Distributed crawler for assessing the adoption of web application security countermeasures

Team: TrojanHorse

Aashray Arora, Ajay Lakshminarayanrao

Krishna Chaitanya Bandi, Navin Agarwal

Modern browsers support a wide range of server-set policies that can be used to strengthen the security of a website. The goal of this project is to obtain information from a website to ascertain the presence and absence of server-set policies. The policies we are checking are Content-Security Policy, HTTP Strict Transport Policy, HttpOnly and Secure Cookies, Anti Clickjacking headers (X-Frame-Options), Nonces in web forms.

Security Policies

This section briefly describes the server-set security policies that we are identifying for a given domain.

HTTP Strict Transport Security (STS)

HTTP Strict Transport Security (STS) is a web security policy mechanism which helps to protect secure HTTPS websites against downgrade attacks and cookie hijacking.

STS is an opt-in security enhancement that is specified by a web application through the use of a special response header. Once a supported browser receives this header that browser will prevent any communications from being sent over HTTP to the specified domain and will instead send all communications over HTTPS. It also prevents HTTPS click through prompts on browsers.

X-Frame Options

X-Frame-Options is a technology that allows an application to specify whether or not specific pages of the site can be framed. This is meant to help prevent the clickjacking problem.

The technology is implemented as an HTTP response header that is specified per-page. Browsers supporting the X-Frame-Options header will respect the declaration of the page, and then either allow or disallow the page to be framed, depending upon the specification. A page can specify 3 different options for how it wants to be framed:
Option 1: DENY. This option means the page can never be framed by any page, including a page with the same origin.
Option 2: SAMEORIGIN. This option means the page can be framed, but only by another page with the same origin.
Option 3: Allow-From. This option means the page can be framed, but only by the specified origin.

HttpOnly

HttpOnly is an additional flag included in a Set-Cookie HTTP response header. Using the HttpOnly flag when generating a cookie helps mitigate the risk of client side script accessing the protected cookie (if the browser supports it).

If the HttpOnly flag (optional) is included in the HTTP response header, the cookie cannot be accessed through client side script (again if the browser supports this flag). As a result, even if a cross-site scripting (XSS) flaw exists, and a user accidentally accesses a link that exploits this flaw, the browser (primarily Internet Explorer) will not reveal the cookie to a third party.

If a browser does not support HttpOnly and a website attempts to set an HttpOnly cookie, the HttpOnly flag will be ignored by the browser, thus creating a traditional, script accessible cookie. As a result, the cookie (typically your session cookie) becomes vulnerable to theft of modification by malicious script.

Secure cookie

A cookie can be marked as secure cookie by a web application. Normally, the cookies are sent over the network with every HTTP or HTTPS request. The with secure cookies is that they are only sent over HTTPS only and not under HTTP.

Content Security Policy (CSP)

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting (XSS) and data injection attacks. These attacks are used for everything from data theft to site defacement or distribution of malware.

A primary goal of CSP is to mitigate and report XSS attacks. XSS attacks exploit the browser's trust of the content received from the server. Malicious scripts are executed by the victim's browser because the browser trusts the source of the content, even when it's not coming from where it seems to be coming from.

CSP makes it possible for server administrators to reduce or eliminate the vectors by which XSS can occur by specifying the domains that the browser should consider to be valid sources of executable scripts. A CSP compatible browser will then only execute scripts loaded in source files received from those whitelisted domains, ignoring all other script (including inline scripts and event-handling HTML attributes).

In addition to restricting the domains from which content can be loaded, the server can specify which protocols are allowed to be used; for example (and ideally, from a security standpoint), a server can specify that all content be loaded using HTTPS.

Nonce

A nonce is a "number used once" to help protect URLs and forms from certain types of misuse, malicious or otherwise. It can be used to protect against login CSRF (a variation of CSRF attacks), in which the attacker forges a login request to an honest site using the attacker's username and password at that site. If the forgery succeeds, the honest server responds with a Set-Cookie header that instructs the browser to mutate its state by storing a session cookie, logging the user into the honest site as the attacker. This session cookie is used to bind subsequent requests to the user's session and hence to the attacker's authentication credentials.

