HTTPS Bicycle Attack

By Guido Vranken <guidovranken@gmail.com>

ABSTRACT

It is usually assumed that HTTP traffic encapsulated in TLS doesn't reveal the exact sizes of its parts, such as the length of a Cookie header, or the payload of a HTTP POST request that may contain variable-length credentials such as passwords. In this paper I show that the redundancy of the plaintext HTTP headers included in each and every request can be exploited in order to reveal the length of particular components (such as passwords) of particular requests (such as authentication to a web application). The redundancy of HTTP in practice allows for an iterative resolution of the length of 'unknowns' in a HTTP message until the lengths of all its components are known except for a coveted secret, such as a password, whose length is then implied. The attack furthermore exploits the property of stream-oriented cipher suites such as those based on Galois/Counter Mode that the exact size of the plaintext can be known to a man-in-the-middle.
The paper furthermore gives insight in how very small differences in the length of intercepted (encrypted) GPS coordinates can be used to estimate the location on the world map for a particular encrypted coordinate. Another example demonstrates that differences in length of intercepted (encrypted) IPv4 addresses are bound to specific IP ranges.
The paper concludes with a set of proposed mitigations against this attack.

Table of Contents

1. Introduction

1. On TLS side-channel leaks

It has long been known that SSL/TLS (from hereon referred to as TLS) is no silver bullet to obscure the behavior of a user on a network. While the sound configuration of both endpoints of a connection is understood to prevent the decoding from ciphertext to plaintext without having access to the private key(s), transactions conducted over a channel embedded in TLS leak various types of information. These side-channel leaks can be the can be the result of the delegation of actions required for proper data transmission on a network to protocols at a higher layer that offer no means of obscuring key information, such as source and destination IP's encoded in the Internet Protocol layer, and source port, destination port, payload fragmentation an so forth encoded in the TCP layer. Other types of side-channel information leaks are consist of variability introduced entirely outside of TLS's control, such as spatial and temporal discrepancies for different payloads. Moreover, the aggregate of the properties of packets sent back and forth between two TLS endpoints constitutes a sequence that may be unique linked to path and resource access on a web application. Some properties of a TLS session are left unobscured by design, such as the exact ciphersuite used and the exact length of the plaintext when stream ciphers are used.

A lot of research has been performed on how to stack up these different 'knowns' in order to meticulously reconstruct the user's actions, given that the encrypted streams are known to an observer who is or has been listening in on the 'secure' transmission between two endpoints.

In this paper I will show that for a presumably large subset of web applications, it is easy to infer the length of parts of the plaintext, or certain attributes thereof, from a recorded stream of encrypted messages. Having access to the private key is not necessary. In fact, the actual ciphertexts embedded in the stream are irrelevant to the deduction, and entry-level arithmetic suffices.

This attack has the property of being entirely passive. That is, unlike attacks such as BREACH (which relies on CSRF attacks), the attacker doesn't need to interfere with a user's session. While my attack typically reveals less detailed information than BREACH, its advantage of my attack lies in the fact that it cannot raise alarm bells, and that it can be applied retroactively; that is, *encrypted streams recorded years ago can still be picked apart in order to divulge confidential information*.

Furthermore, the attack requires some information about the victim to be known to the attacker. The more information the attacker knows, the more information about the victim's plaintexts can be deduced. Broadly speaking, the attack works by subtracting the lengths of known parts of the plaintext from the total plaintext size. If knowing the length of the user's password for a specific website is the attacker's objective, then the attacker must also know the user name belonging to that password, since user name and password are often sent together in an authentication process. The attack can reveal the length of the concatenation of the user name and the password. Subtracting the length of the user name from this value reveals the length of the password.

Another user property that is helpful to know is the browser used. This aids in predicting which headers a browser will send for various types of web resources. This shouldn't be too difficult to determine in a directed attack on a specific person, since just a single HTTP (ie., insecure) request will reveal the User-Agent string.

The name *TLS Bicycle Attack* was chosen because of the conceptual similarity between how encryption hides content and gift wrapping hides physical objects. My attack relies heavily on the property of stream-based ciphers in TLS that the size of TLS application data payloads is directly known to the attacker and this inadvertently reveals information about the plaintext size; similar to how a draped or gift-wrapped bicycle is still identifiable as a bicycle, because cloaking it like that retains the underlying shape. The reason that I've named this attack at all is only to make referring to it easier for everyone.

2. A note on TLS records and cipher modes

A TLS record, in which encrypted data is encapsulated, has two fields that are of importance to the attacker. One is the Content Type field. In this document I will focus exclusively on content types with the values 23 (0x17), since these types of records are used to embed the actual encrypted payloads. The other field of interest to us is the 'length' field. This is a 16-bit field which reflects the exact size in bytes of the encrypted payload.

Another important property to be aware of is the ciphersuite being used, in particular whether it concerns a block cipher or a stream cipher. I will focus on stream ciphers, which have the convenient property that their output lengths corresponds 1:1 with the input (plaintext) size, although they do not necessarily have the same size. That is, each byte added to the plaintext results in one byte added the encrypted message, though the encrypted message may be larger than the plaintext (ie., encryption may add a constant number of bytes, such as overhead).

2. Overview of the attack

The attack consists of two components.

1. Fingerprinting

Once an encrypted stream has been intercepted, the attacker must employ some form of fingerprinting in order to know which resources in a web application were accessed. There are many ways this can be achieved. This paper will not elaborate extensively on fingerprinting strategies. For demonstration purposes I will be employing a simple fingerprinting mechanism, which consists of taking the full sequence of payload lengths of requests from the client to the server and calculating the Pearson correlation coefficient of each sub-sequence with a precomputed sequence in order to locate the user's retrieval of an authentication page.

