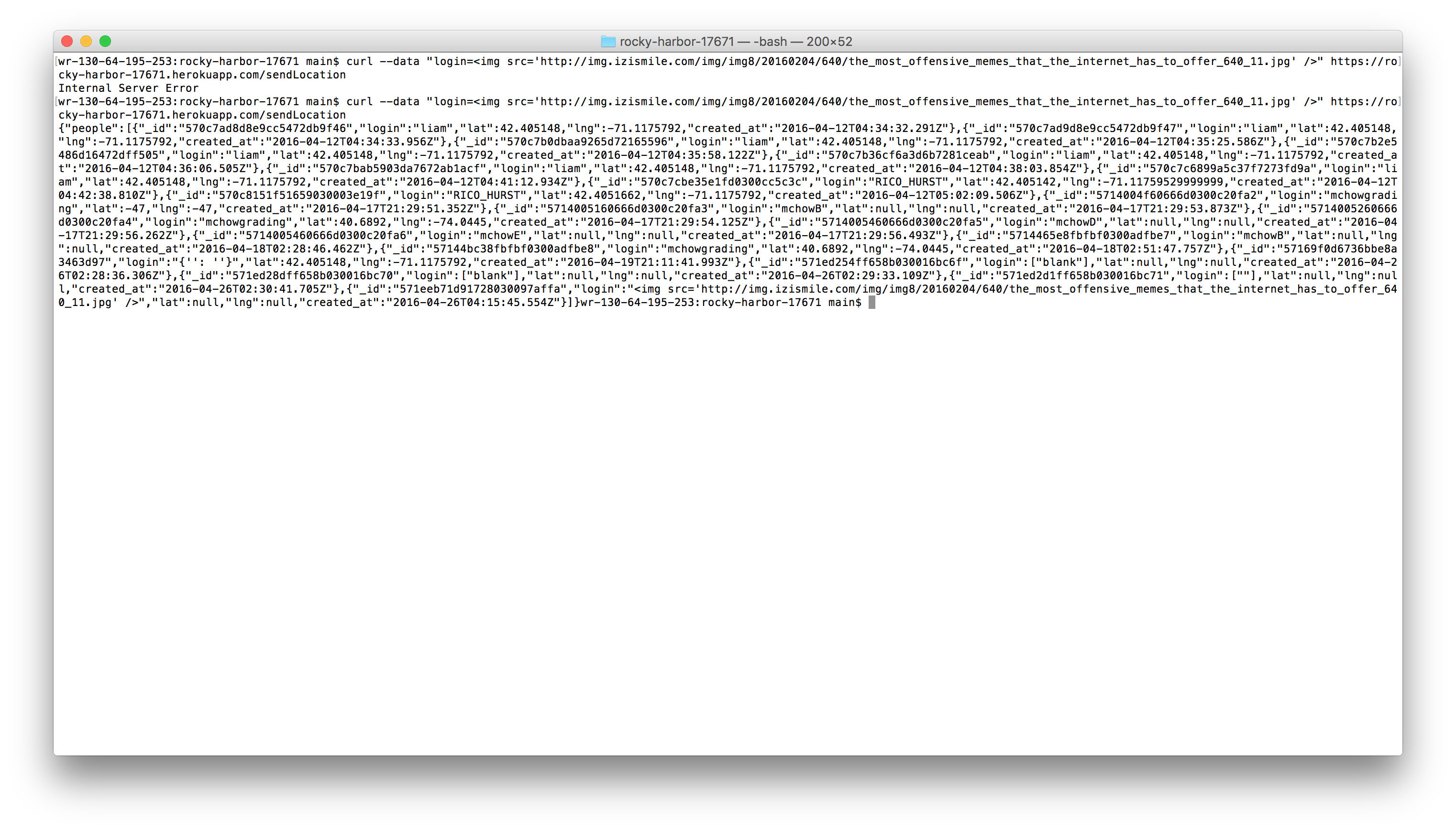
****

***Fig 3. Curl request to perform DoS attack on server***

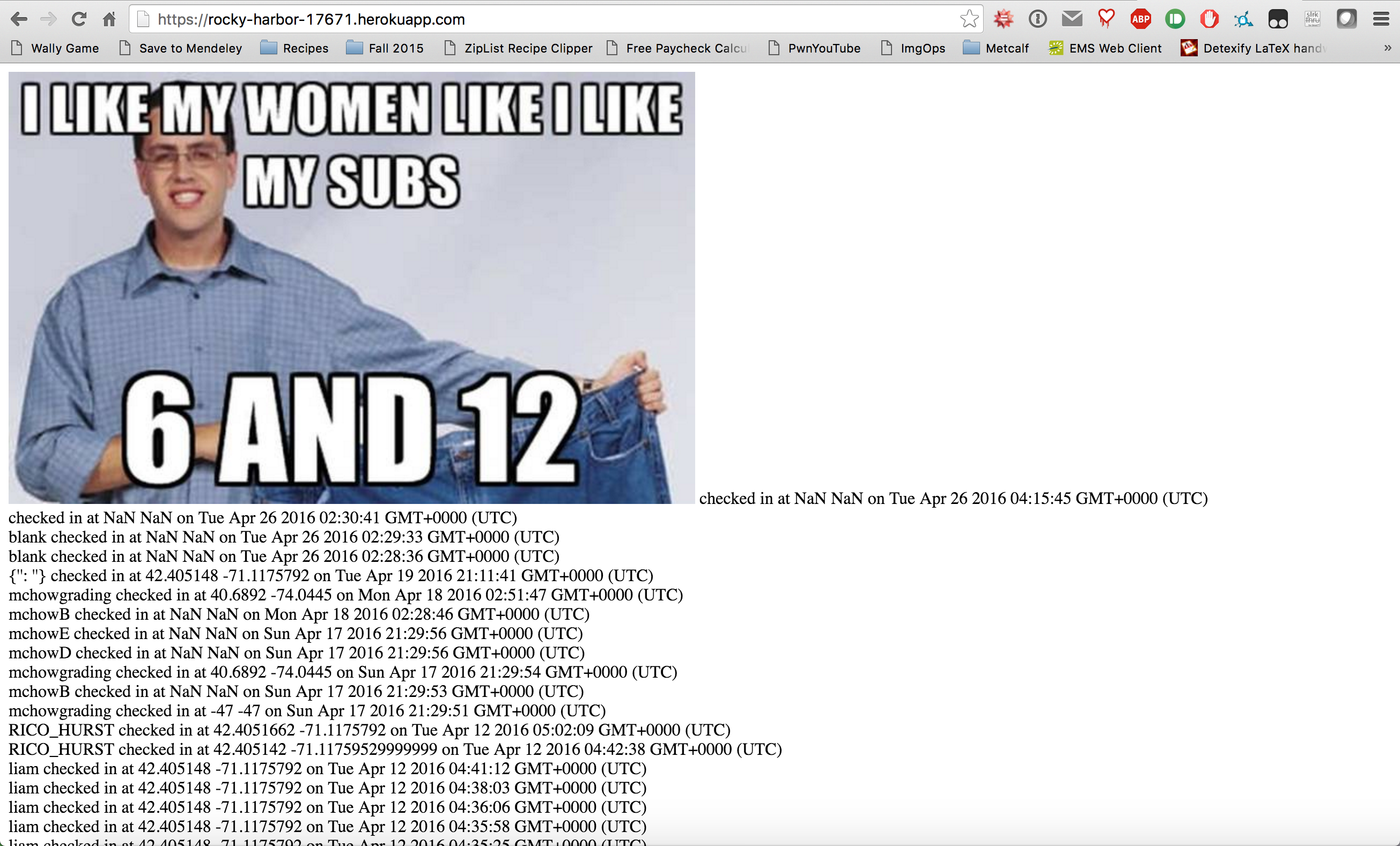
****

***Fig 4. Proof of concept for attack shown in Fig. 3***

1. HTML Injection via HTTP POST – ***medium severity***
   * Location of issue: ***sendLocation*** execution path
   * Using the curl suite to submit an HTTP POST request, one field of the request’s parameters can be altered to include HTML text. This text then gets stored verbatim in the server-side database, and then rendered when retrieved
   * This is a serious breach, because arbitrary scripts could track user movements, or act maliciously towards the user without the owner’s knowledge. This could lead to the user entering information they otherwise wouldn’t, or redirect the user to a website that could compromise a user’s system further
   * This attack can also render artificial content that may be offensive to users or otherwise against the intended purpose of the product
   * Images that show the attack and proof of concept can be seen in Figures 5 and 6 on the following page
   * **Recommended Resolution:** parse user input using the sanitize() function:
     1. e.g. 🡪 login: sanitize(req.body.login)

