Overhead of SSL in Network Programming

Paul Kevin Short

Abstract— **This paper looks at the overhead of SSL (Secure Socket Layer) in network programming by comparing the speed of a not secure socket versus a secure socket using Python. Now that a lot of companies use the Internet for their services, network security has been an important concept at the company end and the client end. Secure Socket Layers are being used all across the web to set up connections that cannot be read by any outside source, so this paper tests the speed of a SSL server and compares it to the speed of a server that does not use SSL.**

I. INTRODUCTION

The Secure Socket Layer is a security protocol for setting up secure connections between a web browser and a server. SSL is important in network programming because ensures that the connection between the two computers will be encrypted. Without the use of a Secure Socket Layer, the packets being sent from the web browser which can contain private information can be read by others on the network who are using a packet analyzer. SSL uses what is called a certificate that holds the information about the owner of the server as well as two encryption keys, a public and private key. In the further sections I will go over how a secure connection works as well as how it performs in comparison to a socket connection without security.

II. BACKGROUND

A. Expected Results

When thinking about the runtime efficiency of SSL, it would be common to assume that it would take more time than a socket layer that doesn't use security due to the extra work that needs to be done in order to create the connection as well as transfer data. In order to understand why SSL in theory would take longer than a non-secure socket there needs to be an understanding of how the SSL handshake works.

B. SSL Handshake

The SSL handshake establishes the encrypted layer between the client and server which uses a public-private key encryption. When a connection is established the first thing that happens is the client sends a "client hello" message that contains the cryptographic information and also a random byte string that will be used for the computations. Next, the server will respond with a "server hello" message that lists the CipherSuite chosen by the server from the list provided by the client, another random byte string, and also the session ID. The server will also send the digital SSL certificate which will be processed by the client's browser for authentication. After this message the client will send the random byte string to the server which will allow the client and the server to compute the secret key that will be used to encrypt the data. There is also another message that usually gets sent by the server which asks for the client certificate to verify the authenticity of the client, this only happens in implementations that need client authentication. Once the authentication is completed the client will send the server a "finished" message which will also be encrypted by the security key. The server will then reply with a "finished" message encrypted by the security key which will tell the client that the server part of the handshake is complete. Now while the socket is still open the server and client can exchange messages that are symmetrically encrypted by the shared secret key.

III. METHOD OF EVALUATION

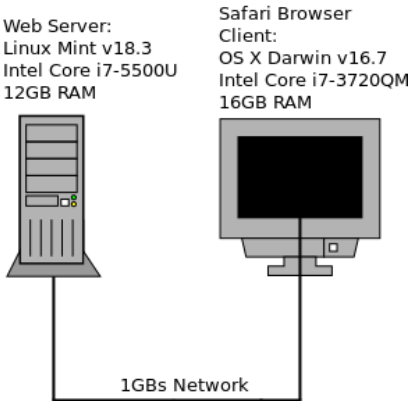
When considering the environments to test SSL two primary development environments came to mind, Java and Python. These two programming languages are used widely across web-based applications. For this experiment I used Python 2.7 for two reasons: the amount that can be done in a single line of code, and also because of my familiarity with Python. Python is a high-level interpreted language that allows rapid development and deployment of applications. Due to it's popularity, it also has wide support and extensive libraries.

A. Testing Environment

The test was run on two different systems. This removed the possibility that local file caching between the client and server may occur. This setup also attempted to simulate the network traffic that occurs in real world applications. The Python code was run on a Linux Mint v18.3 workstation. The server was accessed through Apple's Safari Browser from a OS X Darwin v16.7 workstation.

There are several methods to measure the time to serve the http and https requests, however Python's unique decorator functionality provided the easiest measurement facility. Decorator's are executed immediately before and immediately after a function call providing an excellent method to time a function call. I used the @timeit decorator that can be easily found on multiple websites.

The measurements provided in the tests will measure the actual time the server takes to respond to a GET request from a client. The difference between the SSL measure and the non-encrypted measure will measure the overhead of encrypting the SSL communication.



