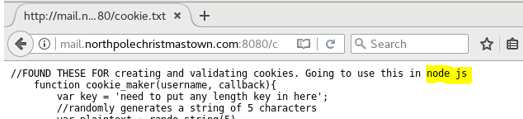
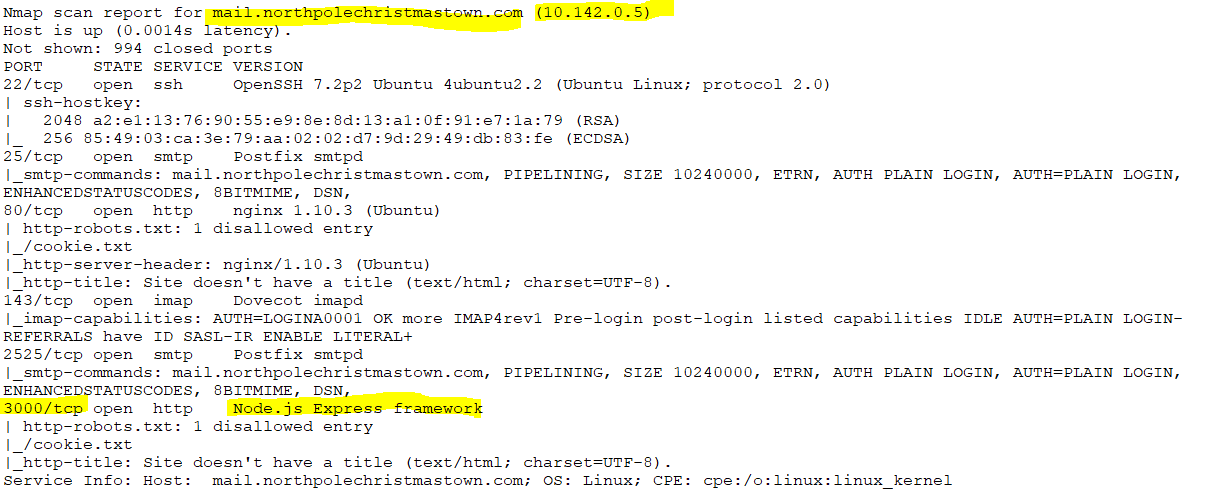
# Elf Web Access--Encryption Gone Wrong Part 3--It’s all about the cookie

## Question 1

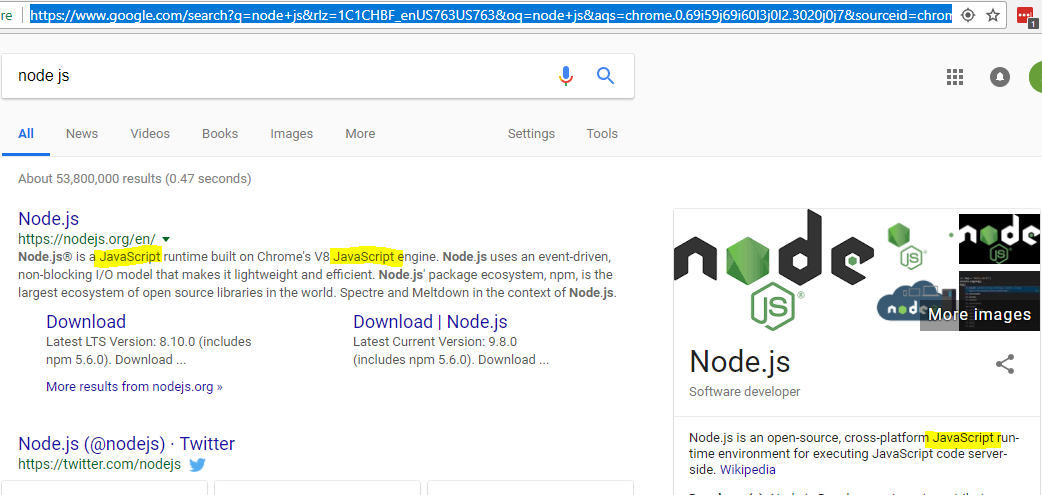
The EWA site probably runs on Node.js.



The Nmap scan also helps us.