For instance, loading a page that consists of a couple of JavaScript, CSS and image files will be reflected as a sequence of requests from the browser to the server with a distinct size. This sequence of distinct sizes must computed by the attacker before the attack is executed. This means that the attacker will have to load the page in their browser and record the size of each request. This precomputation will serve as a template. The attacker can compute the Pearson correlation coefficient from this template sequence and the sequences found in the recorded encrypted stream. Once a 1:1 match is found, the attacker can safely assume that the client has been accessing the same page at this point in the encrypted stream.

The method can be summarized as follows:

Let S be a sequence of the values of the 'length' field of TLS Application Data records of HTTP requests (not responses) to a particular web application (identified by its IP address and the hostname encoded in the TLS SNI extension). Each HTTP request corresponds to a separate TLS Application Data payload.

Let T be the sequence of request sizes that is unique for access to a particular resource.

Loading wordpress/wp-login.php on a WordPress installation implies the loading of other resources and results in five requests with the following sizes:

URI	Total size of corresponding HTTP request
https://localhost/wordpress/wordpress/wp-login.php	368
https://localhost/wordpress/wordpress/wp-includes/css/buttons.min.css?ver=4.4	409
https://localhost/wordpress/wordpress/wp-includes/css/dashicons.min.css?ver=4.4	411

https://localhost/wordpress/wordpress/wp-admin/css/login.min.css?ver=4.4	404
https://localhost/wordpress/wordpress/wp-admin/images/wordpress-logo.svg?ver=20131107	454

In this case T = [368, 409, 411, 404, 454].

```
from scipy.stats import pearsonr
import random

S = [   327, 470, 453, 351, 399,
        368, 409, 411, 404, 454,
        390, 430, 458, 318, 305]
T = [368, 409, 411, 404, 454]

for i in xrange(len(S) - len(T) + 1):
    if pearsonr(T, S[i:i+len(T)]) == (1.0, 0.0):
        print "Access to wp-login.php detected at element %i in sequence S" % i
```

In this example the sequence T is embedded in sequence S, which will be detected by comparing T to each sub-sequence in S of the same length as T (5).

Note that it is the Pearson correlation coefficient is not a one-to-one numeric comparison.

Setting T to these values:

```
[920.0, 1022.5, 1027.5, 1010.0, 1135.0]
```

(in which each original value has been multiplied by 2.5) does not hamper detection of the sub-sequence. This is quite useful because the intercepted, encrypted sequence might stem from plaintext with headers of a different length than the attacker's precomputed sequence. For instance, the plaintext of the intercepted stream might contain a different User-Agent header.

2. Length deduction through subtraction

The attack works on the basis of information deduction through length subtraction. Broadly speaking, it deduces the length of a single *unknown* (such as a password) by subtracting the known length of a payload from the total length of the payload.

The deduction process consists of two components:

1. The preparation: the attacker must use the application themselves and record the exact requests their browser sends for specific actions (such as authentication)

2. The analysis: using addition and subtraction of knowns and unknowns in order to determine the length of a single unknown. This is an iterative process.

Step 1: preparation

The attacker can load wordpress/wp-login.php in their browser and see that the following files are loaded as well:

```
/wordpress/wordpress/wp-includes/css/buttons.min.css?ver=4.4
/wordpress/wordpress/wp-includes/css/dashicons.min.css?ver=4.4
/wordpress/wordpress/wp-admin/css/login.min.css?ver=4.4
/wordpress/wordpress/wp-admin/images/wordpress-logo.svg?ver=20131107
```

The requests look like this:

```
GET /wordpress/wordpress/wp-includes/css/buttons.min.css?ver=4.4 HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:43.0) Gecko/20100101
Firefox/43.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://localhost/wordpress/wordpress/wp-login.php
Cookie: wordpress_test_cookie=WP+Cookie+check
Connection: keep-alive


GET /wordpress/wordpress/wp-includes/css/dashicons.min.css?ver=4.4 HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:43.0) Gecko/20100101
Firefox/43.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://localhost/wordpress/wordpress/wp-login.php
Cookie: wordpress_test_cookie=WP+Cookie+check
Connection: keep-alive


GET /wordpress/wordpress/wp-admin/css/login.min.css?ver=4.4 HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:43.0) Gecko/20100101
Firefox/43.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://localhost/wordpress/wordpress/wp-login.php
Cookie: wordpress_test_cookie=WP+Cookie+check
Connection: keep-alive


GET /wordpress/wordpress/wp-admin/images/wordpress-logo.svg?ver=20131107 HTTP/1.1
Host: localhost
```

```
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:43.0) Gecko/20100101
Firefox/43.0
Accept: image/png,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://localhost/wordpress/wordpress/wp-admin/css/login.min.css?ver=4.4
Cookie: wordpress_test_cookie=WP+Cookie+check
Connection: keep-alive
```

Trying to log in as user 'guido' and password 'password' yields the following request:

```
POST /wordpress/wordpress/wp-login.php HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.5
Connection: keep-alive
Cookie: wp-settings-time-1=1451442362; wordpress_test_cookie=WP+Cookie+check
Host: localhost
Referer: https://localhost/wordpress/wordpress/wp-login.php
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:43.0) Gecko/20100101
Firefox/43.0
Content-Type: application/x-www-form-urlencoded
Content-Length: 126

log=guido&pwd=password&wp-submit=Log+In&redirect_to=https%3A%2F%2Flocalhost
%2Fwordpress%2Fwordpress%2Fwp-admin%2F&testcookie=1
```

Step 2: analysis

Once the attacker isolates a POST request to wp-login.php by the victim using fingerprinting, then the following information is known:

```
1. POST /wordpress/wordpress/wp-login.php HTTP/1.1
2. Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
3. Content-Type: application/x-www-form-urlencoded
4. Content-Length: 1xx
<UNKNOWN HEADERS>
5.
6. log=guido&pwd=&wp-submit=Log+In&redirect_to=https%3A%2F%2Flocalhost%2Fwordpress
%2Fwordpress%2Fwp-admin%2F&testcookie=1
```

Line 1: because fingerprinting was employed, the attacker knows that the resource being accessed is wp-login.php. The rest of the line can be known from the preparation (step 1).