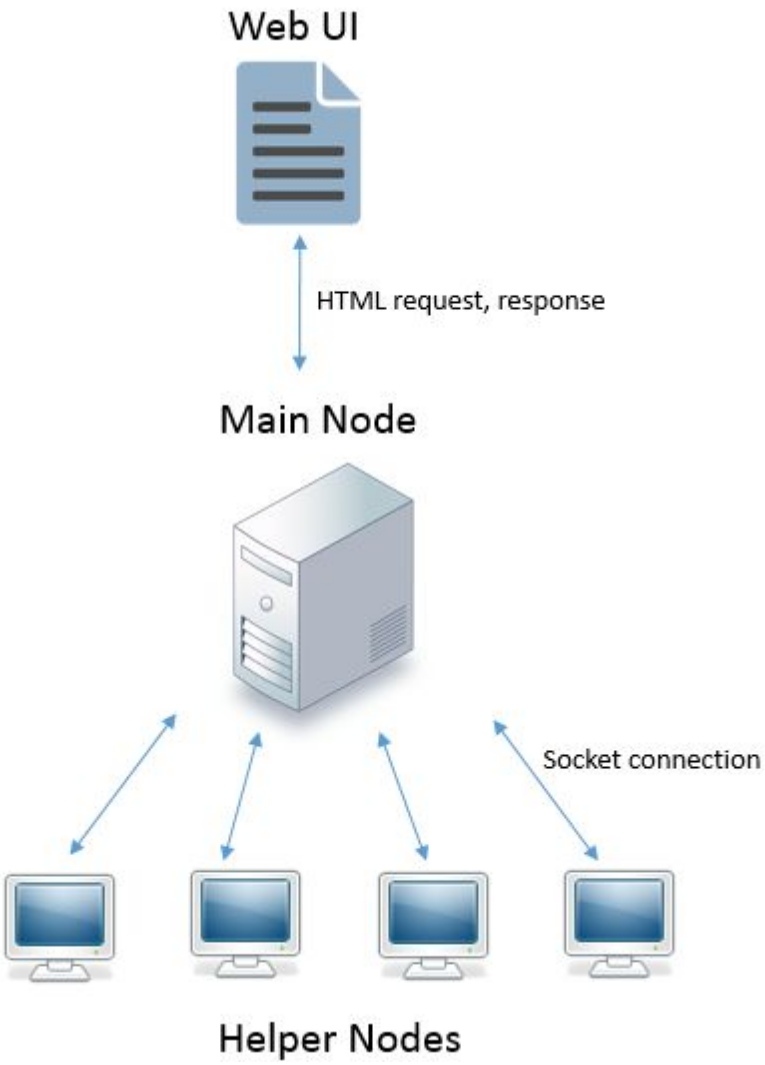
The server associates the user session with a nonce value. The browser in the client adds this nonce as a hidden input field in the login form. Whenever the user enters credentials and sends it, the nonce is sent along with the credentials. The server then verifies if the nonce received is the one which is associated with the user session. When the attacker now tries to send the credentials, it does not have the nonce associated with the user session and so the server denies authentication.

Design

The deliverable is a web based application that accepts a domain name for which the security properties are to be determined and the total number of links to be crawled in the domain. A distributed system with a main node and a set of helper nodes are established which will help crawl the domain to a required depth and to obtain server-set policies. The main node and helper nodes communicate via TCP sockets.

The web application runs Javascript that issues requests to a mod_python based server script running on the main node. The main node crawls through the given page recursively to obtain the required number of links. The main node then divides the obtained set of links and sends them to the helper nodes to obtain the server-set policies. The main node receives the policy results for the links from all the helper nodes. The main node returns this information to the webUI in a tabular format. The below diagram shows a pictorial representation of the architecture.

Architecture Diagram



Implementation

Web UI

A Web UI is created to interact with the distributed system. The Web UI is hosted on an apache web server and the backend logic is executed using mod-python. The UI consists of a webpage containing a submit form having two text fields. The first text field is used to enter the URL of the domain to crawl and the second text field is the number of URLs to be crawled and processed.

Once the form is submitted, the javascript code fetches the values from the text fields. It creates a HTTP POST request to communicate with the backend python form handler. This request to the form handler contains the domain URL and the number of links to process. The form handler calls the main node to execute with domain URL and the number of links to be processed as arguments. The main node returns a result object containing the links and their respective security policies. The results are sent to the web server to be displayed in a tabular form on the Web UI.

Main Node

The main node is responsible for receiving requests from the web UI and distributing the jobs among the helper nodes. On receiving the domain link, the main node starts crawling the link using beautifulsoup, a python library. The library helps to detect the href tags to get the html links present in the page. Links which point to file formats, like pdf are ignored. It is also ensured that the crawler crawls only links in the given domain. The crawler is stopped at a depth when at least the required number of links are obtained. We finally get a list of the required number of links along with its respective depth of crawl.