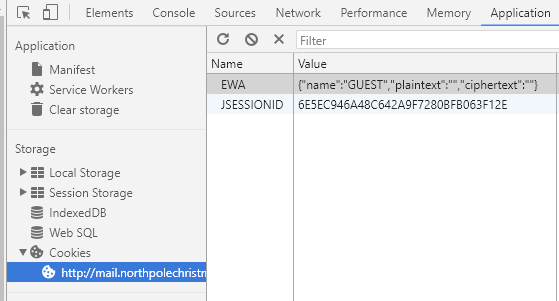


## Question 2

Node.js runs JavaScript server-side. 

## Question 3--Chrome

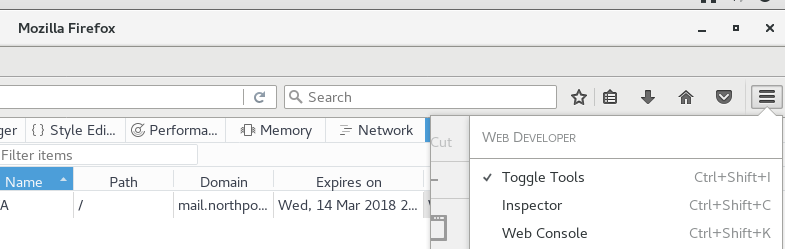
To view the cookies in Chrome, select More Tools -> Developer Tools, and view the Application tab.



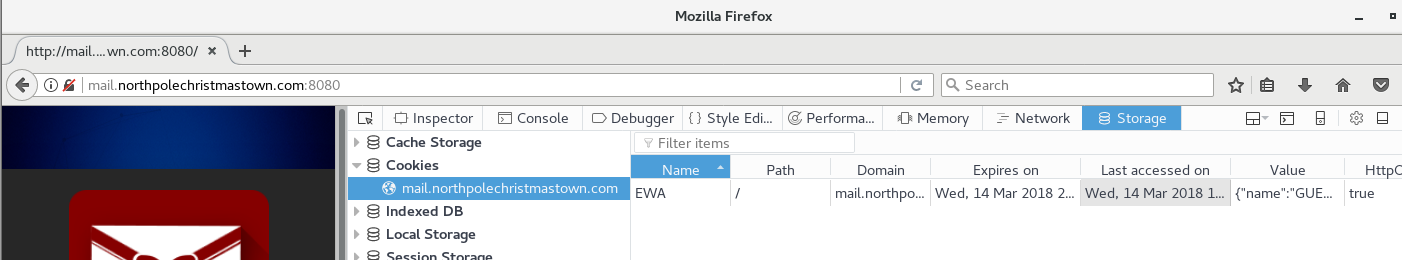
There is a cookie named EWA that has the format {“name”:”GUEST”,”plaintext”:””,”ciphertext”:””}. Note the repeated quotes after plaintext and ciphertext. That is where the plaintext and ciphertext values will go.

## Question 3--Firefox

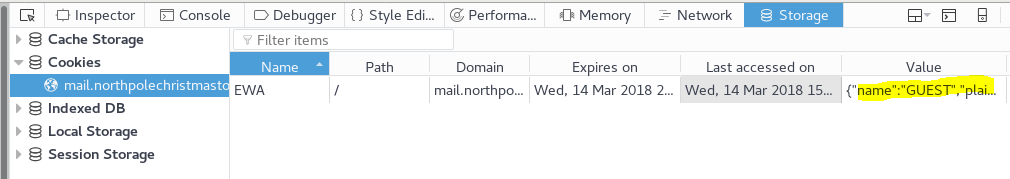
To access developer tools in Firefox, select Developer -> Toggle Tools



