HTTP BASICS

According to **RFC 7230**, the **Hypertext Transfer Protocol (HTTP)** is a stateless **application-level** protocol for distributed, collaborative, hypertext information systems. It was defined almost 30 years ago and became the foundation of the Internet.  
  
**HTTP** resides on the **application layer** of the **ISO OSI** model (described in **ISO/IEC 7498-1** standard). There are seven layers in the **ISO OSI** model and each one has its own role in the transmission of data from the client to the server.  
  
As an application layer protocol, **HTTP** provides the definition of how to transfer the data formatted as hypertext between different locations on the Internet. The data formatted in a way the **HTTP** protocol dictates is transformed all the way down to the Physical level on a sender side and then does it back to the top on a receiver side.  
  
When data is sent, the protocol on every layer appends additional data to it so that the original data can be transferred and reconstructed on the receiver side.

HTTP is a client-server protocol meaning that the client side sends a request and receives a response from the server side. In addition, there are intermediate devices that can read or manipulate requests and responses (e.g. on the Application Layer there can be filtering devices like antiviruses or Web Application Firewalls, load balancers, caches and so on).

A user sends requests to the server through the user-agent: a browser or any kind of program designed for communication with a particular server.

In the request, user-agent defines what web page it wants to receive from the server.

Web server either serves a static page or dynamically generates it and serves to the user.

A web page is a document written in HyperText Markup Language (HTML) that is usually accompanied by styles and scripts that make it look better. Scripts on the web page may retrieve additional data that is used to update the web page.

Alternatively, a server can respond with formatted data (JSON, XML, etc) that is further displayed in the user-agent application.

**SAMPLE GET REQUEST**

Let us consider an example of the GET request to some server.

Our hero Alice wants to open the https://codebashing.com/courses page. Her browser sends the following GET request to the https://codebashing.com server.

GET /courses HTTP/1.1

Host: codebashing.com

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/68.0.3440.84 Safari/537.36

Referer: https://codebashing.com/

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8

Accept-Encoding: gzip, deflate

Accept-Language: en-US,en;q=0.9,ru;q=0.8

Cache-Control: max-age=0

Upgrade-Insecure-Requests: 1

Connection: close

**HTTP security headers**

There is a number of HTTP response headers that you should use to increase the security of your web application. They are referred to as HTTP security headers.

Once implemented, HTTP security headers restrict modern browsers from running into easily preventable vulnerabilities. They also provide yet another, additional layer of security by helping to mitigate security vulnerabilities and prevent attacks (like XSS, Clickjacking, information leakage, etc.). But it is important to mention that HTTP security headers are not intended to replace proper, secure code.

In this lesson, we will explore the most important HTTP security headers to help you better understand their purpose and how to properly implement them.

In the right pane, you can see a sample HTTP response where all the security-related headers are highlighted