Line 2: If it is known that the user's browser is Firefox, the attacker can assume that the browser sends this Accept header.

Line 3 and 4: Known from the preparation (step 1).

Line 5: CRLF to separate headers and data

Line 6: If the attacker knows beforehand that the username is 'guido', the rest of the string can be reconstructed from the preparation (step 1).

The length rest of the headers, such as Cookie and User-Agent might not be directly known to the attacker. For this the attacker has to look at other requests made to the same web application.

In the encrypted stream, isolate the request for /wordpress/wordpress/wp-includes/css/dashicons.min.css?ver=4.4 and take the plaintext size from the request.

From this size, subtract the length of the lines in blue:

```
GET /wordpress/wordpress/wp-includes/css/dashicons.min.css?ver=4.4 HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:43.0) Gecko/20100101
Firefox/43.0
Accept: text/css,*/*;q=0.1
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: https://localhost/wordpress/wordpress/wp-login.php
Cookie: wordpress_test_cookie=WP+Cookie+check
Connection: keep-alive
```

The length of the first blue line is 77 bytes (this includes carriage return + line feed).

The length of the second blue line is 28 bytes (this includes carriage return + line feed).

The sum of these values is 77 + 28 = 105 bytes.

Say the plaintext size of the intercepted request for dashicons.min.css?ver=4.4 is 409 bytes. Then 409 – 105 = 403. This is exactly the size of the length of the unknown headers in the POST request:

```
1. POST /wordpress/wordpress/wp-login.php HTTP/1.1
2. Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
3. Content-Type: application/x-www-form-urlencoded
4. Content-Length: 1xx
<UNKNOWN HEADERS> length = 304 bytes
5.
6. log=guido&pwd=&wp-submit=Log+In&redirect_to=https%3A%2F%2Flocalhost%2Fwordpress
%2Fwordpress%2Fwp-admin%2F&testcookie=1
```

Now take the plaintext size of the intercepted POST request issued by the client, and from it subtract all the known lengths:

```
1. POST /wordpress/wordpress/wp-login.php HTTP/1.1 length = 49 bytes (includes
CSRF)
2. Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 length =
73 bytes (includes CSRF)
3. Content-Type: application/x-www-form-urlencoded length = 49 bytes (includes
CSRF)
4. Content-Length: 1xx length = 21 bytes (includes CSRF)
<UNKNOWN HEADERS> length = 304 bytes
5. length = 2 bytes (CSRF)
6. log=guido&pwd=&wp-submit=Log+In&redirect_to=https%3A%2F%2Flocalhost%2Fwordpress
%2Fwordpress%2Fwp-admin%2F&testcookie=1 length = 118 bytes
```

The total length of the knowns is 49+73+49+21+304+2+118 = 616 bytes

Total plaintext size – 616 = length of password.

3. Implications

The attack does not reveal plaintext contents, but only lengths of parts of the plaintext. On the upside, the attack is entirely passive and may be executed retro-actively (ie., on encrypted streams recorded in the past). From access to web applications whose authentication process is vulnerable to this attack, the password length of a targeted user can be known, which can give the attacker an indication of what the password is, or indicate the feasibility of a brute-force attack. It may also be executed on a larger scale on TOR exit nodes, VPN's, proxies and other Internet traffic conduits in order to detect weak or short passwords susceptible to a brute-force or an attack based on a dictionary of often-used passwords.

The example described above extracts credentials from a POST request, but the range of application for this attack is much wider. It can also be used to extract the length of user names (from pages where all content is static or known except for a string “Welcome back, {username}”), or the user's account balance on pages in an online banking application, or the number of new messages in an online messaging system. One particular avenue that would be interesting to explore is HTTP API's. HTTP API's typically have a limited amount of unpredictable header lengths, and API's are designed to efficiently communicate isolated bits of information back and forth, so requests and responses are typically not cluttered by unpredictable dynamic content.

4. Other examples

1. Location leaks through encrypted GPS coordinates

1. Deducing location from GPS coordinate length

GPS coordinates consist of a (latitude, longitude) tuple. There are a total of 180 possible degrees latitude (-90 to 90) and a total of 360 possible degrees longitude (-180 to 180). The minus sign is used to denote negative degrees. The coordinates usually also have a floating point part. The floating point part typically consists of 6 digits, regardless whether it may be shortened (so 12.0 is written as 12.000000). From this it follows that latitude consists of at least 8 characters: a single digit before the dot, the dot, and 6 digits after the dot, for example 6.000000.

To put it differently, a latitude constists of these parts:

1) An optional minus sign -
2) an integer from 0 to 90; so either 1 or 2 digits
3) a mandatory dot
4) an integer consisting of 6 characters (digits)

A longitude consists of these parts:

1) An optional minus sign -
2) An integer from 0 to 180; so either 1, 2 or 3 digits
3) A mandatory dot
4) An integer consisting of 6 characters (digits)

The only variable fragment in either encoding is the second part (the number of degrees); the integer ranging from 0 to 90 and 180, respectively.