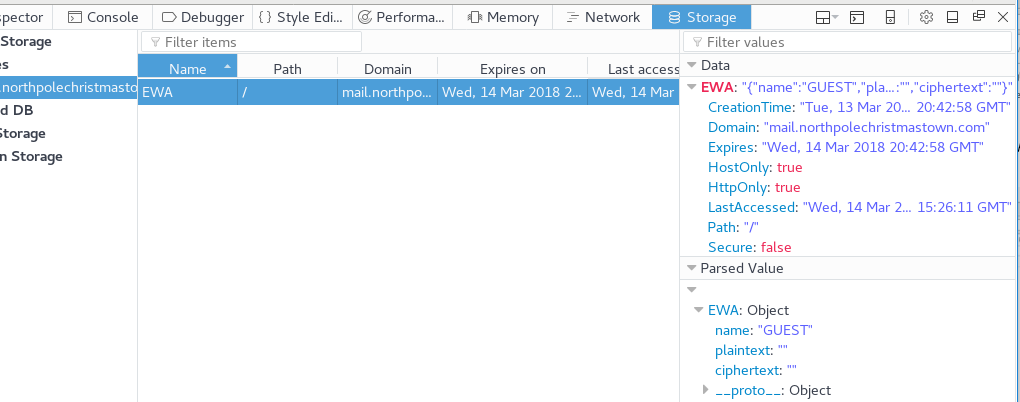
Remember that the Storage tab must be enabled (see Elf Web Access Part 2). Select the Storage tab, and then Cookies.



The cookie is here.

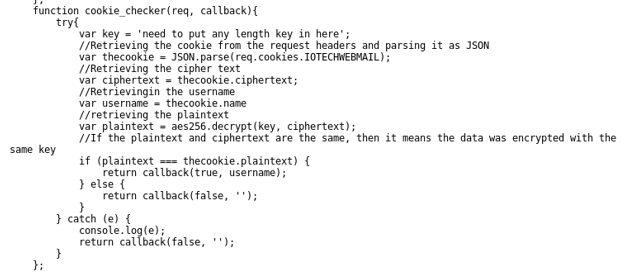


You can see more detail by clicking in the Value field.



## Question 4

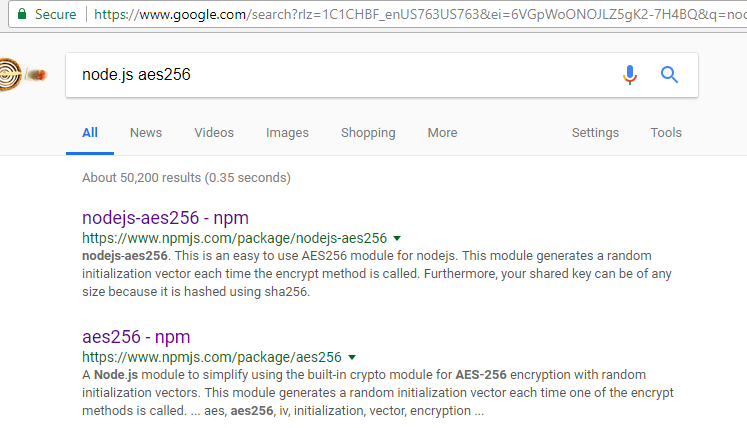
Here is the server-side code for the cookie checker, assuming it is the same as what is in the cookie.txt file. It does match the cookie we saw in the browser. The cookie name has been changed from IOTECHWEBMAIL to EWA, but it has the same attributes: name, plaintext, and ciphertext.



The code is simple. It sets plaintext = aes256.decrypt(key, ciphertext). If the decrypted plaintext matches the plaintext value in the cookie, it assumes the cookie is valid. It returns “true” and returns the cookie name value as the user name. Note: This is an awful way to do cookies. Don’t do this on your web sites!

## Question 5

If you look long enough, and are lucky, you should come across two aes256 modules at [www.npmjs.com](http://www.npmjs.com).



At the time of the contest, both exhibited the vulnerability we need to make the attack work. The [second one, by James M Greene](https://www.npmjs.com/package/aes256), was fixed just after the contest so that’s not useful for us. The [first one by Jay Svoboda](https://www.npmjs.com/package/nodejs-aes256) has not been fixed, so we will use it. There is a link on the [www.npmjs.com](http://www.npmjs.com) site that takes us to the aes256.decrypt [source code on GitHub](https://github.com/jaysvoboda/nodejs-aes256).

## Question 6

According to Pepper’s hints 3 and 4, Alabaster told her that he uses the first 16 bytes of the ciphertext as an Initialization Vector (IV), so he removes that and decrypts the remaining bytes. Pepper then asks what would happen if the ciphertext was only 16 bytes long. After the IV was removed there would be nothing to decrypt. Hmmm. Maybe making a cookie with a 16-byte cipher text might helpful.

