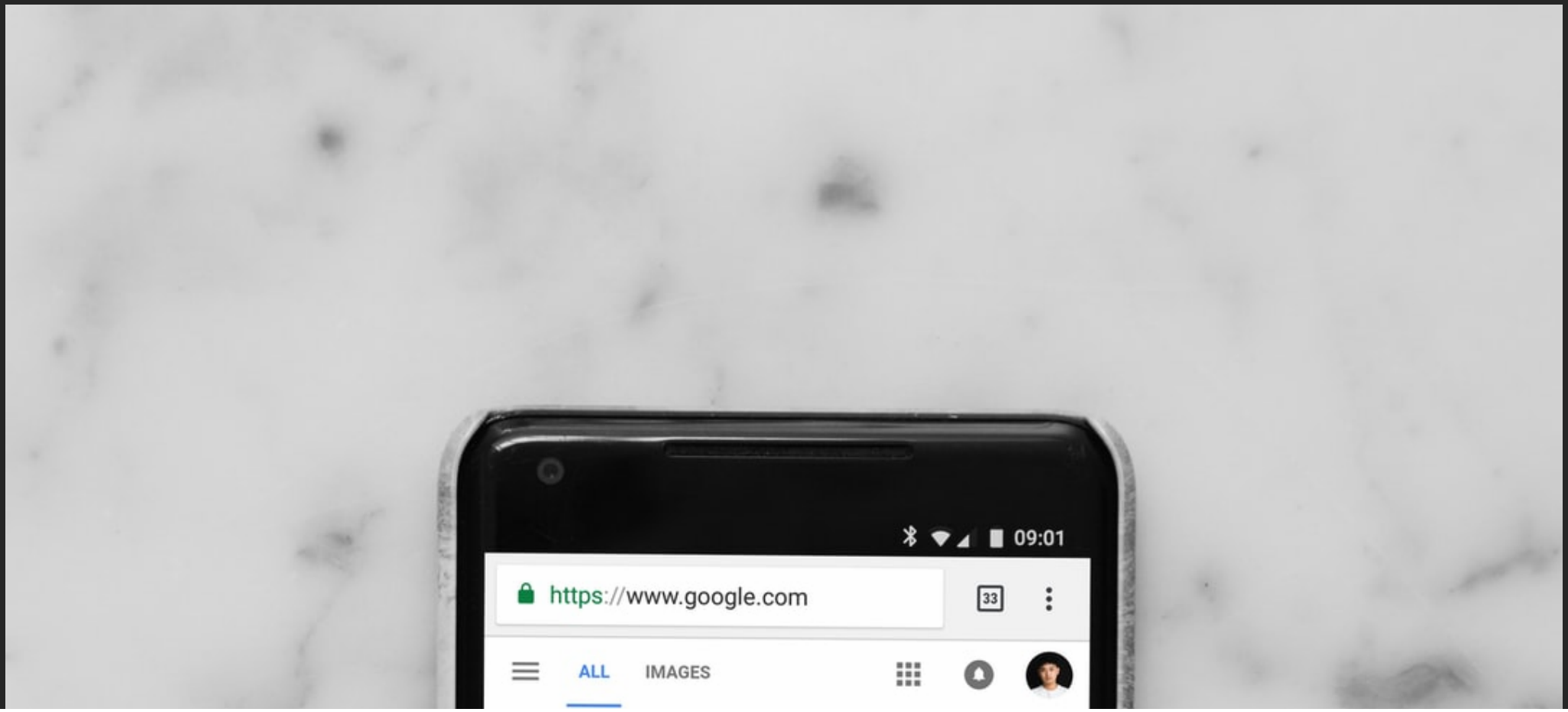