Length of degrees in characters	Ranges this length encodes as tuples
1	(0, 9)
2	(10, 99), (-9, -1)
3	(-99, -10)

Length of degrees in characters	Ranges this length encodes as tuples
1	(0, 9)
2	(10, 99), (-9, -1)
3	(100, 180), (-99, -10)
4	(-100, -180)

From these two sets, the following Cartesian product can be constructed:

```
[((0.0, 9.0), (0.0, 9.0)),
 ((0.0, 9.0), (10.0, 99.0)),
 ((0.0, 9.0), (-9.0, -1.0)),
 ((0.0, 9.0), (100.0, 180.0)),
 ((0.0, 9.0), (-99.0, -10.0)),
 ((0.0, 9.0), (-100.0, -180.0)),
 ((10.0, 99.0), (0.0, 9.0)),
 ((10.0, 99.0), (10.0, 99.0)),
 ((10.0, 99.0), (-9.0, -1.0)),
 ((10.0, 99.0), (100.0, 180.0)),
 ((10.0, 99.0), (-99.0, -10.0)),
 ((10.0, 99.0), (-100.0, -180.0)),
 ((-9.0, -1.0), (0.0, 9.0)),
 ((-9.0, -1.0), (10.0, 99.0)),
 ((-9.0, -1.0), (-9.0, -1.0)),
 ((-9.0, -1.0), (100.0, 180.0)),
 ((-9.0, -1.0), (-99.0, -10.0)),
 ((-9.0, -1.0), (-100.0, -180.0)),
 ((-99.0, -10.0), (0.0, 9.0)),
 ((-99.0, -10.0), (10.0, 99.0)),
 ((-99.0, -10.0), (-9.0, -1.0)),
 ((-99.0, -10.0), (100.0, 180.0)),
 ((-99.0, -10.0), (-99.0, -10.0)),
 ((-99.0, -10.0), (-100.0, -180.0))]
```

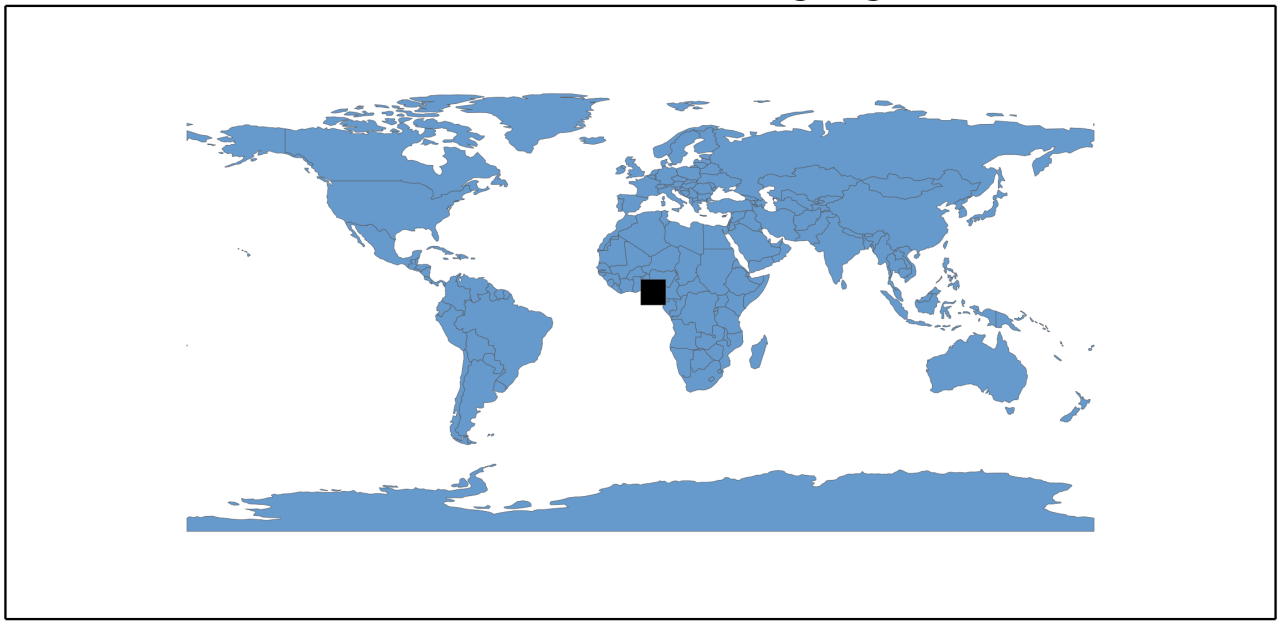
All the unique sums (ie., the unique sums of the character length of the degrees in both the latitude and the longitude):

[2, 3, 4, 5, 6, 7]

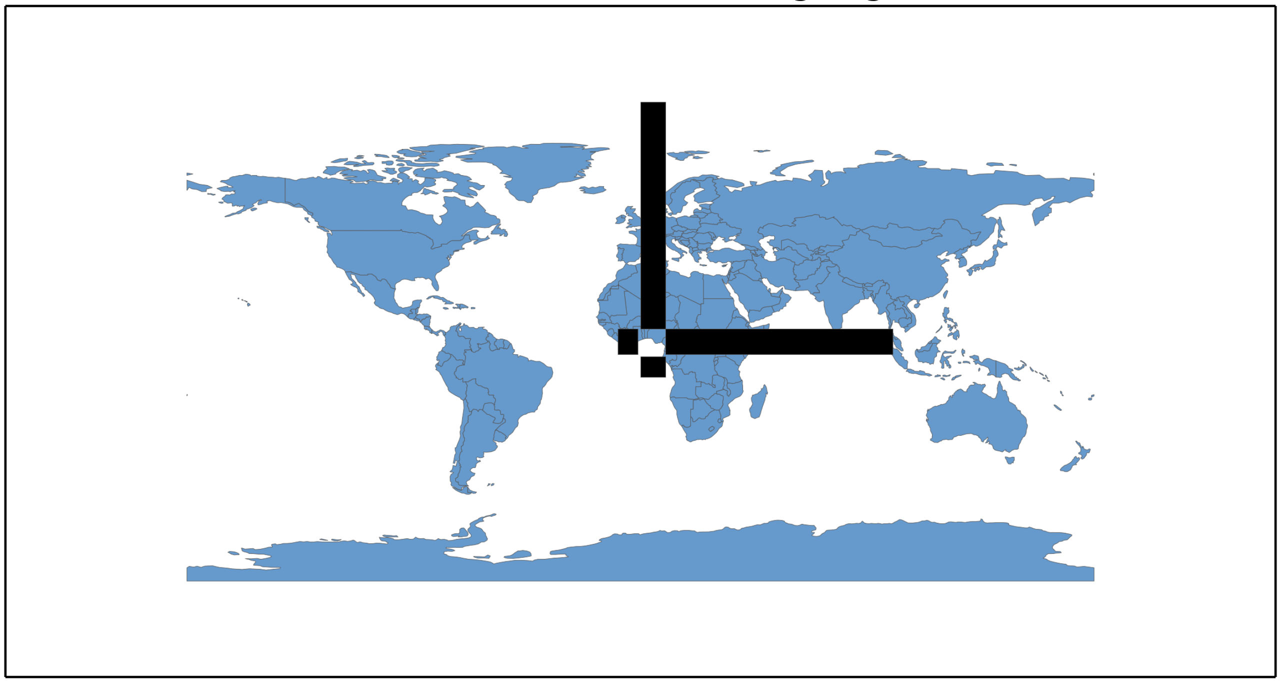
In other words, for each possible GPS coordinate encoded using the method described earlier, the sum of the number of characters which constitute the degrees latitude (eg, -45, which are 3 characters) and the degrees longitude (eg. 12, which are 2 characters) is either 2, 3, 4, 5, 6 or 7 (in this example it is 2 + 3 = 5).

