A Security-focused HTTP Primer

What follows is a primer on the key security-oriented characteristics of the HTTP protocol. It's a collection of a number of different sub-topics, explained in my own way, for the purpose of having a single reference point when needed.

BASICS

- Message-based
  You make a request, you get a response.
- Line-based
  Lines are quite significant in HTTP. Each header is on an individual line (each line ends with a <crlf>), and a blank line separates the header section from the optional body section.
- Stateless
  HTTP doesn't have the concept of state built-in, which is why things like cookies are used to track users within and across sessions.

QUERY STRINGS, PARAMETERS, AND AMPERSANDS

- Query Strings (?)
  A query string is defined by using the question mark (?) character after the URL being requested, and it defines what is being sent to the web application for processing. They are typically used to pass the contents of HTML forms, and are encoded using name:value pairs.

```
http://google.com/search?query=mysearch
```

- Parameters (something=something)
  In the request above the parameter is the "query" value–presumably indicating it's what's being searched for. It is followed by an equals sign (=) and then the value of the parameter.

```
http://google.com/search?q=mysearch
```

- The Ampersand (&)
  Ampersands are used to separate a list of parameters being sent to the same form, e.g. sending a query value, a language, and a verbose value to a search form.

```
http://google.com/search?q=mysearch&lang=en&verbose=1
```

[Ampersands are not mentioned in the HTTP spec itself; they are used as a matter of convention.]

URL ENCODING

URL encoding seems more tricky than it is. It's basically a workaround for a single rule in RFC 1738, which states that:

> …Only alphanumerics [0-9a-zA-Z], the special characters "$-_.+!*'(),"
> [not including the quotes – ed], and reserved characters used for their
> reserved purposes may be used unencoded within a URL.

The issue is that humans are inclined to use far more than just those characters, so we need some way of getting the larger range of characters transformed into the smaller, approved set. That's what URL Encoding does. As mentioned here in a most excellent piece on the topic, there are a few basic groups of characters that need to be encoded:

1. **ASCII Control Characters:** because they're not printable.
2. **Non-ASCII Characters:** because they're not in the approved set (see the requirement above from RFC 1738). This includes the upper portion of the ISO-Latin character set (see my encoding primer to learn more about character sets)
3. **Reserved Characters:** these are kind of like system variables in programming–they mean something within URLs, so they can't be used outside of that meaning.
   - Dollar ("$")
   - Ampersand ("&")
   - Plus ("+")
   - Comma (",")
   - Forward slash/Virgule ("/")
   - Colon (":")

- Semi-colon (“;”)
- Equals (“=”)
- Question mark (“?”)
- ‘At’ symbol (“@”)

4. **Unsafe Characters:** characters that could be misunderstood and cause problems.
   - Space ( )
   - Quotes (“”)
   - Less Than and Greater Than Symbols (<>)
   - Pound (#)
   - Percent (%)
   - Curly Braces ({})
   - The Pipe Symbol (|)
   - Backslash ()
   - Caret (^)
   - Tilde (~)
   - Square Brackets ([ ])
   - Backtick (`)

For any of these characters listed that can’t (or shouldn’t be) be put in a URL natively, the following encoding algorithm must be used to make it properly URL-encoded:

1. Find the ISO 8859-1 code point for the character in question
2. Convert that code point to two characters of hex
3. Append a percent sign (%) to the front of the two hex characters

This is why you see so many instances of %20 in your URLs. That’s the URL-encoding for a space.

AUTHENTICATION

Here are the primary HTTP authentication types:

BASIC

- A user requests page protected by basic auth
- Server sends back a 401 and a `WWW-Authenticate` header with the value of `basic`
- The client takes his username and password–separated by a colon–and Base64 encodes it
- The client then sends that value in an `Authorization` header, like so: `Authorization: Basic BTxhZGRpbjpbcGAuINMlc2FtZC==`

[As the authors of The Web Application Hacker's Handbookpoint out, Basic Authentication isn't as bad as people make it out to be. Or, to be more precise, it's no worse than Forms-based Authentication (the most common type). The reason for this is simple: Both send credentials in plain-text by default (actually, at least Basic offers Base64, whereas Forms-based isn't even encoded). Either way, the only way for either protocol to even approach security is by adding SSL/TLS.]