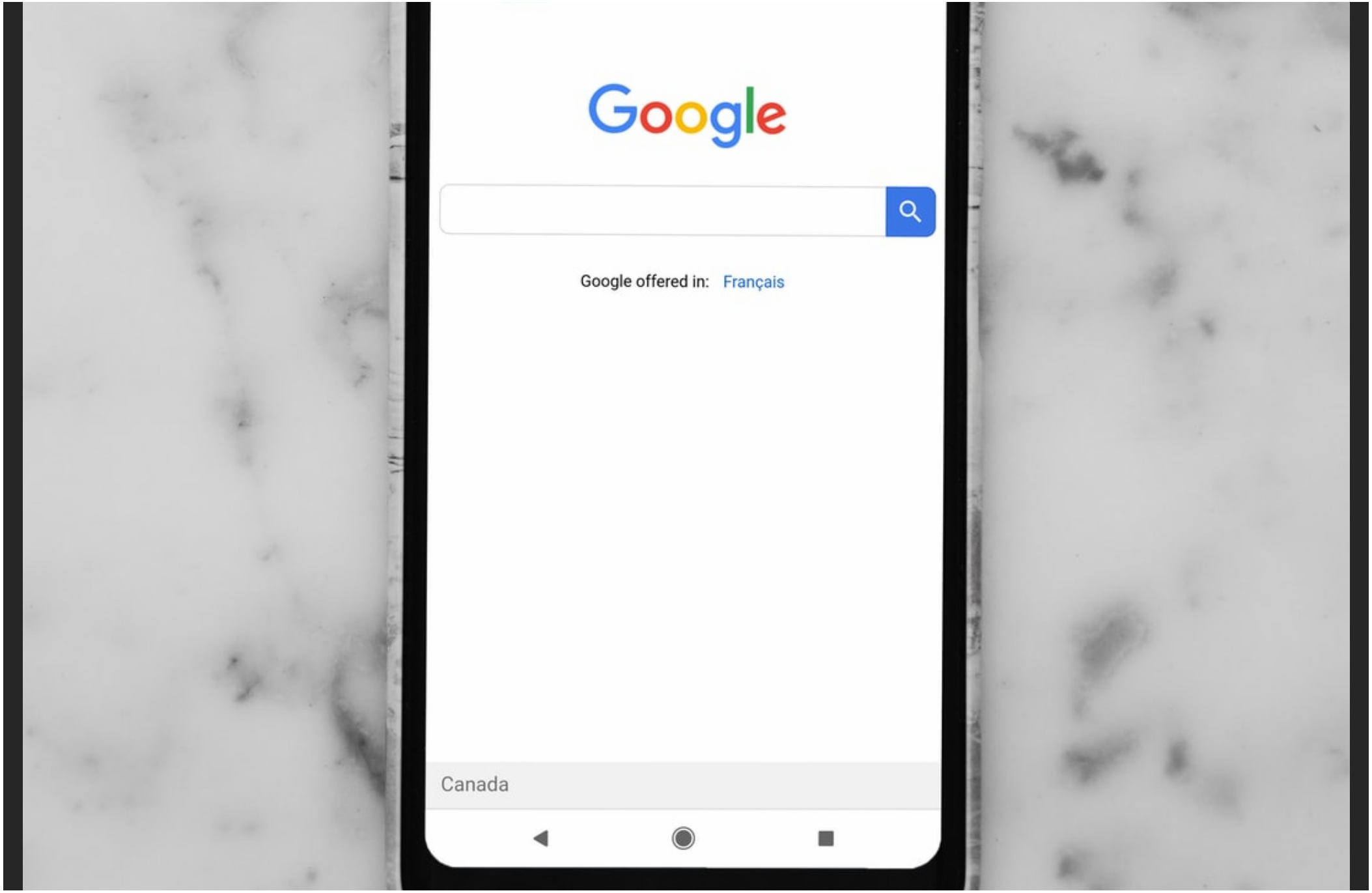