From the tuples of ranges produced by the Cartesian product of each possible latitude and longitude length 6 different regions on the world map can be demarcated.

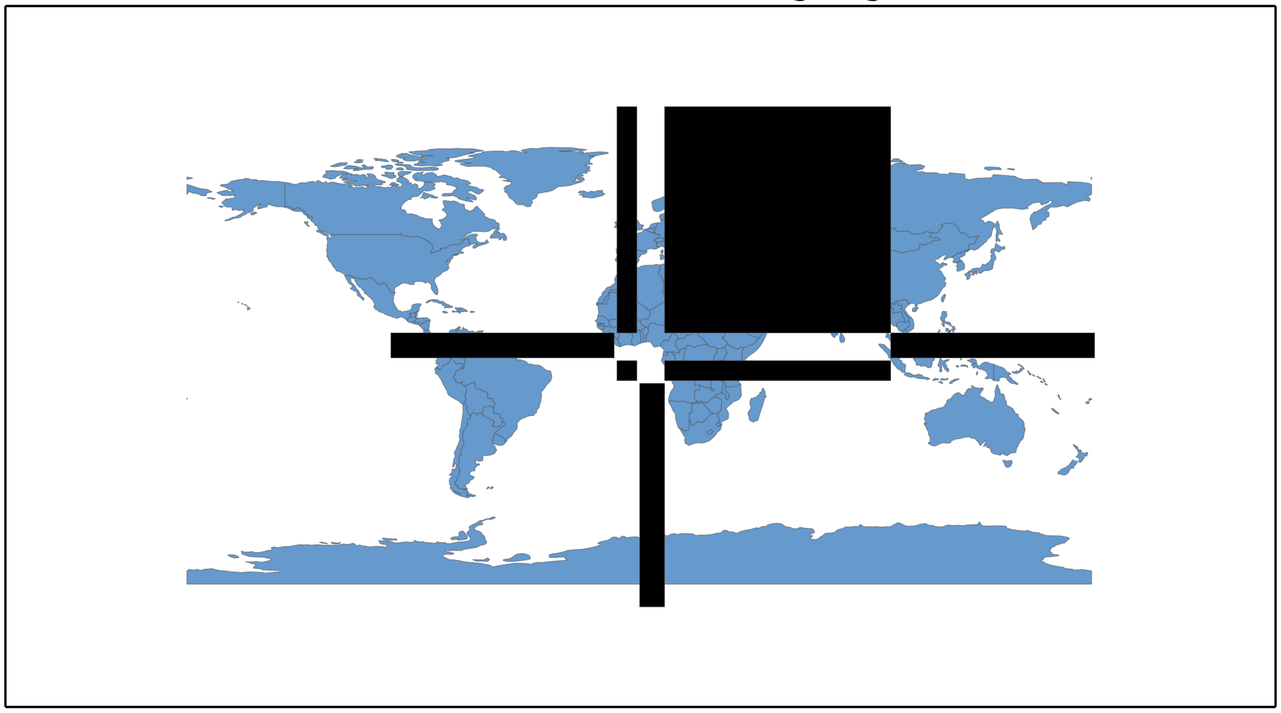
Sum of characters in lat and long degrees = 2