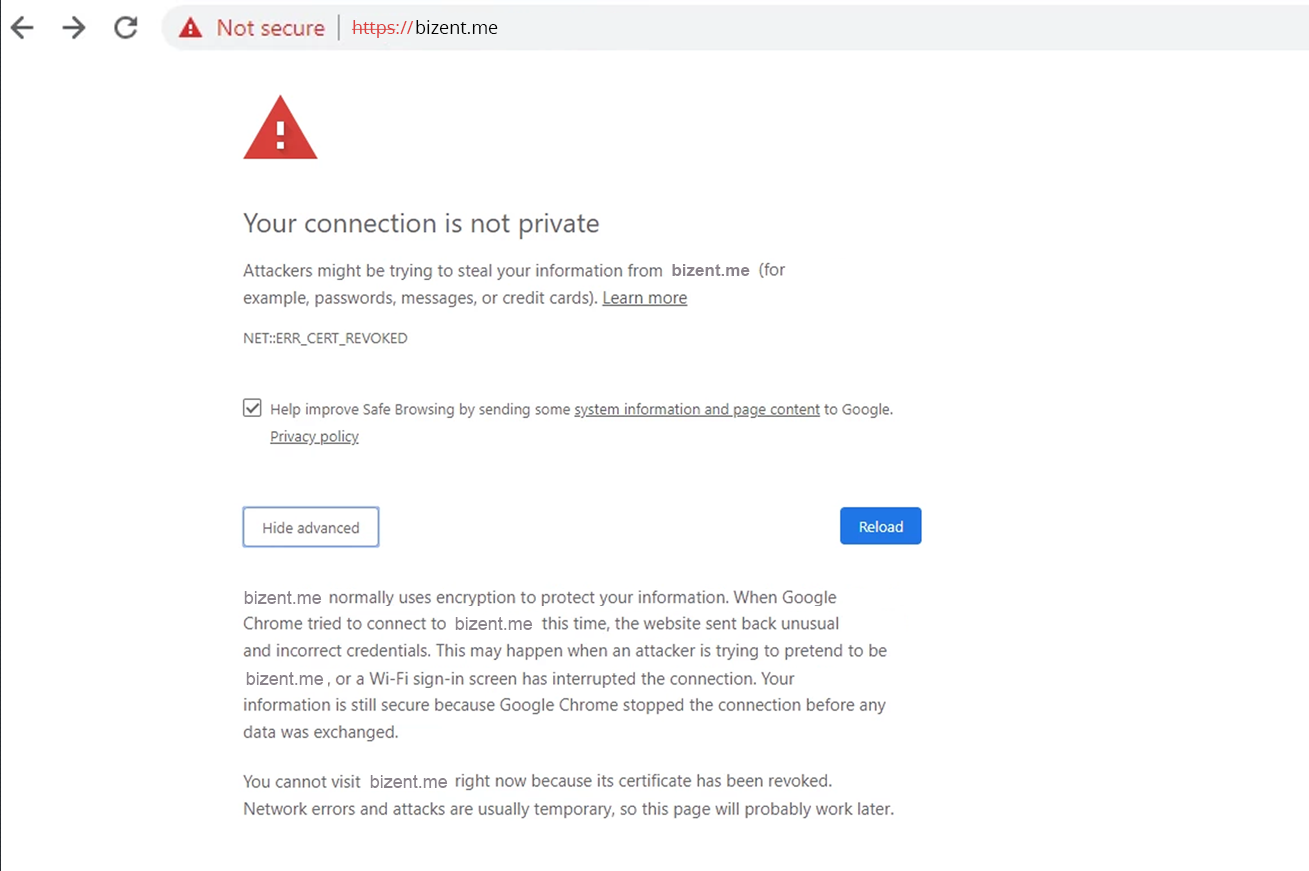
**HTTP STRICT TRANSPORT PROTOCOL**

HTTP STRICT TRANSPORT PROTOCAL (HSTS) - is a mechanism that prevents user-agents (a browser or any kind of program designed for communication with a particular server) from browsing a website via an unencrypted connection in case an encrypted connection can be established, and only using a trusted certificate.

If the request is communicated through an unencrypted channel, it can be captured and tampered with by an attacker. The attacker then can steal or modify any information transmitted between the client and the server or redirect the user to a phishing website. So, the first goal of HSTS is to ensure traffic is encrypted, so it instructs the browser to always use HTTPS instead of HTTP.

Usually, browsers allow users to ignore TLS errors and continue browsing potentially insecure websites. With HSTS enabled, the user will be unable to skip the browser warning and continue. The second important goal of HSTS is to make sure that the traffic is encrypted using a trusted and valid certificate.

In the right pane, you can see a screenshot with the error that the user gets when the certificate of the target website is no longer valid (note that the user is unable to proceed to the target website).



X-XSS PROTECTION

Some modern browsers have built-in XSS protection mechanisms that can be used as an additional layer of security against Reflected XSS. The main problem with that is that all of the browsers implement built-in XSS filtering differently, so to add more control to the process and make sure that the loading of a page with the malicious content will be blocked, the X-XSS-Protection header is needed.

X-XSS-Protection header is an optional HTTP header that performs XSS filtering by defining the anti-XSS mechanism behavior: from sanitizing the page by blocking injected Javascript to preventing page rendering and reporting the violation.

By default, browsers that support XSS filtering have it enabled. Though it can be disabled, this is considered a bad practice; often, if an application requires XSS protection to be disabled in order to function properly, it is an indication that the application is quite likely vulnerable to XSS.