Navigating To A Web Site Step By Step

RYAN VILLARREAL / AUGUST 01, 2019

Hello Internet, let's talk about the Internet. Using a web browser and the Internet on a daily basis has become an integral part of our lives. Even getting to this blog post and the sharing of this blog post across multiple social media platforms takes a large amount of network traffic that happens behind the scenes. Hold onto your hats, because today we will be taking some time to discuss some of the common Web Application and Web Server designs that happen behind the scenes. This blog post *will* be lengthy, but if you make it through that's awesome! If you don't make it all the way through, well maybe you can reference back at a later date.

Finally, before beginning I want to credit Alex and all of those individuals on GitHub for the tremendous amount of work that has been put into the GitHub repo about `What Happens When`. It inspired and helped organize my blog post.

Let's answer the age old question (and in some cases interview question) what happens when you type `google.com` into your browser and hit enter. We won't be talking about keyboard electrical circuits or even operating system level interrupts instead we will focus mostly on requests at a network and application layer.

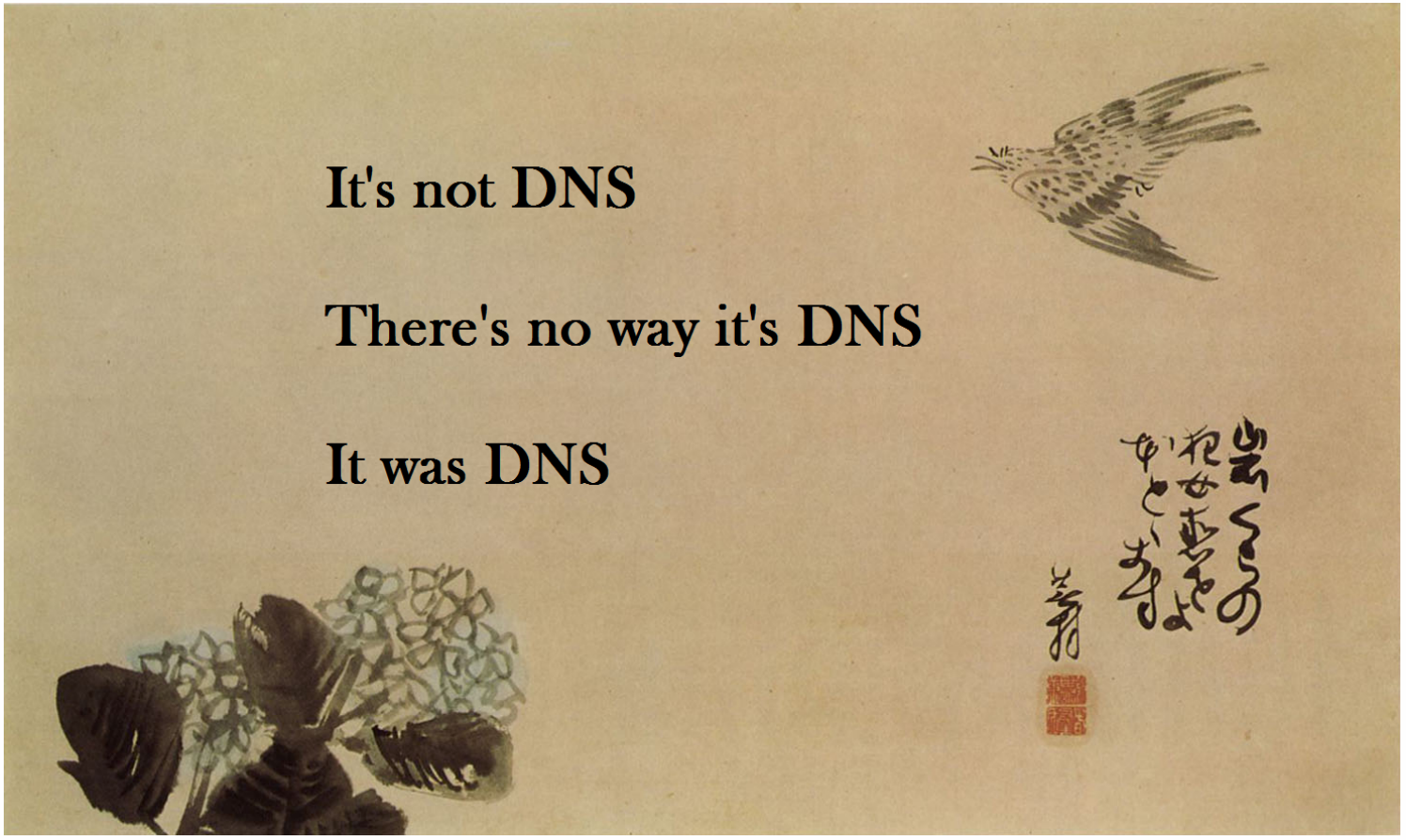
To begin you open a browser and for the sake of the article it does not open a home page immediately. The browser is sitting there, blinking the cursor in the address bar, waiting for input to instruct it how to perform. Upon striking the first key `g` the browser kicks into gear and begins trying to assist you in navigation. The auto-complete (dependent on your browser) will suggest various different choices. This auto-complete will use algorithms to sort and prioritize results based on search history, bookmarks, cookies, and popular searches from the

internet as a whole. As you continue you to type 'google.com' those searches and algorithms will continue to update to save you even a fraction of a second.

Since our address we typed in does not contain a protocol the browser checks to make sure the hostname for characters that are not allowed in a hostname. However, since we typed 'google.com' there won't be any non-valid characters. Again, there was no protocol specified which brings us to one of our first requests (possibly). The browser needs to know what protocol to use to access the site. The browser maintains a list of preloaded HTTP Strict Transport Security (HSTS) sites and if the address we have typed in is in that list it will know to only operate over a secure channel. However, if the site has never been visited or is not in the HSTS list then the browser will send the initial request over HTTP (cleartext). Why do we need HSTS? HSTS ensures that the browser is using the correct cryptography that is trusted by the web application. Otherwise if an attacker is able to get in the middle of a transmission could potentially downgrade a connect back to cleartext or an insecure cryptographic algorithm. If the initial HTTP request returns instructions to operate over HTTP it will remain that way.

