November 08, 2018

AUTOMATION TO PREVENT QUERY PARAMETER EXPOSURE ON HYPERTEXT TRANSFER PROTOCOL GET REQUESTS

Amol Borole

AUTOMATION TO PREVENT QUERY PARAMETER EXPOSURE ON HYPERTEXT TRANSFER PROTOCOL GET REQUESTS

AUTHORS:
Amol Borole

ABSTRACT

Techniques are provided to allow an existing web application to seamlessly use custom HyperText Transfer Protocol (HTTP) headers under the hood. This resolves an entire category of web security problems without a major redesign at the feature level of code.

DETAILED DESCRIPTION

Accidental exposure of query parameters on HyperText Transfer Protocol (HTTP) GET requests (in Secure Sockets Layer (SSL) based applications) is a common problem in web applications and there is a risk that exposed information may get compromised unintentionally.

The known alternatives are to not use sensitive data as query parameters or to redesign application to send query parameters as POST data. However, these may not be practical in some scenarios. There is no simple solution that allows limiting the risk while still allowing feature developers to continue using query parameters on GET requests.

This issue may be encountered in various web applications and a common solution would be useful.

In accordance with techniques presented herein, application developers may design the application as usual, and GET requests may have query parameters. But before an Asynchronous JavaScript And eXtensible Markup Language (AJAX) request is sent, it is intercepted and query parameters are converted to a custom header which the server side application code can decode and provide actual query parameters to the backend code that needs to process the HTTP request.

A client-side interceptor may be a web application library that sends the AJAX requests via the browser which can inspect and detect if query parameters are being added to a requested Uniform Resource Locator (URL). The name value pairs are collected, parsed, and sent through custom HTTP header.

A server-side decoder may be a filter component on the server side that processes all incoming requests, checks for the specific custom header, extracts the name value pairs, and passes them to the backend code that is supposed to process the request.

Web application frontend logic may proceed as follows:

1. Override standard XMLHttpRequest's open method so that special processing of the request can be done before the browser processes that request.
2. Inspect request inside open method to check if query parameters are present
3. If query parameters are present, extract the portion of URL that contains them, encode them, add them to a special HTTP header (e.g., "QPH"), set this header on the request, and update the requested URL to strip off the query parameters and invoke the original open method.
4. If query parameters are not found, no special processing is needed, and the original open method may be directly invoked.

Web application backend logic may proceed as follows:

1. To the existing chain of servlet filters, add two new filters: a pre-processor as the first filter in the chain, and a post-processor as the last one in the chain, while maintaining the order/sequence of the remaining ones.
2. If the custom header is not detected in pre-processor, then no special action is needed, and the control will simply be forwarded to the next filter in the chain.
3. If the custom header is detected in pre-processor, query parameters are extracted, a new forwarding Uniform Resource Identifier (URI) is generated with the query parameters added back, and HTTP request wrapper is created that is aware of all query parameters, the forwarding URI is set as an attribute on the wrapper, and the next filter is executed using the wrapper instead of the original request.
4. Using the wrapper allows existing filters to access query parameters seamlessly.
5. When control reaches the post-processor, it is determined whether the forwarding URI attribute is set. If it is not found, no special action is needed, and control is passed to the next filter in the chain. If the forwarding URI attribute is found, the request dispatcher is obtained for forwarding the URI and the original request is forward using it.

An example flow is provided as follows:

1. A web app is named "webapp1" and an existing feature is passing some data as query parameters resulting in a request URL that looks like "/webapp1/feature/action?a=1&b=2&c=3".
2. Using the techniques described herein, the web application frontend code sends a request with the URL "/webapp1/feature/action".
3. The pre-processor filter may extract the QPH header value and create a forwarding URI "/feature/action?a=1&b=2&c=3".
4. The post-processor filter may detect the forwarding URI attribute and create a dispatcher for "/feature/action?a=1&b=2&c=3" and forward the original request to that.
5. Backend code in webapp1 may seamlessly access query parameter values using the standard http.getParameter Application Programming Interface (API).

Figure 1 below illustrates an example flow on the client side.

