Project 4:

No Due Date but highly Recommended to Complete to learn more on Web Security on the Client Side

No Deliverables

In this project you have to retrieve the compressed folder wasec.zip, and unzip it. You will obtain a directory structure – the top level directory is ''wasec'' and with the wasec directory, you will find 6 folders as follows:

Clickjacking/

Cookies/

Cors/

Hsts/

Sub-resource-integrity/

Xss/

In the top level directory, there is a README file with basic instructions for installation. The instruction basically tells you to configure by editing your /etc/hosts file to add an entry so that without having to run a webserver like nginx or apache or lighthttpd, you can run a web service through node.js.

If you already do not have node.js on your ubuntu machine, please do that using the following steps:

- sudo apt install node.js
- sudo apt install npm

  Test the version of node.js by typing ''nodejs –version'' and/or ''node –version"" on your terminal and check that the version is above 8.x.x at least.

Once you are done with installation, please go to the clickjacking directory. Run the web application prototype in the index.js file by typing ''node index.js''. Make sure no other node instance is running on your machine at the same time serving another web application.

Open the README.md file and try the 3 experiments there – first when there is no protection against click jacking and 2nd when X-Frame-Options is correctly specified, and 3rd when CORS is used.

Note that when a server sends X-Frame-Options: DENY – then an iframe/frame cannot be rendered on the page sent by the server. Most modern browsers implement this.

Similarly, if the CORS header is sent telling that frame-ancestors header should be ‘none’ then the page cannot have an iframe embedded in it.

Check the index.js file in this directory – to figure out how this prototype web application on the server side is setting the headers to specify the CORS or X-Frame-Options.

Next you move to the cookies directory. You run the application entailed in index.js and then check the README file for the different experiments on how browsers can actually retrieve cookies from the DOM and when they cannot, when they cannot show the cookie values by looking at the document.cookie. Note that when you go from one experiment to the next, make sure to kill the previous experiment’s instance of node – otherwise it won’t work.

Try to recall various header directives such as secure, httponly etc., and try to observe by doing the experiments specified in the README.md file in this directory.

Next you move to the CORS directory. You run the application entailed in index.js and then check the README file for the different experiments. To understand the application, you need to know that in javascript “require()” call loads a javascript library module. For example, require(‘url’) would load the entire URL processing js module. var url = require(‘url’) gives a shorthand name to the url module and using url.parse for example you can access the parse function in that module.

After understanding that, you find that the query part of an URL in these experiments is checked for ‘’cors=on” string – if found the CORS header to allow all origins will be sent to the browser otherwise not. Knowing this – try to make sense of the code and the experiments you do based on the README.md file of this directory.

Next you move to the HSTS directory. For this you have to install a locally trusted certificate using ‘mkcert’.

How to install mkcert :

- sudo apt-get update
- sudo apt install wget libnss3-tools (if you do not have these installed already)
- export VER=’’v1.3.0” (I have tested with this version)
- wget -O mkcert https://github.com/FiloSottile/mkcert/releases/download/${VER}/mkcert-${VER}-linux-amd64
- chmod +x mkcert
- sudo mv mkcert /usr/local/bin

Now in the wasec directory do

- mkcert –install

Now you are ready to do the experiments by running "node index.js" inside the HSTS/ directory. Read the index.js file to understand what is happening.

HSTS is strict transport security – which means that if used – servers will tell browser to not interact over http even if an http link is inside the page.

Now use README.md to do the two experiments and using index.js file's code flow – try to understand how the strict transport security is enforced by redirection.

Next go to the directory sub-resource-integrity/ -- open README.md file and run the web application by typing "node index.js".

Read about subresource integrity check (SRI) at https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity

In short:

Subresource Integrity enables you to mitigate some risks of attacks such as this, by ensuring that the files your web application or web document fetches (from a CDN or anywhere) have been delivered without a third-party having injected any additional content into those files — and without any other changes of any kind at all having been made to those files.

You use the Subresource Integrity feature by specifying a base64-encoded cryptographic hash of a resource (file) you're telling the browser to fetch, in the value of the integrity attribute of any <script> or <link> element.

An integrity value begins with at least one string, with each string including a prefix indicating a particular hash algorithm (currently the allowed prefixes are sha256, sha384, and sha512), followed by a dash, and ending with the actual base64-encoded hash.

After reading about SRI from the above mentioned page, try the experiments in the README.md and also check the index.js file and try to understand how the application is sending the HASH values – in one case that does not match the integrity hash and in another it does.

Finally go to the directory xss and figure out how X-XSS-Protection header works and how CSP works by reading the index.js file as well as doing the experiments in the README.md.