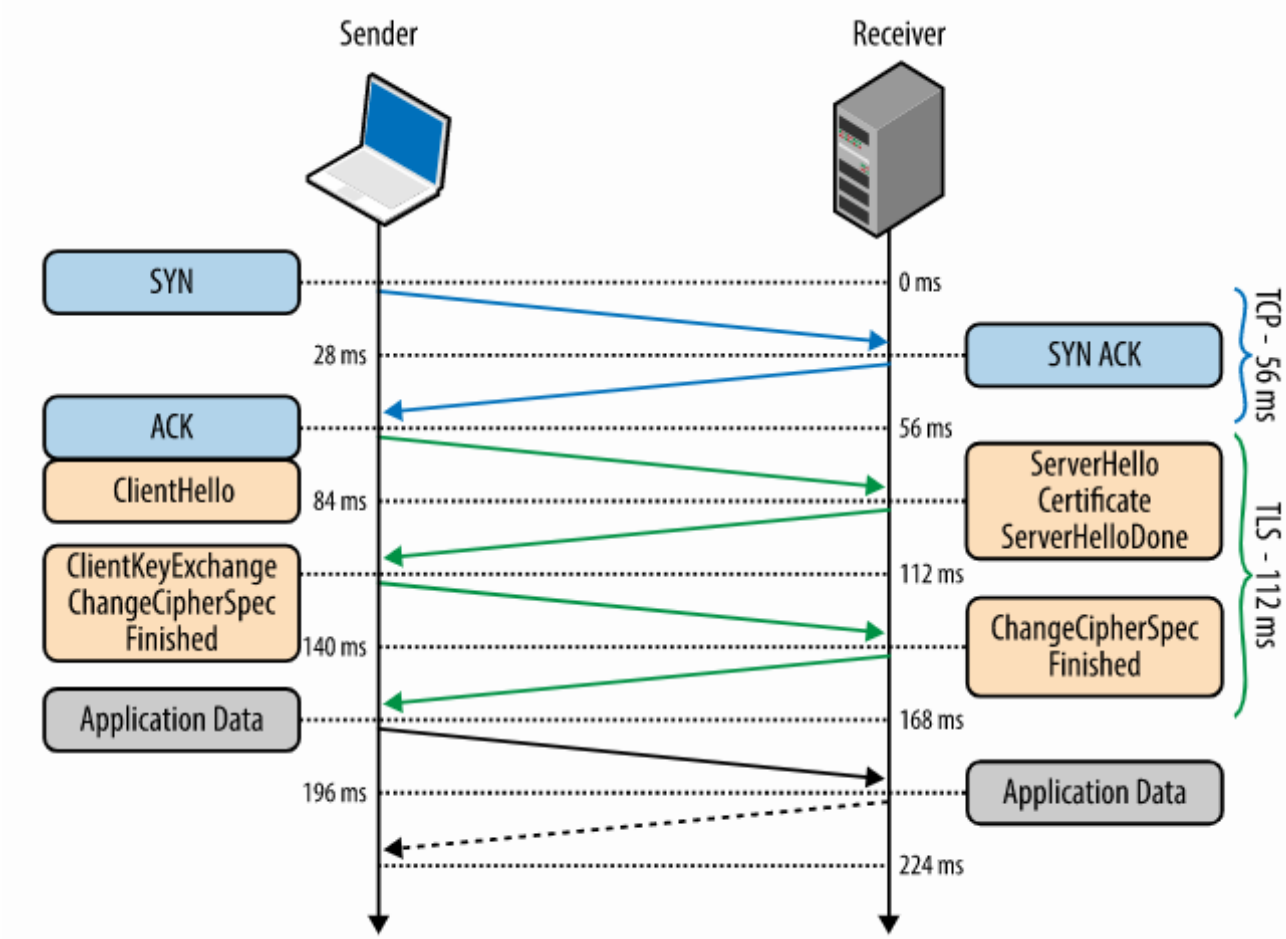
Alright we have figured out what protocol to use, but now we must discover what IP address to send the TCP/UDP connections to. Yes, you guessed it! DNS! It's not as simple as making a nslookup though. The browser will actually check to see if a domain is in the cache. You can view the browsers' internal DNS cache by navigating to `chrome://net-internals/#dns` (For chrome only, search for how to view in other modern browsers). If the site is not already in the browser cache it will make the call `gethostbyname` which will be handled by a library

function which might vary by OS. As mentioned earlier, operating system functions were not going to be discussed, therefore we will assume the OS has performed checks at different levels and returned the correct IP address to the browser.



Up to this point we have the IP address and the protocol to be used which directly correlated to HTTP port 80 and HTTPS port 443 (these ports can change, but generally speaking this is a standard). The packet will be passed down the network stack that will ultimately be sent to the

address and port. It should be noted at this time that the connection will only succeed if the network devices in between the client and server allow. This provides a layer of security by implementing firewalls or Web Application Firewalls that might block certain users from accessing the web server port. Thus initiates the TLS handshake. The SSL protocol was first developed by Netscape in order to protect personal data, and was later deprecated for the TLS protocol due to security concerns. The TLS handshake refers to the multiple steps taken in order to cryptographically secure the connection between the two parties. It should be noted that each of the exchanges during the TLS handshake require new packet round-trips between the client and the server.



