TLS 1.3 Performance and Security

TBC
TBC@live.napier.ac.uk
Edinburgh Napier University - Module Title (CSN11131, Applied Cryptography)

Abstract

Transport Layer Security (TLS) is a widely adopted security protocol that is used to ensure security and privacy for communications over the internet and other public and private networks. It was first introduced in 1999 by the Internet Engineering Task Force (IETF) [Dierks and Allen, 1999]. This paper looks into the security of the most recent version of TLS, TLS 1.3 and looks at the security and performance of TLS 1.3 versus its predecessor TLS 1.2 when used in the HTTPS protocol.

1 Introduction

TLS provides strong Confidentiality, Integrity and Authentication to network connections. As documented in the TLS 1.3 standard [Rescorla, 2018] "The primary goal of TLS is to provide a secure channel between two communicating peers; the only requirement from the underlying transport is a reliable, in-order data stream."

- Confidentiality - The content of messages can be encrypted using a selection of algorithms.
- Integrity - A hash-based message authentication code (HMAC) is used to ensure messages aren't changed during transit.
- Authentication - The identity of the client, server or both parties can be verified.

Without TLS we may not be able to safely perform many of the online activities that we take for granted including online shopping and online banking. The convenience, ease of use and the ubiquity of the protocol within all modern web browsers has no doubt contributed to the explosion of the use of the internet as we know it today.

TLS is used to secure HTTP connections over the Internet as part of Hypertext Transfer Protocol Secure (HTTPS). This protocol was first proposed in 2000 by The Internet Society [Rescorla, 2000] and forms the RFC2818 standard. HTTPS is supported by all modern web browser software including Microsoft Internet Explorer, Firefox and Google Chrome. It is this integration that means TLS has become the de-facto method for securing connections over the internet.

Alexa, an Amazon company, compiles a list of the top one million websites visited by traffic volume. A security researcher, Scott Helme, has done some analysis on this list [hel,] to determine what percentage of the sites are using key security features. Helme determined that of February 2019, 552,429 of the top 1 million websites were using HTTPS.

As can be seen in Figure 1 the percentage of sites using HTTPS has been rapidly increasing since 2016. Use of TLS in the top 1 million websites has increased from 6.71% in August 2015 to 58.44% in February 2019.

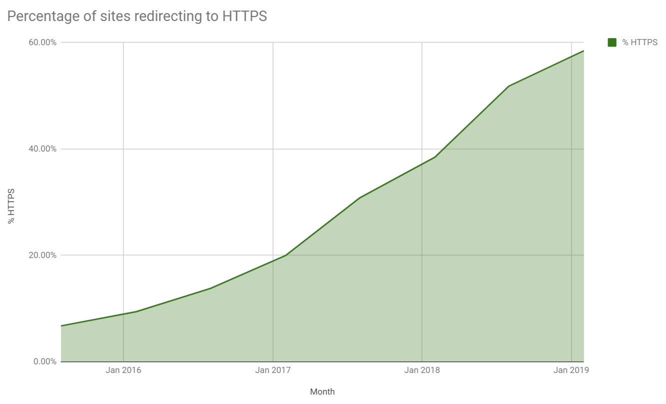


Figure 1: A graph of the percentage of sites redirecting to HTTPS over time (from [hel,])

TLS 1.3 has been designed to have higher performance and to be more secure than TLS 1.2. These performance and security enhancements are explored in more detail in the following sections.

1.1 Performance

TLS performance has been enhanced by reducing the number of round trips required to complete the TLS handshake process and receive a response to the initial request. Latency in network connections over the internet is often one of the the major factors in TLS performance. In TLS 1.2 it takes three round trips for the TLS handshake to be completed and an HTTP response to be returned to the client. In TLS 1.3 this has been reduced to two round trips which should result in a significant performance improvement.

TLS uses both symmetric and asymmetric encryption to establish a secure tunnel. The process of establishing this secure tunnel involves multiple steps. The client and the server must negotiate the algorithm used and exchange key information. Figure 2 shows a graphical representation of the TLS 1.2 handshake flow.

1. The first step is the **Client Hello** message. The client initiates the connection to the server and specifies the TLS versions that may be used, the list of