DIGEST

- A user requests page protected by digest auth
- The server sends back a 401 and a `WWW-Authenticate` header with the value of `digest` along with a nonce value and a realm value
- The user concatenates his credentials with the nonce and realm and uses that as input to MD5 to produce one has (HA1)
- The user concatenates the method and the URI to create a second MD5 hash (HA2)
- The user then sends an `Authorize` header with the realm, nonce, URI, and the response–which is the MD5 of the two previous hashes combined

---

FORMS-BASED AUTHENTICATION

This is the most common type of web authentication, and it works by presenting a user with an HTML form for entering his/her username and password, and then sends those values to the server for verification. Some things to note:

- The login information should be sent via POST rather than GET
- The POST should be sent over HTTPS, not in the clear
- Ideally, the entire login page itself should be HTTPS, not just the page that the credentials are being sent to

Shown below is a typical structure of a login form (this one from wordpress.com):

```
<form name="loginform" class="login-form" id="adminbarlogin" action="https://e
```

Notice that the `action` URL is HTTPS, which means that the credentials entered into the form will be sent encrypted to that page.

INTEGRATED WINDOWS AUTHENTICATION (IWA)

IWA isn't an authentication protocol itself, but rather a means of assigning a preferential order to various protocols, such as Kerberos, NTLMSSP, and SPNEGO.

NTLM AUTHENTICATION (NTLMSSP)

This is being replaced with Kerberos now, but it's still out there.

- Client sends a Type 1 message telling the server what it supports in terms of key sizes, etc.
- The server responds with its own list of supported values, as well as a pseudo-randomly generated challenge in a Type 2 message
- The user concatenates his credentials with the challenge, implements MD5 and DES, and sends the response back to the server in a Type 3 message

HTTP REQUESTS

There are four parts to an HTTP request:

1. **The Request Line:** the method, the URL, the version of the protocol
2. **The Request Headers [OPTIONAL]**: a series of lines (one per) in the format of name, colon(:), and the value of the header.
3. **A Blank Line:** required, worth mentioning by itself.
4. **The Request Body [OPTIONAL]:** Used in POST requests to send content to the server.

REQUEST METHODS

Method	Description
GET	This most-commonly-used method is used mostly to retrieve something from the server, but it can also be used to send information via parameters. There is no body in a GET request, but the blank line is still sent after the headers. GET requests are often logged and are sent in referrer headers, so no sensitive information should ever be sent using this method.
POST	This method *sends* performs actions. The content sent can be located either in the URL query string or within the body of the message. POST should always be used to perform actions on the server, such as logging in, etc.
TRACE	Tells the server to echo back whatever is sent by the client. Highly significant for Reflected XSS, obviously.
PUT	Sends data to the server like POST, but as an upload rather than for processing. Think of it like FTP's PUT command.
HEAD	Just like a GET request, only it returns just the headers of the response.
DELETE	Deletes stuff.
OPTIONS	Retrieves a list of the methods supported by the

	server.
CONNECT	Used to connect to resources through a proxy.

HTTP RESPONSES

There are four parts to an HTTP response:

1. **The Response Line:** the version of the protocol, the status code, the status code description (OK, Not Found, etc.)
2. **The Response Headers:** a series of lines (one per) in the format of name, colon(:), and the value of the header.
3. **A Blank Line:** required, worth mentioning by itself.
4. **The Response Body:** contains the response from the server.

STATUS / RESPONSE CODES

Here are the main categories:

- 100's :: Informational
- 200's :: Success
- 300's :: Redirection
- 400's :: Client Error
- 500's :: Server Error

And here are some of the more common ones that are related to security:

RESPONSE CODE	DESCRIPTION
200 – OK	This most-commonly-seen response code, and it means that the resource you asked for was there, and that it has been returned to you.
301 – Moved Permanently	Tells the client that the resource that was asked for has been moved permanently, and supplies a new URL that should be used instead from now on via the `Location:` header.
302 – Found (Used to be "temporary-redirect")	This is the most common redirect code, but it actually has multiple sub-types that have been broken out in HTTP 1.1 (303/307); most servers still use 302, however, even though they should be using 303 or 307 instead.
303 – See Other	The proper, HTTP 1.1 way of sending a client to another page, indicated in the `Location:` header); tells the browser to get the new page via GET,

	regardless of what the original method was; commonly used after POST, e.g. being sent to your account_summary page after logging in; see 307 also.
304 – Not Modified	Tells the client that the resource hasn't been modified since last requested; works based on the client sending the `If-Modified-Since` header beforehand to check against.
307 – Temporary Redirect	Tells the client that the resource is somewhere else, but unlike the 303 it requires the method to be the same for the new request. See 303 also.
400 – Bad Request	Indicates bad syntax sent by the client. Often the result of mangling something while using an intercepting proxy, or having an inconsistent Internet connection that borked what the server received.
401 – Unauthorized	This is like saying you're *currently* not authorized, but it's sent along with a `WWW-Authenticate` header that tells you how you can try to prove yourself. This is in contrast to the 403, which just says "no".
403 – Forbidden	This is saying three things. 1) the thing you asked for is there, 2) you don't have access to it, and 3) you're not allowed to ask for access to it. This is in contrast to the 401 where you're given the option to authenticate.
404 – Not Found	The thing you're asking for isn't there.
405 – Method Not Allowed	You asked for a resource using a method that the server does not allow.
408 – Request Timeout	The request took to long to complete, i.e. longer than the server was prepared to wait.
500 – Internal Server Error	A generic error given when nothing more specific is available.
501 – Not Implemented	The server doesn't understand or speak the requested method. Often seen with PROPFIND requests.
503 – Service	Twitter made this one famous.

Unavailable (Fail Whale)	

** A more complete list of status codes can be found here.*

HTTP HEADERS

Here are the main headers used by HTTP. For a more complete list (including non-security-related ones) look here.

GENERAL (USED BY EITHER SIDE)

NAME	DESCRIPTION
Connection	Tells the other side to either close or keep open the TCP connection after receiving the message, e.g.: `Connection: close`
Content-Type	The MIME type of the body content of the message, e.g.: `Content-Type: text/html; charset=utf-8`
Content-encoding	The encoding type of the body of the message, e.g.: `Content-Encoding: gzip`
Content-length	The size of the body in octets/bytes, e.g.: `Content-Length: 256`
Transfer-encoding	The encoding that was used to send the message, e.g.: `Transfer-Encoding: chunked` Chunked encoding is used to avoid having to define an accurate message size before sending dynamic content, as it breaks it into pieces followed by a chunk of zero bytes.

REQUEST HEADERS (SENT BY THE CLIENT)

NAME	DESCRIPTION
Cookie	Passes a cookie to the server that was previously set by Set-Cookie e.g.: `Cookie: C=CKSdlekgie738FlweF8; Ver=2;`
Host	This is the hostname that's found in the URL being requested; important because of servers hosting multiple domains; required in HTTP 1.1, e.g.: `Host: overcomingbias.com`
Referer	Used to tell the server what page the link was clicked from; should not be used to make security decisions as it is controlled by the client; was misspelled in the

	original spec and remains so today, e.g.: `Referer: http://site.com/first_page`
Authorization	Tells the server that it's sending authentication credentials of a certain type, e.g.: `Authorization: Basic BTxhZGRpbjpbcGAuINMlc2FtZC==`
User-Agent	Tells the server what kind of browser the client is using, e.g.: `User-Agent: Mozilla/5.0 (Plan 9)`
Accept	Tells the server what kind of content it'll accept, e.g.: `Accept: text/plain`
Accept-Encoding	Tells the server what kind of encoding it can accept, e.g.: `Accept-Encoding: compress, gzip`

RESPONSE HEADERS (SENT BY THE SERVER)