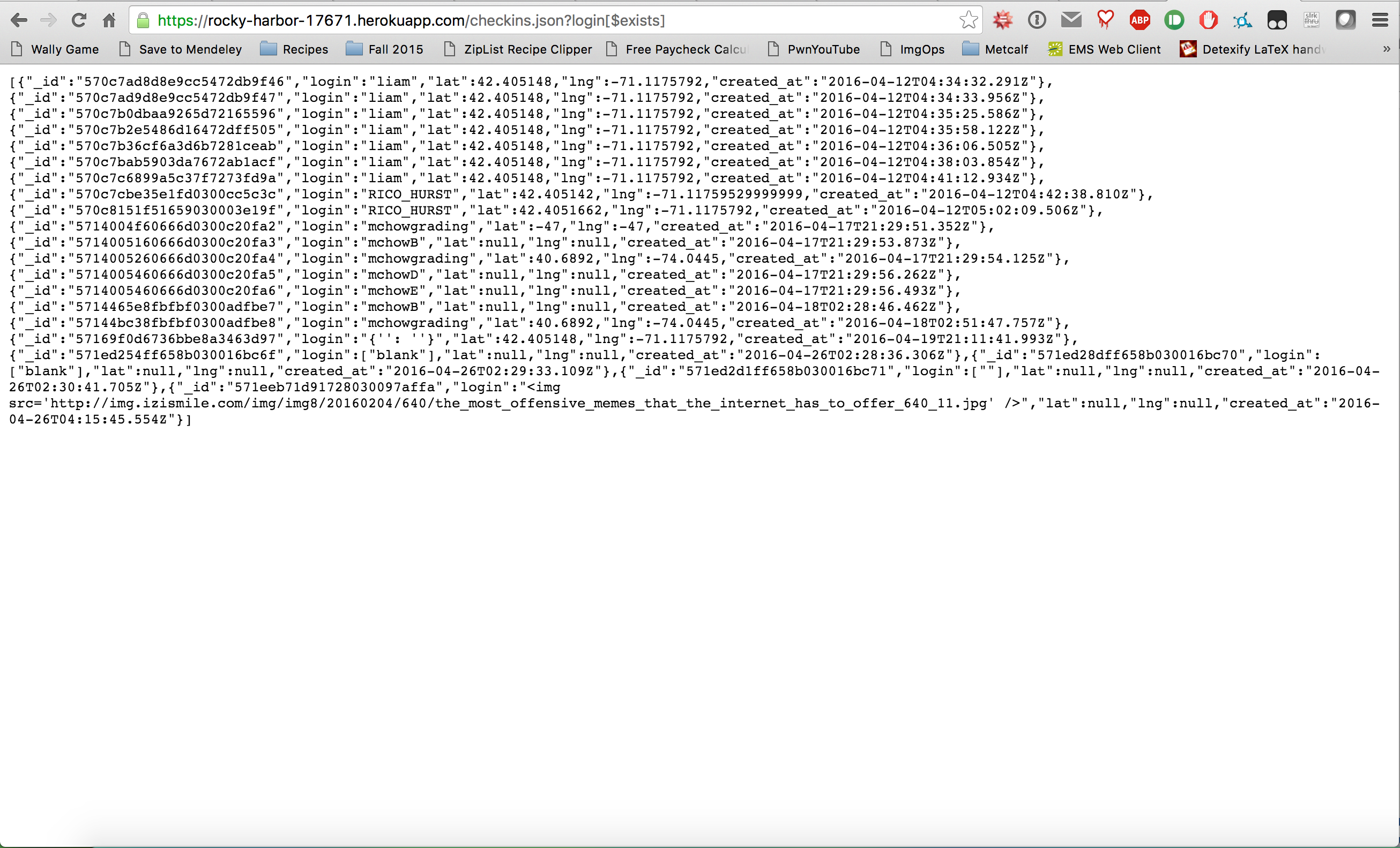


***Fig 5. Curl request to inject HTML code onto a website***

******

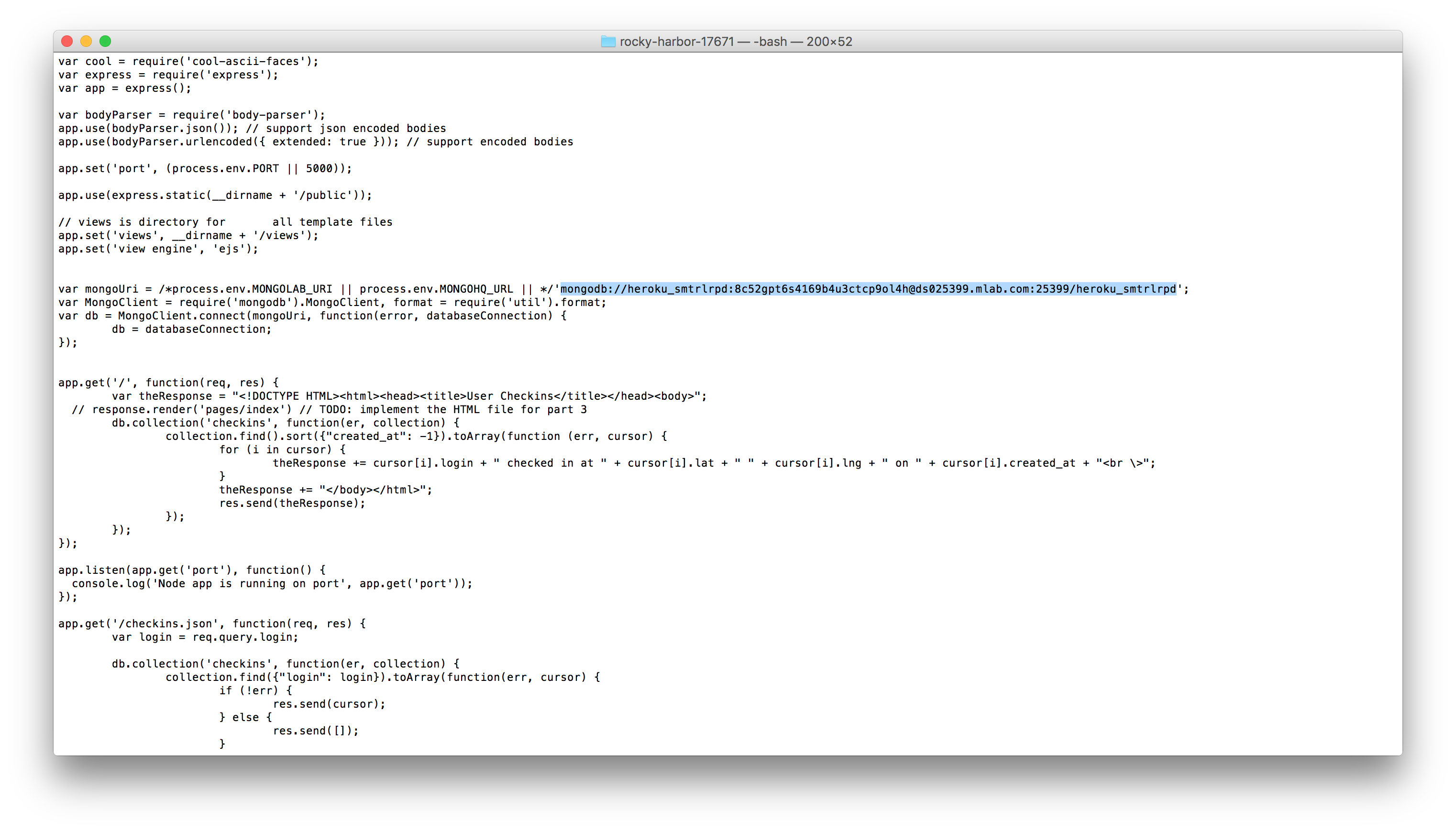
***Fig 6. Proof of concept of attack described in Fig. 5 depicting an offensive meme***

1. MongoDB Query Operator Exploit via HTTP GET – ***high severity***
   * Location of issue: ***checkins.json*** execution path
   * Using simply the address bar of any web browser, the entire contents of the database can be displayed as a JSON object on the webpage
   * This is an extraordinarily serious breach because the only details that should be display are the details intended to be displayed by the page administrator
   * This exploit overrides any previous intentions, and results from a lack of parsing the request parameters (see exploits two and three for the fix)
   * To employ this exploit, one simply needs to enter as the parameter:
     1. URL/checkins.json?login[$exists]
   * This will display every entry that exists in the database, since it is a valid Mongo query operation



***Fig 7. Proof of concept of Mongo query operator exploit***

1. External Database Credentials Leak – ***high severity***
   * Location: ***index.js*** file in the Node.js configuration directory
   * The credentials for the Heroku mLab MongoDB are exhibited in plaintext in the Node.js server file
   * This represents an obvious breach, since anyone with this file can access the database in perpetuity, unless the owner suddenly decided to change it after reading this report
   * To resolve this issue, simply don’t store the credentials in plaintext in the file, or use a local database with no credentials instead



***Fig 8. Proof of data leak in index.js file***

**Conclusion**

The issues described above are serious security and privacy concerns, as they represent a breach in basic database protection, and user data protection. Almost all issues described above could be resolved by parsing user input. However, uncovering these issues took non-zero time, and thus may be worthy of a non-zero reward. Security is important, users don’t want to use a service that may publicly show all of their information, such as location data, to any security sleuth that comes looking for it

1. https://rocky-harbor-17671.herokuapp.com [↑](#footnote-ref-1)