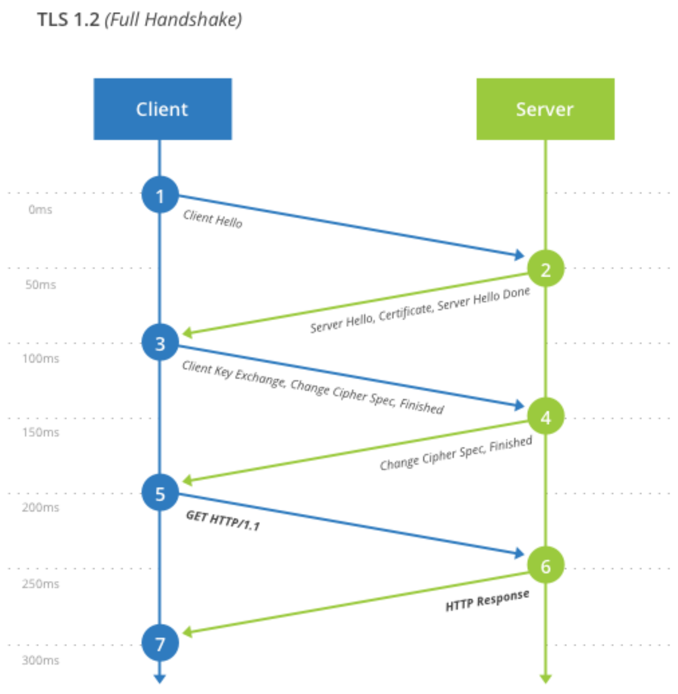

TLS 1.2 (Full Handshake)
Client
Server
0ms
50ms
100ms
150ms
200ms
250ms
300ms
1
Client Hello
2
Server Hello, Certificate, Server Hello Done
3
Client Key Exchange, Change Cipher Spec, Finished
4
Change Cipher Spec, Finished
5
GET HTTP/1.1
6
HTTP Response
7

Figure 2: The TLS 1.2 handshake process (from [Wha,])

cipher suites supported and a 32-byte random number used later to generate the encryption key to be used.

Cipher suites are identified by strings which identify the algorithms used. A sample cipher suite is:

TLS-ECDHE-ECDSA-CHACHA20-POLY1305-SHA256

This contains the following information:

- *TLS* is the protocol
- *ECDHE* is the key exchange algorithm (Elliptic Curve Diffie-Hellman Ephemeral)
- *ECDSA* is the authentication algorithm (Elliptic Curve Digital Signature Algorithm)
- *CHACHA20-POLY1305* is the data encryption algorithm (ChaCha stream cipher and Poly1305 authenticator)
- *SHA256* is the Message Authentication Code (MAC) algorithm (Secure Hash Algorithm 256 bits)

2. The server responds with a **Server Hello** message. The server selects the preferred TLS version and cipher from the option presented by the client. The server includes a a 32-byte random number used later to generate the encryption key to be used.

   The server also sends the client its certificate which the client can use to authenticate the server.

   The server may also send a key exchange that is used for anonymous Diffie-Hellman, Diffie-Hellman Ephemeral and Ephemeral RSA Key-Exchange methods. [Lda,].

   Finally the server will send a **Server Hello Done** message to signify to the client that the Server Hello message is finished.

3. The **Client Key Exchange** message is sent containing a pre-master secret which is encrypted using the server's public key which can be extracted from the certificate. Both the client and server can now compute the master encryption key using the random secrets previously exchanged. The client sends a Change Cipher Spec message to indicate that all future messages should be encrypted. It then sends a Finished message to the server indicating it is done transmitting.

4. The Server replies with it's own **Change Cipher Spec** message and follows it up with a **Finished** message.

5. The client can now send HTTP traffic over a secure TLS 1.2 tunnel to the server. All messages exchanged from this point on will be encrypted.

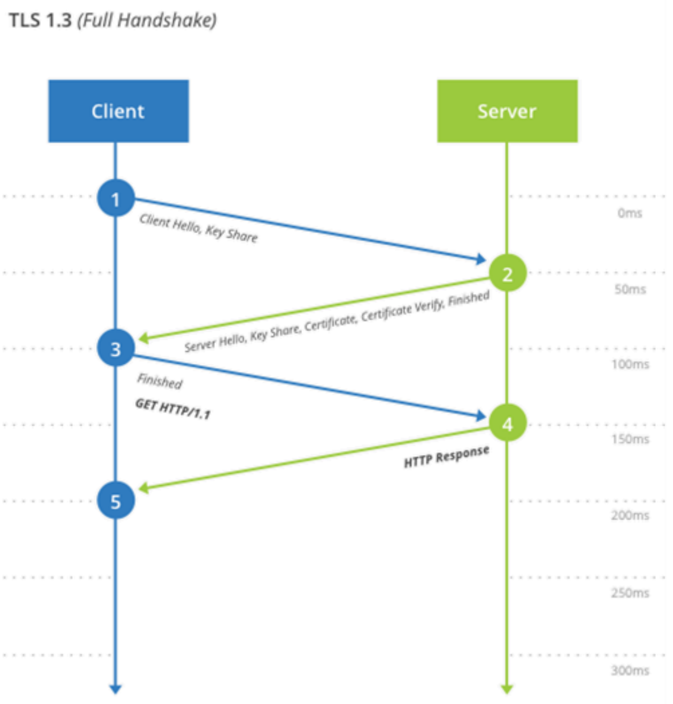


Figure 3: The TLS 1.3 handshake process (from [Wha,])

The TLS 1.3 handshake process involves fewer round trips between client and server than the TLS 1.2 handshake. By removing a round trip the performance of the protocol is improved. Figure 3 shows a graphic representation of the TLS 1.3 handshake flow.