## Let’s Review

Our goal is to make a “specially crafted” cookie that will pass the cookie checker function in the JavaScript code running on the EWA server regardless of the key the server uses. We’re still assuming that the cookie.txt file we found is a good indication of how the EWA server works.

So, here is what we know. The cookie\_maker function generates a random 5-character string for plaintext. It encrypts that string and calls it ciphertext. It adds a user name to the cookie, but the user name is independent of the plaintext and ciphertext (bad idea). When a user successfully logs in, this cookie is sent to the browser.

The browser returns the cookie in every transaction. On the server, the function cookie\_checker gets the cookie through the parameter req, which has the cookie values as a property. The server decrypts the cookie’s ciphertext, and if the result matches the plaintext in the cookie, considers it a valid transaction. Judging by the cookie in the browser, our cookie is probably req.cookies.EWA instead of req.cookies.IOTECHWEBMAIL as in the cookie.txt file. Note that the key is hard coded into both the encrypt and decrypt functions. Ugh. I hope Alabaster didn’t do that. Also, the practice of sending plaintext and ciphertext together is generally a bad idea. There are many problems with the method Alabaster is using.

Here is the beginning of the cookie\_checker code.

function cookie\_checker(req, callback){

try{

var key = 'need to put any length key in here';

var thecookie = JSON.parse(req.cookies.IOTECHWEBMAIL);

Once it has the cookie, it grabs the ciphertext and name values from the cookie.

var ciphertext = thecookie.ciphertext;

var username = thecookie.name

Then it decrypts the ciphertext using the stored key, and stores that in plaintext.

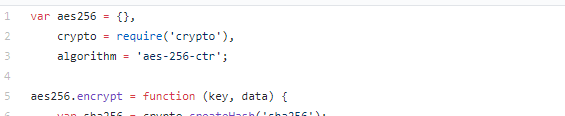
var plaintext = aes256.decrypt(key, ciphertext);

Then it compares the deciphered plaintext to the plaintext in the cookie. If they match, it returns the user name and “true” to signal success. Note that if we can pass this test, we can be any user we want to be by putting the desired user name in the cookie.

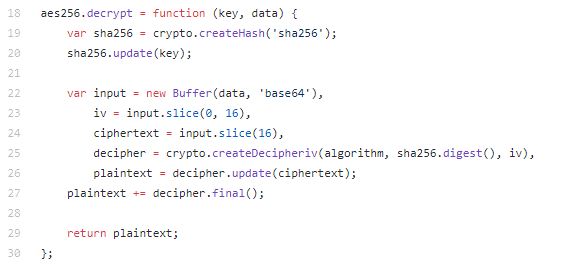
if (plaintext === thecookie.plaintext) {

return callback(true, username);

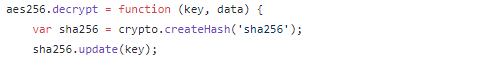
Let’s look at the aes256.decrypt function, using the code at <https://github.com/jaysvoboda/nodejs-aes256/blob/master/nodejs-aes256.js>.



Skip the encrypt function, we don’t need it.



In lines 19 and 20, the code uses the key to create the variable sha256, which is a SHA256 hash of our key. This way, the key used in the encryption (sha256.digest() seen later) is always 256 bits long.



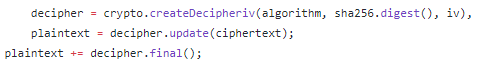
Line 22 takes the data input to the function (our ciphertext) and converts it from base64 encoding to binary.



Lines 23 and 24 are the interesting lines. Line 23 takes the first 16 bytes of input (our ciphertext) and stores them in the variable iv. Line 24 takes whatever remains and stores it in the variable ciphertext.