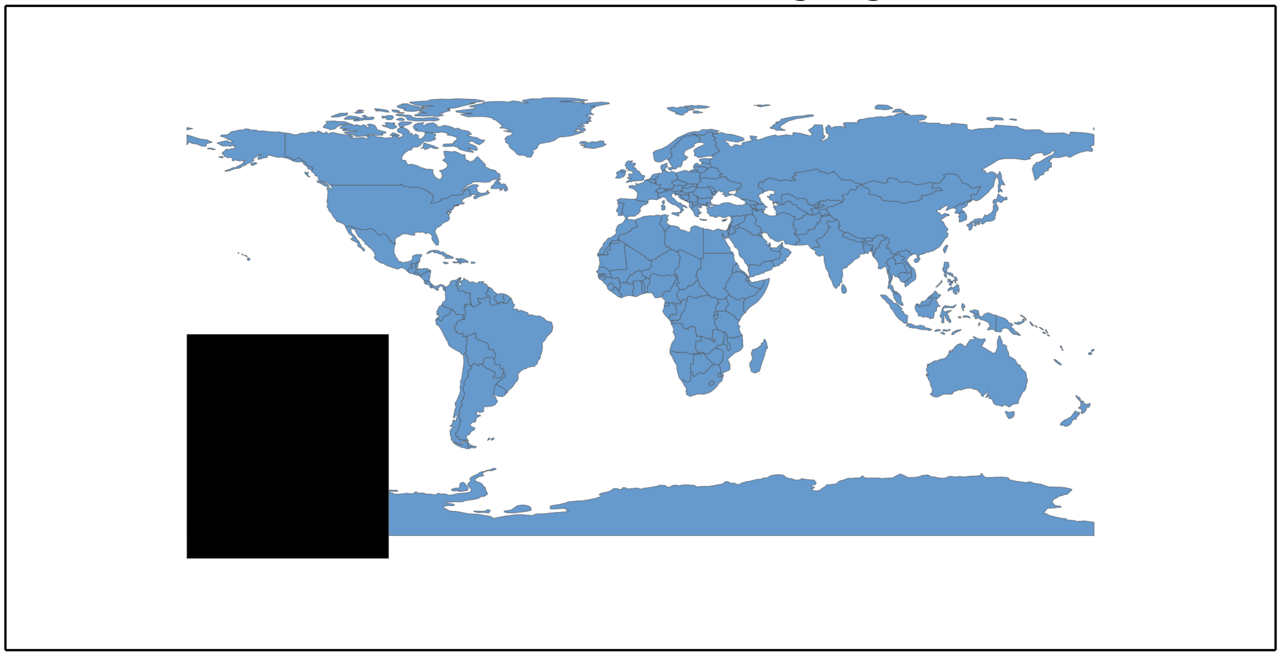
Sum of characters in lat and long degrees = 3



Sum of characters in lat and long degrees = 4



Sum of characters in lat and long degrees = 7



2. Exploitation

If the attacker knows that a certain application is leveraging a geocoding API to translate a set of GPS coordinates to a physical location, the deduction described previously may be applied to passively recorded and encrypted queries containing GPS coordinates to a geocoding API in order to determine a rough estimate of the region on the world map that the callee (the application invoking the API) is inquiring about. This could be used as part of a de-anonimization effort on TOR exit node or VPN outbound data. A mobile (or other) application that has retrieved its own GPS coordinates and subsequently sends it to an API in order to reverse this data to a human-readable location string, or to store its current location in the cloud, leaks information about its location.

For example, say an application is known to query Google's geocoding API in the following manner:

```
https://maps.googleapis.com/maps/api/geocode/json?latlng=<degrees>.<6
digits>,<degrees>.<6 digits>
```

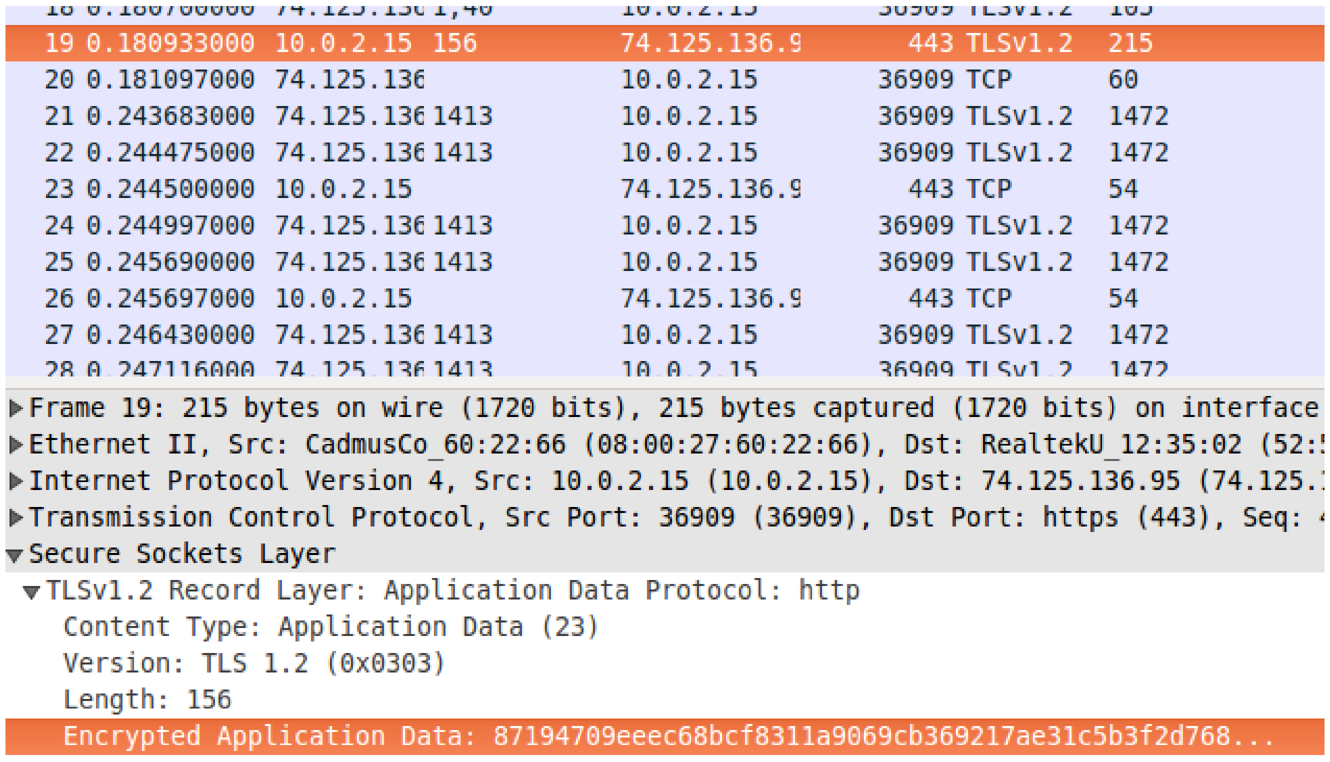
For example:

https://maps.googleapis.com/maps/api/geocode/json?latlng=40.714224,-73.961452

Intercepting and storing the encrypted data transmitted over the wire to maps.googleapis.com while executing this command:

```
curl https://maps.googleapis.com/maps/api/geocode/json?latlng=40.714224,-73.961452
```

results in a capture similar to this one:



```
19 0.180933000 10.0.2.15  156    74.125.136.9     443 TLSv1.2  215
20 0.181097000 74.125.136        10.0.2.15      36909 TCP      60
21 0.243683000 74.125.136 1413   10.0.2.15      36909 TLSv1.2  1472
22 0.244475000 74.125.136 1413   10.0.2.15      36909 TLSv1.2  1472
23 0.244500000 10.0.2.15         74.125.136.9     443 TCP      54
24 0.244997000 74.125.136 1413   10.0.2.15      36909 TLSv1.2  1472
25 0.245690000 74.125.136 1413   10.0.2.15      36909 TLSv1.2  1472
26 0.245697000 10.0.2.15         74.125.136.9     443 TCP      54
27 0.246430000 74.125.136 1413   10.0.2.15      36909 TLSv1.2  1472
28 0.247116000 74.125.136 1413   10.0.2.15      36909 TLSv1.2  1472
```

```
▶ Frame 19: 215 bytes on wire (1720 bits), 215 bytes captured (1720 bits) on interface
▶ Ethernet II, Src: CadmusCo_60:22:66 (08:00:27:60:22:66), Dst: RealtekU_12:35:02 (52:
▶ Internet Protocol Version 4, Src: 10.0.2.15 (10.0.2.15), Dst: 74.125.136.95 (74.125.
▶ Transmission Control Protocol, Src Port: 36909 (36909), Dst Port: https (443), Seq:
▼ Secure Sockets Layer
  ▼ TLSv1.2 Record Layer: Application Data Protocol: http
      Content Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 156
      Encrypted Application Data: 87194709eeec68bcf8311a9069cb369217ae31c5b3f2d768...
```

As you can see, the length of the application data is set to 156. A GCM-based ciphersuite is used, so to determine the length of the original plaintext, subtract by 24 = 132, which is exactly the size of the curl request:

```
GET /maps/api/geocode/json?latlng=40.714224,-73.961452 HTTP/1.1
User-Agent: curl/7.35.0
Host: maps.googleapis.com
```

```
Accept: */*
```

This request, minus the latitude and longitude degrees (40 and -73), can be regarded as the *constant* or *known* part of the request; from their preparation, the attacker can know which headers an application always sends as part of its request to the geocoding API, and what their length is. If it is also known that the fractional part of the coordinates is always coded using 6 digits, then the latitude and longitude degrees are the only variables.

Once fingerprinting has been used to locate a request to the Google geocoding API, the attacker can take the payload length minus the constant part to retrieve the variable part. In this example, the constant length is 129 bytes; this is the entire HTTP request sent minus the actual degrees. This leaves us with a total degree length of 5 characters (40 + -73 = 5 characters). Using the precomputed look-up table, it can be determined which part of the world is corresponds to the aggregate degree length of 5.

3. Disclaimer

Calculations might be slightly off due to the ellipsoid shape of the earth and other factors.

2. Length of IPv4 addresses leaks ranges

Similar to how you can convert the length of GPS coordinates to area's on the world map, you can also map the length of an IPv4 address to an actual set of IP ranges:

```
Length: 7 -- reduces candidate pool to 0.000233% of original
    0.0.0.0
Length: 8 -- reduces candidate pool to 0.008382% of original
    0.00.0.0
    0.0.00.0
    0.0.0.00
    00.0.0.0
Length: 9 -- reduces candidate pool to 0.127684% of original
    0.0.0.000
    0.000.0.0
    00.00.0.0
    00.0.0.00
    000.0.0.0
    00.0.00.0
    0.0.00.00
    0.00.0.00
    0.0.000.0
    0.00.00.0
Length: 10 -- reduces candidate pool to 1.071207% of original
    000.0.0.00
    000.0.00.0
    0.00.0.000
    00.00.00.0
    0.00.000.0
    00.000.0.0
    0.000.0.00
    00.00.0.00
    000.00.0.0
    00.0.000.0
    00.0.0.000
    0.0.00.000
    0.0.000.00
    0.00.00.00
    0.000.00.0
    00.0.00.00
Length: 11 -- reduces candidate pool to 5.398029% of original
    000.00.0.00
    000.0.000.0
    00.00.0.000
    000.00.00.0
    0.0.000.000
    00.00.000.0
    00.00.00.00
    0.000.0.000
    0.000.00.00
    00.000.0.00
    000.000.0.0
    0.00.00.000
    0.000.000.0
    000.0.00.00
    0.00.000.00
    00.0.000.00
    000.0.0.000
    00.0.00.000
    00.000.00.0
Length: 12 -- reduces candidate pool to 16.710833% of original
    0.00.000.000
    0.000.00.000
    00.000.000.0
    00.000.00.00
    000.00.000.0
    000.000.00.0
    000.0.00.000
    0.000.000.00
    00.00.000.00
    00.000.0.000
    00.00.00.000
    000.00.0.000
    000.000.0.00
    000.0.000.00
    000.00.00.00
    00.0.000.000
Length: 13 -- reduces candidate pool to 31.073257% of original
    00.000.00.000
    000.000.000.0
    000.000.00.00
    000.000.0.000
    00.000.000.00
    00.00.000.000
    000.00.000.00
    0.000.000.000
    000.00.00.000
    000.0.000.000
Length: 14 -- reduces candidate pool to 31.821191% of original
    000.00.000.000
    00.000.000.000
    000.000.000.00
    000.000.00.000
Length: 15 -- reduces candidate pool to 13.789183% of original
    000.000.000.000
```

If you manage to isolate an IPv4 address with string length 7 (for example 1.2.3.4) embedded in encrypted traffic, you can know that the plaintext IP is in the range 0-9.0-9.0-9.0.9. The total IPv4 space constitutes 256*256*256*256 = 4294967296 different addresses. Observer that an IP with string length 7 is sent reduces this space to 10*10*10*10 = 10000. This is only 0.000232830643654 percent of the original space.