The first message is from the client computer which says `ClientHello` with a list of cipher algorithms and compression methods available. The server should respond with `ServerHello` message that will contain several different pieces of information, such as: the SSL/TLS version, a selected cipher, a selected compression method, and the server's public key which will be signed by a certificate authority. The public key will be used by the client to encrypt the remainder of the handshake until a shared key can be agreed upon. In the third

packet the client will verify the server's digital certificate against it's list of trusted Certificate Authorities. These certificate authorities are preloaded in the browser, or could be installed into the browser. If the browser trusts the certificate provided by the server, the client will initiate the RSA or Diffie-Hellman key exchange. These cryptographic algorithms will help establish a symmetric key for encrypting the session. Once the encryption has been established the client will send a `Finished` message to the server using the hash that has been negotiated. Finally the server will generate its own hash, and will decrypt the client-sent hash to verify a match. If everything completes successfully the server will send its own `Finished` message. From this point forward the transmission will be encrypted using the symmetric key.

If you are using a modern browser that has been updated at some point since 2015 more than likely the browser will send a request to try and negotiate an upgrade with the server from HTTP/1 to HTTP/2 (formerly known as SPDY). HTTP/2 was developed by Google to try and improve page load speeds. There will be more information on HTTP/2 in future posts, but for now we are going to assume the web server only supports HTTP/1.

The HTTP request will look similar to the following:

```
GET / HTTP/1.1
Host: google.com
Connection: close
```

```
[other headers]
[empty line]
```

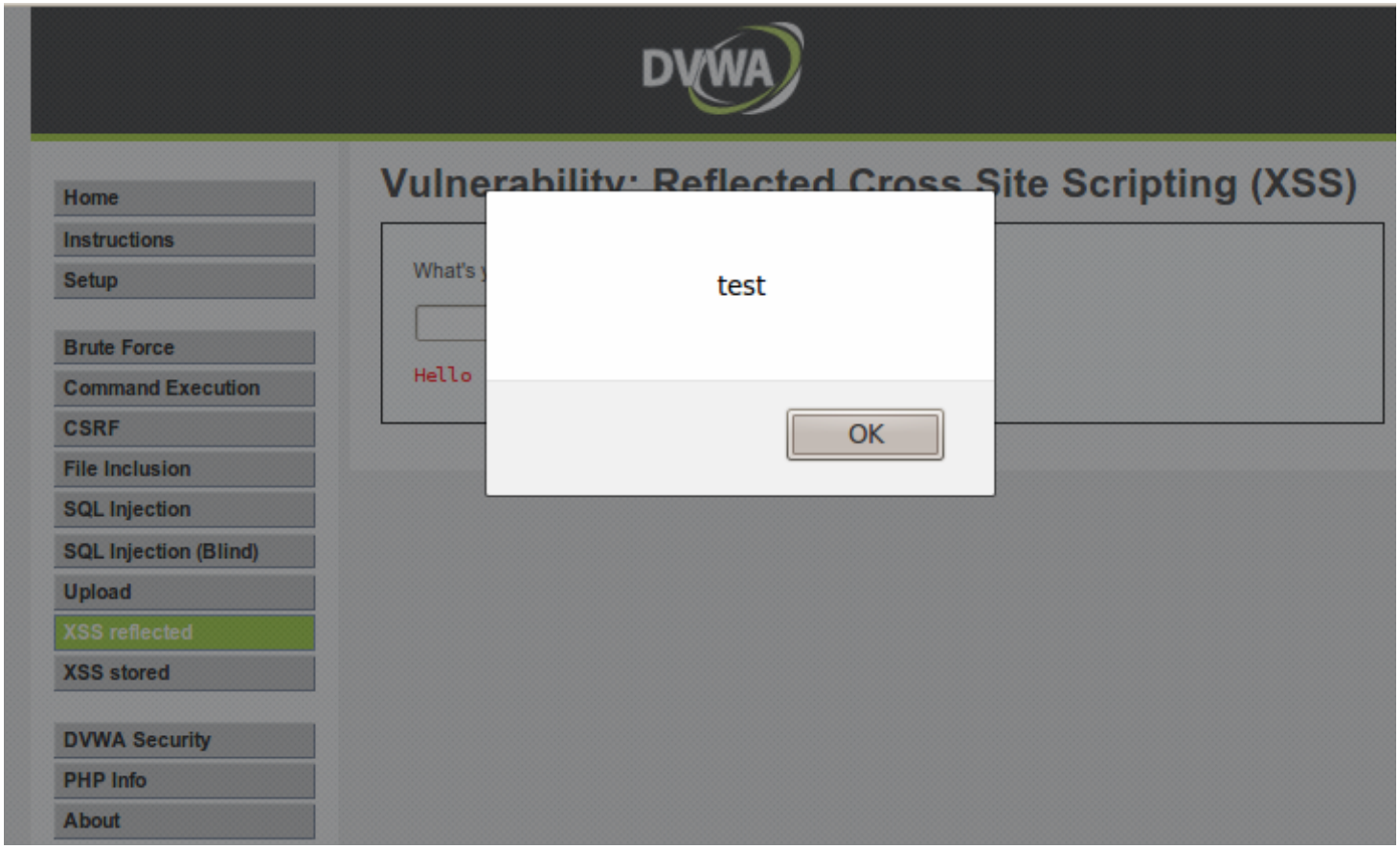
In the code block above the line item that says `Connection:close` is a header for the HTTP/1.1 protocol. Which signals the sender to specify options that are desired for that particular connection type. For example, if the client side sends a line item that says `Connection: keep-alive` and is supported by the server side it persist the connection. If the server side does not support the `keep-alive` connection type or responds with its own `close` connection the connection will be severed.