For the implementation of the non-secure web server the libraries that were used include, SimpleHTTServer, SocketServer, BaseTHHPServer, and BaseHttPRequestHandler. These libraries handled the implementation of the sockets whereas the web server will handle the implementation of the connection between the client which will use the socket implementations. Both the SSL web server and the non-encrypted web server were tested by first establishing the connection, then timing how long it takes to load a 2.4MB text file.[1] The page was loaded up after clearing the browsers cache twice then averaged, then the page was refreshed (cached reload) ten times then averaged. These Python libraries follow the HTTP/1.1 protocol specification. HTTP/2 is out of scope for this test.

The code for the web-server in Python is relatively straight forward:

```
class WebServer(object):
    @classmethod
    def run(cls, port=DEFAULT_PORT,
                        addr='0.0.0.0'):
        httpd = SocketServer.TCPServer((addr, port),
                        GetHandler)
        logging.debug("Running server on port %d",
                        port)
        try:
            httpd.serve_forever()
        except KeyboardInterrupt:
            httpd.server_close()
```

Only one additional line of code to wrap the socket in SSL was required to add SSL support to the web-server:

```
class SSLWebServer(object):
    @classmethod
    def run(cls, port=DEFAULT_SSL_PORT,
                        addr='0.0.0.0'):
        httpd = SocketServer.TCPServer((addr, port),
                        GetHandler)
        httpd.socket = ssl.wrap_socket(httpd.socket,
                        certfile='server.pem',
                        server_side=True)
        logging.debug("Running server on port %d",
                port)
        try:
            httpd.serve_forever()
        except KeyboardInterrupt:
            httpd.server_close()
```

[1]The file contains the text of "The Adventures of Sherlock Holmes" by Sir Arthur Conan Doyle.

The code for the GET method was identical for both implementations:

```
class GetHandler(BaseHTTPRequestHandler):
    @timeit
    def do_GET(self):
        logging.debug("Executing do_GET")
        self.send_response(200)
        self.send_header('Last-Modified',
                self.date_time_string(time.time()))
        self.end_headers()
        f = open("book.txt")
        self.wfile.write(f.read())
        f.close()
        return
```

IV. RESULTS

Figure 1 displays the consolidated results from the tests. The initial request time is broken out from the ongoing refresh rates after the initial load. Breaking out these two types of requests are necessary since a client will load a page from cache if there is no change in the content.

A. *Initial Request*

Client GET –¿ Server
Client ¡– 200 plus full page

B. *Page Refresh*

Client GET –¿ Server
Client ¡– 304 (Page not changed, use prior cached version)

The initial page load was almost 2X slower when using SSL. Additional page loads from cache were significantly faster but still much slower under SSL. The refresh rates were almost 8 times slower under SSL.

It's important to note that there may be a significant increase of time between the encrypted SSL requests vs the non-encrypted results, yet the times are short enough that the end-user interacting with the client may not even notice the difference between the two load times.

TABLE I
TIMING RESULTS

	Secure Socket	Non-Secure Socket
Initial Request	239.21 ms	123.65 ms
Page Reload	136.229 ms	16.701 ms

V. DISCUSSION/CONCLUSION

A. *Expected Results*

There appears to be measurable overhead between encrypted SSL web traffic and non-encrypted web traffic. Modern processors have the computing power to encrypt streaming data without significant overhead. However, the

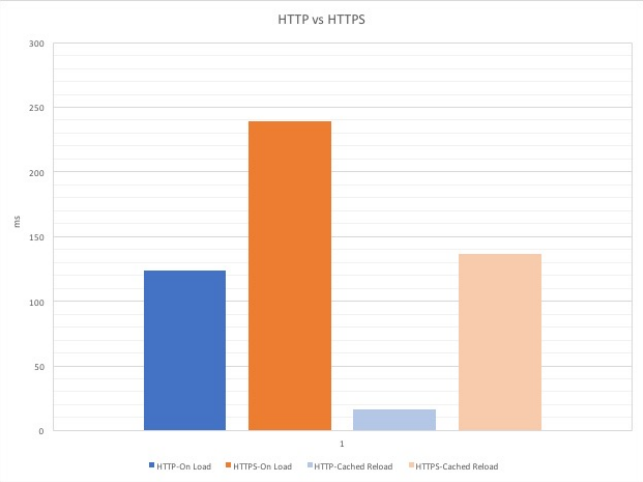


Fig. 1. Empirical results on the HTTP vs HTTPS web servers in milliseconds

results from this straight forward experiment points to the possibility of a high amount of overhead. Expanding the results from this simple test are significant when applied to large farms of web-servers. A 2X overhead could be the difference between 1000 and 2000 servers in a large web application, not an insignificant expense!