If the attacker manages to isolate a page on a web application where an IPv4 address is displayed, and the rest of the page is static or *known* (think of a web application that displays the IPv4 address from where the last login was performed), then an estimate can be made as to which set of IP ranges it concerns. The pool may be further reduced by mapping the resultant IP ranges to Autonomous Systems[1] (AS) and discarding those Autonomous Systems that are not Internet Service Providers (in the case the attacker knows that the IP address they seek to reveal belongs to a home connection and not a company server, for instance).

5. Prevention

1. Hashing before transmission

An obvious one-size-fits-all solution is compute a hash of the password inside the user's browser using JavaScript before it is sent off to the server.

```
$ echo -n 'password' | sha256sum
5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8
$ echo -n 'apasswordthatismuchlonger' | sha256sum
36a9268776dc62211aa00e768052a628d564e3d05b48c1aa65af6c0cfa6570d4
```

Both passwords result in a hash with a length of 64 bytes.

This happens to have the additional advantage that the plaintext password is never stored anywhere except temporarily in the user's browser, as opposed to collectively in the browser, (encrypted) transit, and on your server prior to storing it encrypted in your database. The downside of this approach is that you can't evaluate the password strength on the server side. You could construct a list of a certain

1 https://en.wikipedia.org/wiki/Autonomous_system_%28Internet%29

amount of passwords that are known to be very common or too short, compute their hashes and fail to proceed once a user tries to change their password into one of these strings. However, you cannot verify whether the user has been using all of your required sets of characters (such as letters, numbers and special characters). Obviously, validation can be performed within the browser using JavaScript. Users may undermine this by tampering with the JavaScript – but this is beside the point because it is specifically the user whose safety is attempted to be strengtened by taking these measures, and their own attempts at overthrowing our security considerations are but their own responsibility.

A theoretical issue that automatically emerges is that the leak of password length is moved from the spatial to the temporal domain. That is to say, now the observer will not be able to infer the length of the password, but the length might influence the time and resources required to compute the hash. However, the detection of such microscopic details is usually confined to laboratory settings, and as long as the client-side code isn't programmed to signal the server that it is currently computing the hash (and allowing the observer to discern the exact amount of time elapsed between computation and form submission), this shouldn't really be a problem.

2. Padding the secret

An alternative to this approach is to simply pad the password right before form submission to a length that you consider to be a safe maximum-size constraint for a user password, say, a 1000 characters. How to actually implement the padding might not be straight-forward.

Trailing spaces might be part of the actual password the user has thought up. Trailing spaces might be truncated by some browsers.

Padding it with zeros (as in the ASCII character 0x00) might break some things as well.

Embedding the zero padding in JSON won't work either, because JSON will replace those with Unicode escapes such as \\u0000, which again leaks password lengths.

Adding an additional parameter to the POST submission, say, 'X', (which will be represented as 'X=......' – the X and the is-equal sign are 2 characters) and padding this with 1000 minus 2 minus password_length characters is a hack using hard-coded values and it's ugly.

What I suggest is to pad the password string with zeros (as in the ASCII character 0x00), then convert to a readable hexadecimal representation, and submit.

So if the password is 'password' and the padding length is 15 (for the sake of demonstration), then:

```
>>> pw = "password" + (chr(0x00)*7)
>>> len(pw)
15
>>> pw2 = "".join(hex(ord(c))[2:].zfill(2) for c in pw)
>>> pw2
'70617373776f726400000000000000'
>>> len(pw2)
```

3. On variable-length padding schemes

If you decide to implement some padding mechanism, beware of variable-length padding schemes. By padding a string with a different (and random) amount of characters upon each consecutive run does not make it safer but in fact introduces insecurity:

```
#!/usr/bin/env python
import random

"""
This is the server, which, in an attempt to thwart a man in the middle
from inferring the size of the secret, pads it with a variable and
random amount of bytes before each transmission.
"""
def server():
    secret = "thesecret"

    secret += " " * random.randint(0,50)

    return secret

"""
This is the man in the middle, passively taking note of all payload
sizes emitted by the server.
Once a sufficient amount of payloads with variable-length padding have
been transmitted, an accurate guess at the size of the secret can
be made.
"""
def observer():
    lengths = []
    for i in xrange(1000):
        padded_secret = server()
        lengths += [ len(padded_secret) ]
    probable_padding_length = max(lengths) - min(lengths)
    probable_secret_length = max(lengths) - probable_padding_length
    print "Length of the secret is probably: {}".format(
                                    probable_secret_length)

observer()
```

By observing a page that contains a variable-length padded secret, a sufficient amount of encrypted transmissions of this page (in this example 1000 times) allows the observer to determine the upper and the lower limit of the variable-length padding, which in this case can be expressed as the tuple (0, 50). Once these have been determined, the observer can deduce the length of the secret.

In order to subvert a padding scheme with a lower limit larger than 0 the attacker will first need to observe known values of the secret to be transmitted in order to determine the lower limit, or figure this out by scrutinizing the inner workings of the web application itself.

4. Using constant-length identifiers to refer to objects

Numeric identifiers are often used to refer to various database objects; index.php?pageid=10 loads a different page than index.php?pageId=100. The use of identifiers of constant length, such as UUID's, can prevent the linking of identifier lengths to particular (sets of) resources.