A configurable file is maintained to keep the addresses of available helper nodes. The address is a combination of their IP address and the port number available for connection. For each given address, a TCP socket is created between the main node and a helper node. This results in a list of sockets that the connection was established and is available for processing.

The main node divides the links obtained from the crawl among the active socket connections in equal parts. After connecting with a helper node, the main node sends the length of the links and next sends the links as a combined delimited string to each of the helper nodes.

The main node waits for the results from each helper on the select function from python socket library. The select is a blocking multiplexer which starts listening for the given set of sockets. It is always ready to read data when a helper node connects. The main node receives the length of the result and the result from each helper node. The result contains the security policies for each link and is stored in a map structure. The results from all the helper nodes are bundled to a single global map structure. The global map is sent to the Webserver to display. In addition, the list of links which were not processed due to unavailability of the node or due to a connection error is stored in a list. A node is also deemed as inactive if the results are not returned within a timeout window (set to 30 sec by default). The list of unprocessed links are sent to the web server and is tried to process by distributing to all other nodes.

Helper Node

Each of the helper nodes reads links from main node and determines the security policies for each link. Initially a blocking socket is started on the node which listens

continuously for a connection. Once a connection is established with the master, the length of the links string is read from the socket. The helper node continues to read the links from the socket until the length number of bytes are read.

For each of the links, the following security properties are determined by the node:

- Content-Security Policy
- HTTP Strict Transport Policy
- HttpOnly and Secure Cookies
- Anti Clickjacking headers (X-Frame-Options)
- Nonces in web forms

To obtain these properties, each link is parsed using urlparse to determine if it is valid. Based on the URL scheme a HTTP or a HTTPS connection object is created for each link. This object is used to send a GET request to obtain metadata and properties of the link.

The response header of the webpage contains information on Content Security Policy, and strict transport policy. We read these directly from the header and determine the presence of content security policy and strict transport policy.

We read the x-frame-option from the header to specify whether or not specific pages of the site can be framed. A page can specify 3 different options for how it wants to be framed, SAMEORIGIN, Allow-from or DENY.

To determine the presence of HTTPOnly and secure cookie policies we read the set-cookie property from the response header.

To detect if nonce is true for the link, we parse the page using beautifulsoup to determine if the webpage contains an input form. If present we calculate the entropy value of each hidden field in the form. If the entropy value crosses a threshold, we mark that nonce is true for the form.

The results containing the security policies of each link is communicated back to the master using the same socket connection.

Handling Node Failure

The application is designed to tolerate helper node failures. The master node divides the set of crawled links to each of the helper nodes. It keeps track of which links are sent to a helper node. If a helper node fails to send back complete results, the master node uses that information to know the unprocessed links and stores them in a list. This list is sent to the web application to request the application to process these links on other active nodes.

The web server immediately displays the results for all processed links. It notifies the user to wait for remaining results. The unprocessed links are stored in a hidden field in the webpage. The web server creates a HTTP post request with a delimited string of unprocessed links.

The main handler receives the unprocessed links and distributes to all active connections present. We determine the security policies of each of the links by distributing to the helper nodes. They communicate the results back to the main node which bundles them to a single result. The main node communicates the result to the web server. The web server appends to the results table on the UI and signals the user by removing the wait notification.

Contributions:

In design and architecture decisions we all collaborated during group meetings.

Aashray Arora - Main node handler and Helper node handler (Link extraction)

Ajay Lakshminarayanarao - Main node Handler and Helper node handler (Socket communication)

Krishna Chaitanya Bandi - Front end, Interaction between web, Mod-python, Failure handling

Navin Agarwal - Security properties

References:

- https://www.owasp.org/index.php/HTTP_Strict_Transport_Security
- https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security
- https://blog.whitehatsec.com/x-frame-options/
- https://www.owasp.org/index.php/HttpOnly
- http://cookiecontroller.com/internet-cookies/secure-cookies/
- https://developer.mozilla.org/en-US/docs/Web/Security/CSP/Introducing_Content_Security_Policy
- http://seclab.stanford.edu/websec/csrf/csrf.pdf
- https://codex.wordpress.org/WordPress_Nonces
- http://www.rackspace.com/knowledge_center/article/ubuntu-modpython-installation