Please note that only using the X-XSS-Protection header will not protect your application from XSS, but this header will make an important input in your defense-in-depth strategy and make it more robust.

In the right pane, you can see a screenshot of the Google Chrome webpage that was blocked by the built-in XSS filter.

XFRAME OPTIONS

Some modern browsers have built-in XSS protection mechanisms that can be used as an additional layer of security against Reflected XSS. The main problem with that is that all of the browsers implement built-in XSS filtering differently, so to add more control to the process and make sure that the loading of a page with the malicious content will be blocked, the X-XSS-Protection header is needed.

X-XSS-Protection header is an optional HTTP header that performs XSS filtering by defining the anti-XSS mechanism behavior: from sanitizing the page by blocking injected Javascript to preventing page rendering and reporting the violation.

By default, browsers that support XSS filtering have it enabled. Though it can be disabled, this is considered a bad practice; often, if an application requires XSS protection to be disabled in order to function properly, it is an indication that the application is quite likely vulnerable to XSS.

Please note that only using the X-XSS-Protection header will not protect your application from XSS, but this header will make an important input in your defense-in-depth strategy and make it more robust.

In the right pane, you can see a screenshot of the Google Chrome webpage that was blocked by the built-in XSS filter.

X FRAME OPTIONS is also covered by Content-Security-Policy (CSP). CSP is a suite of headers with security directives for multiple uses (which will be covered in their own lesson). Among these security directives is the frame-ancestorsdirective which pursues the same goal as the X-Frame-Options header. The main difference between them is the implementation of the SAMEORIGIN directive of the X-Frame-Options header.

Various user-agents interpret SAMEORIGIN directive in a different way, and some of them only check the top-level domain. Whereas with the CSP's frame-ancestors directive, the whole chain of origins is checked. (The example of the chain of origins is an iframe inside an iframe, inside an iframe, and so on.)

The CSP W3C recommendation states that with the CSP frame-ancestors directive being introduced, the X-Frame-Options header becomes obsolete. Thus CSP should be used to prevent Clickjacking in the first place and the X-Frame-Options header should be used for backward compatibility with browsers that don't support this CSP directive.

In the right pane, you can see a screenshot of the webpage where the content was denied to be displayed due to Content Security Policy restrictions.

## HTTP VERSION

The HTTP protocol version is the universal component that can be found both in requests and responses.

There are currently two most used versions of HTTP: v1.1 and v2.

When HTTP v1.1 is used in HTTP requests, it presents them in a human-readable format. HTTP v2 requests are binary and require special tools to be read. The main difference between v1.1. and v2 is the way requests and responses are transmitted: HTTP2 is aimed at decreasing the time required for data transfer.

URL

HTTP COOKIES

An HTTP cookie is a small piece of data that a server sends to a client. The server sends the cookie using the Set-Cookie header in the response. It looks like that:

Set-Cookie: trackingCookie=user1357272

User-agent saves a cookie from the response and sends it back in the Cookie request header like that:

Cookie: trackingCookie=user1357272

Cookies are used for the following purposes:

1. Session management

HTTP is a stateless protocol meaning that two requests cannot be correlated to the same source or to each other even if they are sent from the same IP address and contain the same user-agent. In other words, without some additional information, the server is not able to understand that two requests were sent consequently by the same user.

To keep the flow of all the user's requests, session identifiers are used. Usually, a session identifier resides in a cookie and is sent with every request in the Cookie header. Thus the server can track all actions the user makes, display him the right interface, and send responses with correct data

2. Personalization

Sometimes cookies are used to store user preferences and non-sensitive application data.

3. Tracking

Nowadays many companies try to understand their users' behavior and track all the actions users make. Cookies can be used for this purpose as they can persist in the browser, until cleared, and can be tied to the user behavior, even if the user is not authenticated.

And now let us look at the response. Near the HTTP version, we can see the response code (200 OK in our example).

There are five groups of response status codes with the corresponding status message. Let us review these groups and look at the most popular status codes.

## Successful responses

200 – OK

This status shows that the request has succeeded. Either the desired information exists on the server and is transmitted in the response body, or the desired application state change has been performed successfully.

