Verb Tampering Prevention

After seeing a few ways to exploit Verb Tampering vulnerabilities, let's see how we can protect ourselves against these types of attacks by preventing Verb Tampering. Insecure configurations and insecure coding are what usually introduce Verb Tampering vulnerabilities. In this section, we will look at samples of vulnerable code and configurations and discuss how we can patch them.

Insecure Configuration

HTTP Verb Tampering vulnerabilities can occur in most modern web servers, including `Apache`, `Tomcat`, and `ASP.NET`. The vulnerability usually happens when we limit a page's authorization to a particular set of HTTP verbs/methods, which leaves the other remaining methods unprotected.

The following is an example of a vulnerable configuration for an Apache web server, which is located in the site configuration file (e.g. `000-default.conf`), or in a `.htaccess` web page configuration file:

Code: xml

```xml
<Directory "/var/www/html/admin">
    AuthType Basic
    AuthName "Admin Panel"
    AuthUserFile /etc/apache2/.htpasswd
    <Limit GET>
        Require valid-user
    </Limit>
</Directory>
```

As we can see, this configuration is setting the authorization configurations for the `admin` web directory. However, as the `<Limit GET>` keyword is being used, the `Require valid-user` setting will only apply to `GET` requests, leaving the page accessible through `POST` requests. Even if both `GET` and `POST` were specified, this would leave the page accessible through other methods, like `HEAD` or `OPTIONS`.

The following example shows the same vulnerability for a `Tomcat` web server configuration, which can be found in the `web.xml` file for a certain Java web application:

Code: xml

```xml
<security-constraint>
    <web-resource-collection>
        <url-pattern>/admin/*</url-pattern>
        <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>admin</role-name>
    </auth-constraint>
</security-constraint>
```

We can see that the authorization is being limited only to the `GET` method with `http-method`, which leaves the page accessible through other HTTP methods.

Finally, the following is an example for an `ASP.NET` configuration found in the `web.config` file of a web application:

Code: xml

```xml
<system.web>
    <authorization>
        <allow verbs="GET" roles="admin">
            <deny verbs="GET" users="*">
        </deny>
        </allow>
    </authorization>
</system.web>
```

Once again, the `allow` and `deny` scope is limited to the `GET` method, which leaves the web application accessible

through other HTTP methods.

The above examples show that it is not secure to limit the authorization configuration to a specific HTTP verb. This is why we should always avoid restricting authorization to a particular HTTP method and always allow/deny all HTTP verbs and methods.

If we want to specify a single method, we can use safe keywords, like `LimitExcept` in Apache, `http-method-omission` in Tomcat, and `add/remove` in ASP.NET, which cover all verbs except the specified ones.

Finally, to avoid similar attacks, we should generally `consider disabling/denying all HEAD requests` unless specifically required by the web application.

Insecure Coding

While identifying and patching insecure web server configurations is relatively easy, doing the same for insecure code is much more challenging. This is because to identify this vulnerability in the code, we need to find inconsistencies in the use of HTTP parameters across functions, as in some instances, this may lead to unprotected functionalities and filters.

Let's consider the following `PHP` code from our `File Manager` exercise:

Code: php

```php
if (isset($_REQUEST['filename'])) {
    if (!preg_match('/[^A-Za-z0-9. _-]/', $_POST['filename'])) {
        system("touch " . $_REQUEST['filename']);
    } else {
        echo "Malicious Request Denied!";
    }
}
```

If we were only considering Command Injection vulnerabilities, we would say that this is securely coded. The `preg_match` function properly looks for unwanted special characters and does not allow the input to go into the command if any special characters are found. However, the fatal error made in this case is not due to Command Injections but due to the `inconsistent use of HTTP methods`.

We see that the `preg_match` filter only checks for special characters in `POST` parameters with `$_POST['filename']`. However, the final `system` command uses the `$_REQUEST['filename']` variable, which covers both `GET` and `POST` parameters. So, in the previous section, when we were sending our malicious input through a `GET` request, it did not get stopped by the `preg_match` function, as the `POST` parameters were empty and hence did not contain any special characters. Once we reach the `system` function, however, it used any parameters found in the request, and our `GET` parameters were used in the command, eventually leading to Command Injection.

This basic example shows us how minor inconsistencies in the use of HTTP methods can lead to critical vulnerabilities. In a production web application, these types of vulnerabilities will not be as obvious. They would probably be spread across the web application and will not be on two consecutive lines like we have here. Instead, the web application will likely have a special function for checking for injections and a different function for creating files. This separation of code makes it difficult to catch these sorts of inconsistencies, and hence they may survive to production.

To avoid HTTP Verb Tampering vulnerabilities in our code, `we must be consistent with our use of HTTP methods` and ensure that the same method is always used for any specific functionality across the web application. It is always advised to `expand the scope of testing in security filters` by testing all request parameters. This can be done with the following functions and variables:

Language	Function
PHP	`$_REQUEST['param']`
Java	`request.getParameter('param')`
C#	`Request['param']`

If our scope in security-related functions covers all methods, we should avoid such vulnerabilities or filter bypasses.