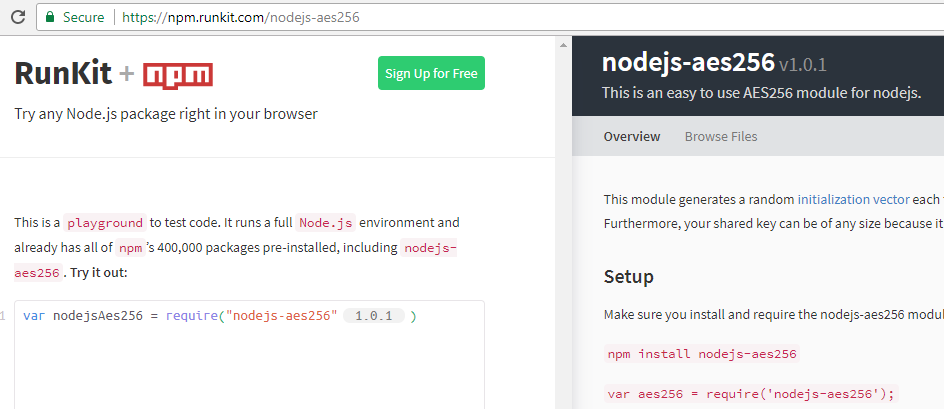
Then the function decrypts the iv and ciphertext combination and returns the result.



Again, our goal is to find ciphertext that always decrypts to some plaintext that we know, no matter what the key is. Pepper’s Hint 4 may apply.

## Testing the aes256.decrypt function

What we need to do now is get the function aes256.decrypt running in Node.js and then test what happens with different lengths of ciphertext. DigitalOcean has nice tutorials about installing Node.js on [CentOS](https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-a-centos-7-server) and [Ubuntu](https://www.digitalocean.com/community/tutorials/how-to-install-node-js-on-ubuntu-16-04). Luckily for us, someone has set up a web-based test lab for Node.js and we don’t have to install anything. Go to <https://npm.runkit.com/nodejs-aes256>.



The setup instructions say that we need to install nodejs-aes256, but that has already been done for us. The statement to require the nodejs-aes256 module is already in the left window. I like to change it so that the variable name is aes256 as we have been using, however. The “1.0.1” is inserted by RunKit.

var aes256 = require('nodejs-aes256');

You will also need to add lines for the variables key and ciphertext. Once those are in, you can just run:

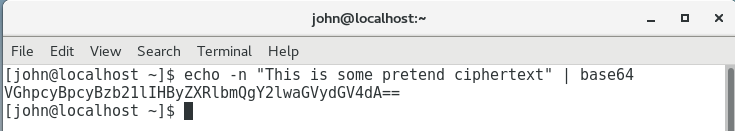
var plaintext = aes256.decrypt(key, ciphertext);

plaintext;

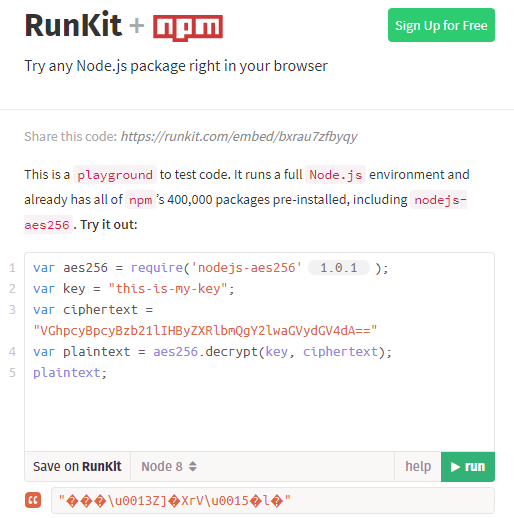
The last line will output the value of plaintext next to the orange double quote icon. Only one variable can be output at a time.

If you want to play with encryption, you can use the aes256.encrypt function as well.

Remember that the decrypt function expects the ciphertext to be base64 encoded. You can do that easily enough in Linux by echoing some text into base64, but there is a “gotcha” you need to be aware of. By default, the echo command adds a newline character to the end of the string it creates. That will cause problems if you are counting characters/bytes (or taking hashes, but that’s for another time.) To suppress the newline character, use echo -n.



The base64 encoded string can be pasted into RunKit as ciphertext. It will decrypt as gibberish, but that doesn’t matter.



## Your Mission

Find a base64 string that decrypts to the same thing, no matter what the key is. Test different strings of differing lengths in RunKit until you find one that works.

## Questions

1. What is a string of pretend ciphertext that decrypts to the same thing for any key?
2. What is the plaintext string that the ciphertext decrypts to?
3. How can we use this to break into the EWA server?