NAME	DESCRIPTION
WWW-Authenticate	Tells the server that the resource it requested requires authentication; returns a status code of 401 e.g.: `WWW-Authenticate: Basic`
Set-Cookie	Sets a cookie on the browser that will be submitted in all later requests; the most common "state" mechanism for HTTP, e.g.: `Set-Cookie: Hash=DHCndkd83Iuw09; Ver=2;`
Server	Tells the client information about the server; a good way to give an attacker information about how to hurt you, e.g.: `Server: Apache/1.3.27 (Unix) (Ubuntu/Linux)`
Cache-Control	New in HTTP 1.1; used to tell the browser how to handle caching, e.g.: `Cache-Control: max-age=3600, must-revalidate`
Pragma	The old way of telling clients how to handle caching; is ignored by many browsers; `Cache-Control` or `Expires` should be used instead.
ETag	An identifier for a specific version of a resource to be used with the `If-None-Match` header. Basically it's a hash that the client can send back to say, "Here's my version of this specific resource; do you have an updated one?", e.g.: `ETag: "937065cd8d287d8af7ad7082f109582h"`
Expires	Tells the client that the content being sent expires at the given time, at which point it will have to be requested again, e.g.: `Expires: Fri, 01 Jan 1900 00:00:00 GMT`

** Additional, excellent information on caching can be found here.*

HTTP PROXIES

There are three primary considerations when looking at how HTTP proxies work: 1) whether you're connecting to an HTTP vs. HTTPS resource, 2) whether the proxy is explicit or transparent, and 3) whether the proxy requires authentication.

1. **Connecting to an HTTP Resource**
   To connect to an HTTP resource through a proxy the client sends the full URL it wants to reach, including the protocol, host, and path. The proxy parses all that information and uses it to make its own new request on your behalf.
2. **Connecting to an HTTPS Resource**
   This won't work for secure resources because an SSL/TLS handshake needs to occur, and we don't generally want our corporate proxy seeing our bank details, or what have you. This is handled via the `CONNECT` method, which tells the proxy that it wants to be connected to the remote server at the TCP layer–not at the HTTP layer. The proxy returns a 200 response and from that point on you're talking to the remote server, and you can then perform your handshake with the endpoint rather than the proxy in the middle.
3. **Dealing with Transparent Proxies and Proxy Authentication**
   The issue with transparent proxies is that a 407 can't be used because the client doesn't even know it's dealing with a proxy.
4. **Proxies and Integrated Windows Authentication**
   Integrated Windows Authentication is one case that often has issues with proxies, but some vendors like BlueCoat have workarounds.

COOKIES

Cookies are used as a state mechanism, since none is built into HTTP natively. There are a few points worth noting regarding cookies:

- Cookies are critical to security because they involve the sever setting information on the client, which it assumes the client will send back unmodified. Many developers make that assumption, and as a result they often place sensitive information within them which attacks can use to break the security of the application.
- Cookies are set on the browser by the `Set-Cookie` header from the server.
- Cookies are meant to be transparent, so they're sent to the server every time the client makes a request.
- The server can set multiple cookies by simply sending multiple `Set-Cookie` header values. All cookie values sent to the client via `Set-Cookie` are then combined into a single cookie, which is then sent to the server with a `Cookie` header. Each cookie is separated by a semicolon (;).
- A major part of attacking web applications involves evaluating what sort of information is stored in cookies by the server, and determining whether it it can be deconstructed, manipulated, reconstructed, and re-sent to the server to gain unauthorized access.
- For security purposes, the following values can be included with cookies within the `Set-Cookie` header: **expires** (creates a persistent cookie, as omitting this value will create a cookie that will expire when the browser closes); **HttpOnly** is supposed to prevent javascript from accessing the given cookie, but this is not foolproof; **secure** cookies will be sent only over HTTPS connections; **path** defines a scope of validity for a given cookie, e.g. /account/; domain specifies the domain that the cookie can be used within–must be the same or a parent of the domain the cookie came from.

SUPPLEMENTAL INFORMATION

- In HTTP version 1.1 the `Host:` header is mandatory.

- The referer (sic) header was misspelled in the original spec, and it remains so in the actual protocol today.
- URLs are actually a specific type of URI, so they are not the same–despite what you may read or hear.

NOTES

[1] The one must-read book on Web Security is The Web Application Hacker's Handbook, by Dafydd Stuttard and Marcus Pinto.
[2] Most everything you need to know about caching can be found here.
[3] Wikipedia's HTTP article.