The line item that says `[other headers]` is referencing to other HTTP headers that allow the client and server to pass additional information with the request or the response. An HTTP header consists of its case-sensitive name followed by a colon `:`, the by a value without line breaks. There are many different HTTP headers that are accepted by most web browsers, but also developers can add custom proprietary headers by adding the `X-` prefix to any header. In this post we will discuss some of the more popular headers and their security intentions.

To begin we will discuss a security header that we have already described prior in this discussion. The HSTS header which forces communication using HTTPS instead of HTTP. HSTS is an important header in order to ensure full encryption of traffic between client and server. Another HTTP header is the Referer-Policy, which dictates the information sent in the referer header. A liberal Referer-Policy might produce undesirable levels of information

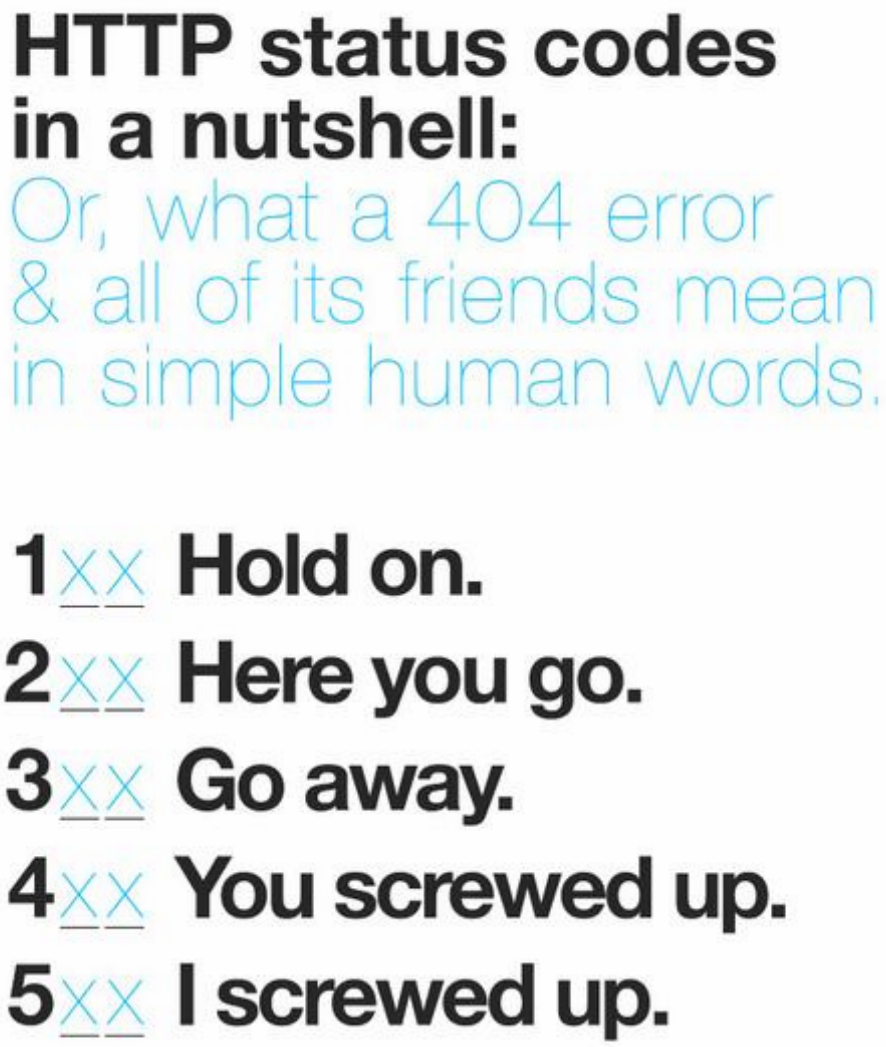
leakage to the client side. The leakage of information can assist an attacker in learning more about the attack surface and making more targeted specific attacks towards the web server. There are browser specific HTTP headers as well, one example is the X-XSS Protection header which is only recognized in Microsoft's Internet Explorer. This header will toggle on XSS filters that will help prevent certain categories of Cross-Site Scripting (XSS) attacks.