A search of websites provides ample information regarding SSL and non-encrypted traffic. One site I found interesting was http://www.httpvshttps.com and https://www.httpvshttps.com. This site allows you to interact with a page via HTTPS or HTTP. In most cases, I found HTTPS to be faster than the HTTP. The site states that is uses a non-caching, nginx server with a non-proxied connection. This test also compares HTTP/1.1 vs HTTPS/2 so the comparison is not as strict as my test was using HTTP/1.1 for both tests.

B. Discussion

Upon further investigation there are several factors taking place that are not fully isolated within this test.

1) This test measures the overall time from the time the client connects to when the page is fully served to the client over the network. The SSL protocol handshake had additional overhead when compared to a non-encrypted handshake.

2) This test is using an interpreted language rather than a compiled language. To ensure that there was no differences in the coding between SSL and non-SSL traffic, I used a generic HTTP server object then wrapped this object in SSL. Interpreted languages provide easier implementation but not necessarily faster speeds. In addition, since the language is interpreted line by line, there is the possibility that the underlying execution may not be as efficient as possible. An example of this in Python is the speed to run a sort with a single operation vs with Python code that appears to be highly optimized. The single sort command may run underlying optimized code written in C/C++ to sort the data. When written out step by step, the interpreter must load and execute each statement in order.

3) Upon further review, our test environment does not reflect the complexity of real world applications. Modern applications are tiered into multiple layers or services. The client will only interact with a small portion of a web-site. Many web-sites implement caching mechanisms and content distribution networks to ensure high and rapid availability of data.

4) Clients are even more complex then they were in the past. During my testing I did notice that the server would respond BEFORE I completed typing the full URL. Browsers such as Safari and Chrome are optimized to predict behavior and start loading data prior to the end-user request.

C. Conclusion

There is overhead when running SSL. The handshake between the client and server required additional steps and the CPU does have to spend time encrypting data. However, the differences between the two are becoming insignificant with modern hardware and web architectures. Speed is no longer a reason to use non-encrypted traffic over the Internet. The shear risk of transmitting data over the Internet in an non-encrypted fashion cannot justify speed savings. During these tests, it was obvious that I was unable to see the difference in speed between SSL and non SSL connections even if there are measurable differences.

FUTURE WORK

The next steps in this research is to strip out any additional components to eliminate some of the variables I encountered.

1) Project B would be writing a socket server in C - one with SSL, and one without. The server could simply accept a connection, and send a present amount of data, then close the connection. This test will eliminate any HTTP overhead I may have encountered during testing. This test can then be expanded to utilize HTTP/1.1 and HTTP/2.

2) Modern applications are written with larger frameworks. Project C would be to create a website using competing frameworks using both HTTP and HTTPS. Creating the same website using Node.js and Django then measuring the SSL overhead in both HTTP/1.1 and HTTP/2 will provide insight to how each framework behaves with encrypted and non-encrypted traffic.

References are important to the reader; therefore, each citation must be complete and correct. If at all possible, references should be commonly available publications.

REFERENCES

[1] What Is SSL? — Definition and How SSL Works — Comodo SSL Wiki. Instant SSL, 12 Sept. 2017, www.instantssl.com/ssl.html.
[2] HTTP vs HTTPS Test. HTTP vs HTTPS - Test Them Both Yourself, www.httpvshttps.com/.
[3] What Is SSL (Secure Sockets Layer)? DigiCert, www.digicert.com/ssl/.
[4] 17.3. Ssl - TLS/SSL Wrapper for Socket Objects. 17.3. Ssl - TLS/SSL Wrapper for Socket Objects - Python 2.7.14 Documentation, docs.python.org/2/library/ssl.html.