Intro to HTTP Verb Tampering

The HTTP protocol works by accepting various HTTP methods as verbs at the beginning of an HTTP request. Depending on the web server configuration, web applications may be scripted to accept certain HTTP methods for their various functionalities and perform a particular action based on the type of the request.

While programmers mainly consider the two most commonly used HTTP methods, GET and POST, any client can send any other methods in their HTTP requests and then see how the web server handles these methods. Suppose both the web application and the back-end web server are configured only to accept GET and POST requests. In that case, sending a different request will cause a web server error page to be displayed, which is not a severe vulnerability in itself (other than providing a bad user experience and potentially leading to information disclosure). On the other hand, if the web server configurations are not restricted to only accept the HTTP methods required by the web server (e.g. GET/POST), and the web application is not developed to handle other types of HTTP requests (e.g. HEAD, PUT), then we may be able to exploit this insecure configuration to gain access to functionalities we do not have access to, or even bypass certain security controls.

HTTP Verb Tampering

To understand `HTTP Verb Tampering`, we must first learn about the different methods accepted by the HTTP protocol. HTTP has 9 different verbs that can be accepted as HTTP methods by web servers. Other than GET and POST, the following are some of the commonly used HTTP verbs:

Verb	Description
`HEAD`	Identical to a GET request, but its response only contains the `headers`, without the response body
`PUT`	Writes the request payload to the specified location
`DELETE`	Deletes the resource at the specified location
`OPTIONS`	Shows different options accepted by a web server, like accepted HTTP verbs
`PATCH`	Apply partial modifications to the resource at the specified location

As you can imagine, some of the above methods can perform very sensitive functionalities, like writing (PUT) or deleting (DELETE) files to the webroot directory on the back-end server. As discussed in the Web Requests module, if a web server is not securely configured to manage these methods, we can use them to gain control over the back-end server. However, what makes HTTP Verb Tampering attacks more common (and hence more critical), is that they are caused by a misconfiguration in either the back-end web server or the web application, either of which can cause the vulnerability.

Insecure Configurations

Insecure web server configurations cause the first type of HTTP Verb Tampering vulnerabilities. A web server's authentication configuration may be limited to specific HTTP methods, which would leave some HTTP methods accessible without authentication. For example, a system admin may use the following configuration to require authentication on a particular web page:

Code: xml

```xml
<Limit GET POST>
    Require valid-user
</Limit>
```

As we can see, even though the configuration specifies both GET and POST requests for the authentication method, an attacker may still use a different HTTP method (like HEAD) to bypass this authentication mechanism altogether, as will see in the next section. This eventually leads to an authentication bypass and allows attackers to access web pages and domains they should not have access to.

Insecure Coding

Insecure coding practices cause the other type of HTTP Verb Tampering vulnerabilities (though some may not consider this Verb Tampering). This can occur when a web developer applies specific filters to mitigate particular vulnerabilities while not covering all HTTP methods with that filter. For example, if a web page was found to be vulnerable to a SQL Injection vulnerability, and the back-end developer mitigated the SQL Injection vulnerability by the following applying input sanitization filters:

Code: php

```php
$pattern = "/^[A-Za-z\s]+$/";

if(preg_match($pattern, $_GET["code"])) {
    $query = "Select * from ports where port_code like '%" . $_REQUEST["code"] . "%'";
    ...SNIP...
}
```

We can see that the sanitization filter is only being tested on the GET parameter. If the GET requests do not contain any bad characters, then the query would be executed. However, when the query is executed, the `$_REQUEST["code"]` parameters are being used, which may also contain POST parameters, `leading to an inconsistency in the use of HTTP Verbs`. In this case, an attacker may use a POST request to perform SQL injection, in which case the GET parameters would be empty (will not include any bad characters). The request would pass the security filter, which would make the function still vulnerable to SQL Injection.

While both of the above vulnerabilities are found in public, the second one is much more common, as it is due to mistakes made in coding, while the first is usually avoided by secure web server configurations, as documentation often cautions against it. In the coming sections, we will see examples of both types and how to exploit them.

← Previous   Next →   Mark Complete & Next

Cheat Sheet

My Workstation

OFFLINE

Start Instance

∞ / 1 spawns left