I have devoted an entire paragraph to a HTTP header that I feel is one of the more important ones. The Content Security Policy header. The primary benefit of CSP uses a strict policy of allowed scripts to execute on the page. Using a strict CSP policy, combined with other security best-practices such as adopting template systems with strict contextual auto-escaping, vulnerability scanning, and a manual security review, can significantly decrease the risk that an XSS bug will be introduced and that it can be exploited against users of modern browsers. Especially in cases where otherwise well-designed applications suffer from subtle, difficult to spot XSS issues, CSP can provide an important second layer of defense and protect the application's users.



Reviewing the code block of the HTTP request shown above has the final line `[other headers]` refers to the series of colon-separated key-value pairs formatted as per the IETF HTTP specifications. It should also be noted that in the code block above there is a single blank line at the end of the request. This instructs the server indicating that the content of the request is done. Depending on the web server and the requested page the server will respond with a response code denoting the status of the requested item. There are many different status codes, but can be broken down into general categories. Status codes beginning with 1xx

are for `Information` responses, 2xx is for `Successful` responses, 3xx is for `Redirection` messages, 4xx is for `Client` based error messages, and 5xx is for `Server` side based error messages.



Up to this point we have focused heavily on the client side interaction, but now we must switch sides and talk about how these requests are handled by the web server. The server side which is referred to most commonly as the HTTP Daemon (HTTPD) is responsible for handling requests/responses to the client. There are many well known web servers such as: Apache, nginx, IIS, and lighttpd. The web server will reserve a port to listen on for incoming requests. Upon the arrival of a request the server will begin to parse the line items of the request and perform various actions. On the first line will include the HTTP request method, these are accepted actions that are desired actions. Although the following methods are nouns, these requests are referred to as HTTP `verbs`. Some common verbs are described here:

```
GET - method requests a representation of the specified resource.
HEAD - method asks for a response without the response body.
POST - method used to submit an entity to the specified resource.
PUT - method replaces all current target resources with the  request
```

```
payload
DELETE - method deletes the specified resource
CONNECT - method establishes a tunnel to the server
OPTIONS - method describes the communication options for the target
resource
TRACE - method performs a loop-back test along the path to the target
resource
PATCH - method used to apply partial modifications to a resource.
```

It should be noted that just because these verbs are common and exist does not mean the web server will accept them. It is important to be able to deny certain verbs otherwise an attacker could easily DELETE or PUT to destroy or gain access to the web server.

Back to the web server side, parsed the request method from the verb supplied it will verify the domain, in our instance the `google.com` address. Also the requested path or page in our case `/` was passed, if no page or path is specified it will default with the root directory represented by the / backslash. The server can now verify that the hostname that was sent in the request corresponds with the Virtual Host configuration. The term `Virtual Host` refers to the practice of running more than one web sites on a single machine or IP address. The configuration file will correspond the domain name passed with the appropriate directory. Once the correct root directory for that domain is found, the next step the web server performs is to check and see if the HTTP verb is accepted. Another layer of security is to verify that the client is allowed to use the method, for example the web server could include

instructions that only allow internal IP addresses to perform the PATCH method. The web server can now check against the other configuration instructions known as modules. Modules can be used by the webserver to perform certain actions based on requests, for example the Apache mod_rewrite module uses a rule-based engine to map a URL to a file-system path if not previously done in the Virtual Hosts file. After all of the previous steps the web server will pull the content that corresponds to the request in our case the `/` or commonly known as the `/index` page, the page will be parsed according to the handler. This means that if the web server is configured to deliver PHP files it will interpret the index file, and stream the output back to the client.

Assuming all resources have made it back to the client side successfully the browser will undergoes the parsing and rendering steps of the files. Whatever browser you are using its purpose is to present the web resource delivered by the web server. Typically this is an HTML document, but other resources that modern browsers can support are: PDFs, images, and JSON data. The rendering engine is typically different based on the browser, but for the sake of this example we will assume the client is using Google Chrome browser. In that case the rendering engine is now the Blink engine, previously used was WebKit created by Apple for use in their own browser, Safari. The rendering engine will typically get the contents of the requested document from the network layer and will split the data in 8 KB chunks. These chunks will then be parsed into what is known as a parse tree. The output of the parse tree is a DOM element and attribute nodes. DOM is short for Document Object Model, which is the logical structure of documents and the way a document is accessed and manipulated.