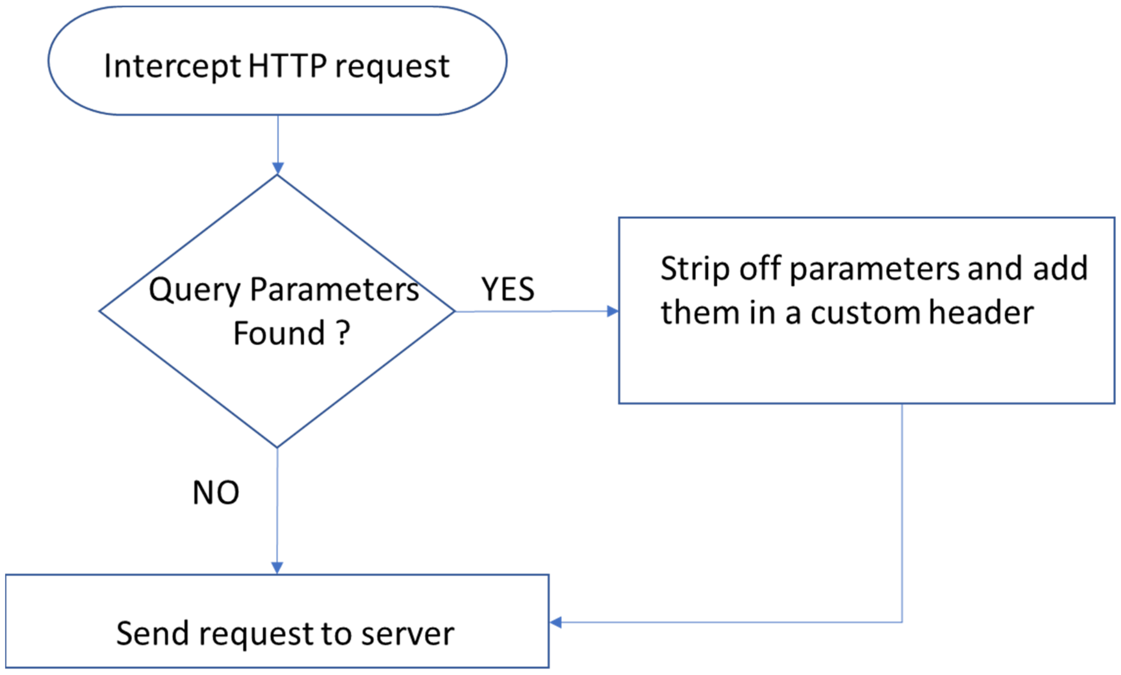

Figure 1

Figure 2 below illustrates an example flow on the server side.

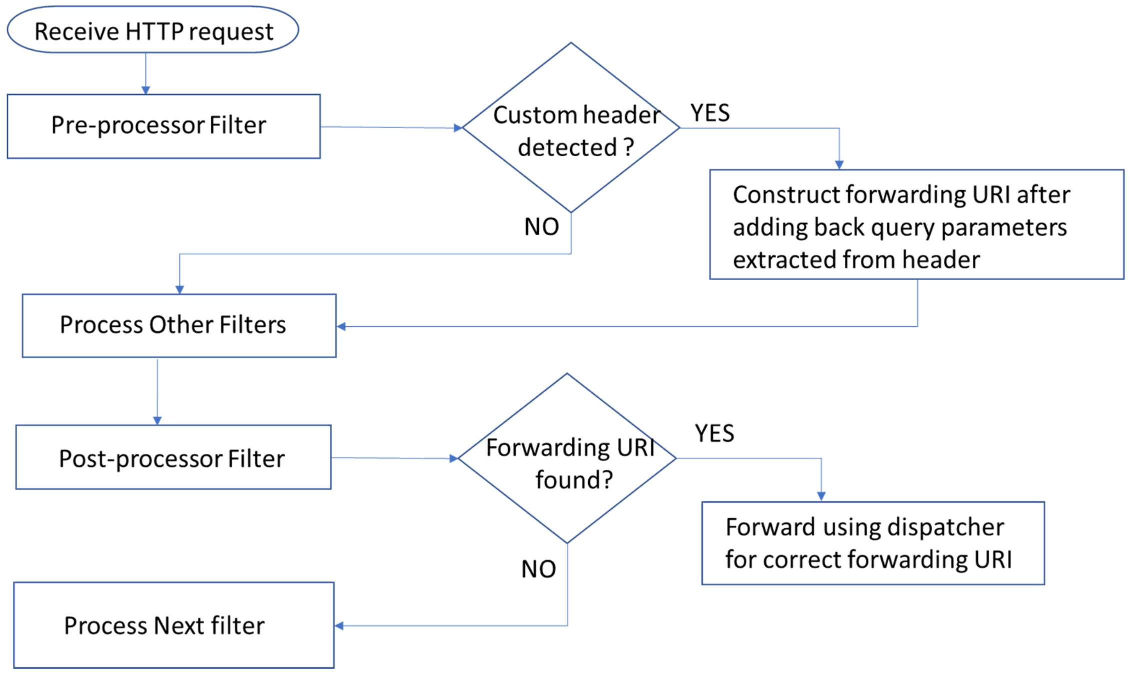

Figure 2

In summary, techniques are provided to allow an existing web application to seamlessly use custom HTTP headers under the hood. This resolves an entire category of web security problems without a major redesign at the feature level of code.