Redirection responses

301 – Moved permanently

The status means that the desired resource resides at another location (which is usually specified in the Location header of the response).

302 – Found

The status also means that the desired page has been moved and a new location is specified in the Location header of the response. But in this case, the page has been moved temporarily.

304 – Not Modified

This code is used to instruct the browser to used cached data because it hasn’t changed since the previous request.

Client error responses

400 – Bad Request

This status means that the request syntax is invalid and the server was not able to process it.

401 – Unauthorized

This status means that the requested resource requires a client to be authenticated.

403 – Forbidden

This status means that the authenticated user does not have permissions to access the requested resource.

404 – Not Found

This code shows that the requested resource does not exist on the server.

Server error responses

500 – Internal Server Error

This code means that the request syntax is correct, but the server could not prepare a response for the request due to an exception thrown by the application code.

# [**Is SQL injection possible via integer?**](https://security.stackexchange.com/questions/142373/is-sql-injection-possible-via-integer)

Below is the following pseudo-code:

public updated(int id) {

// Note that variable \*\*id\*\* is not surrounded by single/double quotes.

sql = "Update table user set status=2 where user\_id=\*\*id\*\* ";

// execute command

}

Is this vulnerable to SQL injection ( Arithmetic SQL injection perhaps) ?

No, SQL injection is not possible in this case – paj28 Nov 11 '16 at 12:36

As long as this is a language which enforces types, it should be safe. If the "int" is considered a type hint which can be ignored, might not be. However, this would be unusual... – Matthew Nov 11 '16 at 13:12

No, your query expects an integer and is guaranteed to get one. An SQL injection vulnerability only emerges when an attacker can supply unexpected data that would alter the query. Different integers don't influence the syntax. But as soon as you stop enforcing the type it becomes unsafe.

As @CodesInChaos remarked, if you are the author of this snippet you should consider using parametrized queries instead. That way the supplied type becomes irrelevant which is easier to audit and harder to break by accident.

## SESSION FIXATION

combination of the following best practices could help to defend against Session Fixation attacks:

1. Ensure that only server-generated session values are accepted by the application.

2. Upon a successful login, invalidate the original session token, and re-issue a new session token.

3. Prevent the application from accepting session tokens via GET or POST requests and instead store session values within HTTP cookies only.

Let us see how a potential fix can be applied to our vulnerable example to remediate the session fixation vulnerability.

## USE OF INSUFFICIENT RANDOM NUMBERS

The best way to remediate the Insufficiently Random Values vulnerability is to use an algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations with the seeds of the adequate length.

In general, if a pseudo-random number generator is not advertised as being cryptographically secure, it should not be used in security-sensitive contexts.

Pseudo-random number generators can produce predictable numbers if the generator is known and the seed can be guessed. A 256-bit seed is a good starting point for producing a "random enough" number.

Let's see how the above recommendations can be applied to our vulnerable example to improve our session token generator.

## STORED XSS PREVENTION

To defend against Stored Cross-Site Scripting attacks, it is important to ensure that user-supplied data output is encoded before being served by the application to other users.

Output encoding effectively works by escaping user-supplied data immediately before it is served to users of the application.

When the data is correctly escaped before being served to the user for display in their browser, the browser does not interpret it as code and instead interprets it as data, thus ensuring it does not get executed.

For example, the string <script> is converted to &lt;script&gt; when properly escaped and is simply rendered as text in the user's browser window rather than being interpreted as code.

Let's see how these techniques can be applied to our vulnerable example to remediate the Stored XSS vulnerability.

To defend against **Stored Cross-Site Scripting** attacks, it is important to ensure that user-supplied data output is encoded before being served by the application to other users.  
  
**Output encoding** effectively works by escaping user-supplied data immediately before it is served to users of the application.  
  
When the **data is correctly escaped** before being served to the user for display in their browser, the browser does not interpret it as code and instead interprets it as data, thus ensuring it does not get executed.  
  
For example, the string **<script>** is converted to **&lt;script&gt;** when properly escaped and is simply rendered as text in the user's browser window rather than being interpreted as code.  
  
Let's see how these techniques can be applied to our vulnerable example to remediate the **Stored XSS** vulnerability.