Once the browser finishes parsing the standard HTML language it will begin fetching external resources linked to the page such as CSS, images, JavaScript files, etc. At this stage the browser marks the document as interactive and starts parsing scripts that are in "deferred" mode: those that should be executed after the document is parsed. The document state is set to "complete" and a "load" event is fired. It should be noted there is never an "Invalid Syntax" error on an HTML page. Browsers will attempt to fix any "errors" that creep up, and if unable to fix will simply continue on.

To conclude this post the browser now has everything needed from a network request standpoint to fully render the page with instructions from the HTML and styling sources such as CSS. After the rendering the browser will execute any JavaScript code as a result of any timing mechanisms or user interaction. It should be noted as well that scripts can continually make network requests, but at this point we have navigated to `google.com`. What was it we were going to search for? I forgot at this point. Until next time continue browsing and learning!

References and Inspiration:

- https://github.com/alex/what-happens-when
- https://www.troyhunt.com/understanding-http-strict-transport/
- https://en.wikipedia.org/wiki/Moxie_Marlinspike#SSL_stripping
- https://hpbn.co/transport-layer-security-tls/

- https://stackoverflow.com/questions/28592077/what-is-the-difference-between-http-1-1-and-http-2-0
- https://tools.ietf.org/html/rfc2616
- https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
- https://csp.withgoogle.com/docs/why-csp.html
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Status
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- https://httpd.apache.org/docs/2.4/vhosts/index.html
- https://en.wikipedia.org/wiki/Browser_engine
- https://www.w3.org/TR/DOM-Level-2-Core/introduction.html

SHARE

TAGS: CERTIFICATES CYBERSECURITY HTTP/2 PHP WEB APPLICATION WEBAPP CORS XSS HTTP

— ABOUT RYAN VILLARREAL

DENVER, COLORADO TWITTER

NEXT

Information Gathering With Cobalt Strike

AUGUST 16, 2019

PREVIOUS

Atomic Red Team

JULY 30, 2019

— ABOUT —

Two cybersecurity professionals trying to get better at all things security.

— LATEST POSTS —

Information Gathering With Cobalt Strike
AUGUST 16, 2019

Navigating To A Web Site Step By Step
AUGUST 01, 2019

Atomic Red Team
JULY 30, 2019

— AUTHORS —

- Ryan Smith
- Bestest RedTeam
- Ryan Villarreal

— TAGS —

802.11 802.1X ACTIVE DIRECTORY ANTI-CSRF AUTOMATE AUTOMATION AWS BETA BETTERCAP BGP BITCOIN BLOODHOUND BLUE TEAM BURPSUITE BYPASS BYT3BL33D3R C2 CA CAPTURE THE FLAG CERTIFICATES CLOUD CLUSTER CME COBALT STRIKE COMMAND AND CONTROL

COMMAND LINE CONTAINER CORS CRACKMAPEXEC CSRF CTF CYBERSECURITY DETECTION DOCKER DOMAIN ADMIN DOMAIN CONTROLLER

DVWA ELEARNSECURITY ELK ELKSTACK ENUMERATION EWPT EXECUTIONPOLICY FREERADIUS GHOST GNU RADIO GOOGLE CLOUD GOPHISH

GRAPH THEORY HACKING HACKRF HASHCAT HIJACKING HTTP HTTP/2 IMPACKET INFORMATION GATHERING INTERNAL NETWORK

INTERNET OF THINGS JAVASCRIPT JUICESHOP JWT KALI LINUX KALI TOOLS KERBEROS LATERAL MOVEMENT LINUX MERLIN MICROSOFT

MICROSOFT OFFICE MINING NE0ND0G NEO4J NETWORKING NULL SESSION OFFENSIVE SECURITY OFFSEC OPEN REDIRECT OSWP OWASP

PASSWORD CRACKING PENETRATION TEST PENTEST PHISHING PHP PINEAPPLE PIXEL TRACKING PORTAINER POST EXPLOITATION POWERSHELL

PROTOCOLS PYTHON RADIO FREQUENCY RECON RED TEAMING RED-BARON REDTEAMING REPORTING REVIEW RF RFC RTL-SDR S3 SAMBA

SCANS SCAPY SCRIPTING SERVICE PRINCIPAL NAME SERVICES SHODAN SMB SMBCLIENT SOCIAL ENGINEERING SOFTWARE DEFINED RADIO SPN

SWARM SYSADMIN TERRAFORM TERRAFORMFUN TRAINING UUID VULNERABILITY SCANNING WARDRIVING WEB APP WEB APPLICATION

WEB TOKEN WEBAPP WIFI WIFU WIGLE WINDOWS WIRELESS WPA XSS

OPINIONS EXPRESSED ARE SOLELY OUR OWN AND DO NOT EXPRESS THE VIEWS OR OPINIONS OF OUR EMPLOYERS.