1. The first step is the **Client Hello** message. The client initiates the connection to the server specifying the TLS versions that may be used and the list of cipher suites supported. It also guesses which cipher the server will select and sends a public key.
2. The server responds with a **Server Hello** message which contains the selected cipher, its public key and its certificate. This message is encrypted using the client's public key.
3. Both client and server have exchanged public keys and can calculate a shared secret from the messages exchanged. The clients can now exchange messages over an encrypted TLS 1.3 tunnel.

The TLS 1.3 handshake completes in one round-trip time (1-RTT). The protocol also introduces a feature known as zero round-trip time (0-RTT) which can be used when resuming a pre-negotiated TLS session. 0-RTT allows a client to send data with the initial connection request therefore improving performance by removing the latency of a single round trip to the server. The 0-RTT handshake flow is shown in Figure 4.

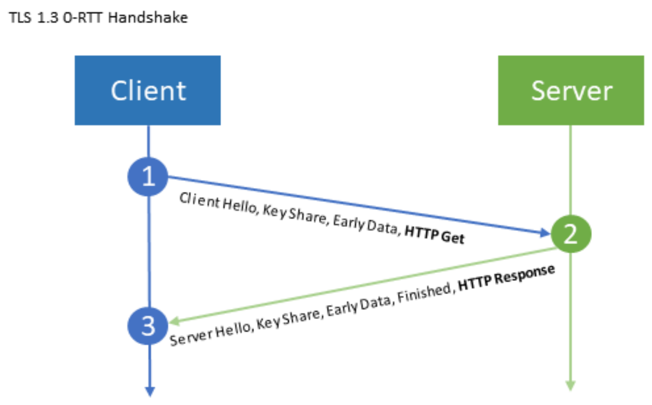

TLS 1.3 0-RTT Handshake
Client
Server
1
Client Hello, Key Share, Early Data, HTTP Get
2
Server Hello, Key Share, Early Data, Finished, HTTP Response
3

Figure 4: The TLS 1.3 0-RTT handshake process

1.2 Security

A number of the security enhancements provided by TLS 1.3 have been done by trimming the number of cipher suites that are supported.

- RSA and Diffie-Helman key exchange have been removed as they do not provide forward secrecy.
- CBC mode ciphers have been removed as they enabled attacks such as BEAST and Lucky 13.
- RC4 stream cipher is no longer supported due to weaknesses in the algorithm.
- SHA-1 hash function has been deprecated in favor of SHA-2 which is stronger.
- MD5, SHA-224 and DSA have all been deprecated due to weaknesses.
- Arbitrary Diffie-Hellman groups used in the Logjam attack are not supported[Adrian et al., 2015].
- Export ciphers used in the FREAK[Bhargavan et al.,] and Logjam[Adrian et al., 2015] attacks are no longer supported.

Forward secrecy ensures that should the servers private key be compromised then not all sessions using this key are compromised. It achieves this by generating a unique session key for each session initiated.

Furthermore compression of data is no longer supported reducing the likelihood of CRIME [Rizzo and Duong, 2012] style attacks. In addition, Renegotiation is not possible in a TLSv1.3 connection due to issues such as the protocol design issue highlighted in the Transport Layer Security (TLS) Renegotiation Indication Extension paper [Rescorla et al., 2010].

Only the initial Client Hello request is sent in cleartext in a TLS 1.3 connection. The server encrypts its response to the request with the public key the client specifies in the request. This means that the negotiation of keys and ciphers is kept secret.

One of the downsides of the increased security in TLS 1.3 is that it can break many security tools which rely on logging network traffic and decrypting this traffic. As TLS 1.3 has forward secrecy and a new public key is created for each session, it is not possible to use a single private key to decrypt the traffic.

Tooling which relies on passive offloading of network traffic cannot decrypt TLS 1.3 traffic as there isn't a single private key. However, it is still possible to create a man-in-the-middle setup in a corporate infrastructure using a proxy server. This will allow decryption of TLS 1.3 traffic for inspection before the traffic is forwarded on using a new TLS connection.

Another downside of TLS 1.3 is that the 0-RTT feature introduces new risks as the 0-RTT data is vulnerable to replay attacks and is not forward secret.

2 Literature Review

The TLS 1.3 protocol was published in 2018 [Rescorla, 2018], ten years after the TLS 1.2 protocol specification was published in 2008 [Dierks and Rescorla, 2008]. Both specifications are published Internet Standards. As articulated in [Bradner, 1996], "an Internet Standard is a specification that is stable and well-understood, is technically competent, has multiple, independent, and interoperable implementations with substantial operational experience, enjoys significant public support, and is recognizably useful in some or all parts of the internet."

The TLS 1.3 standard [Rescorla, 2018] details the differences between TLS 1.2 and TLS 1.3. Major differences include a removal of insecure ciphers, the introduction of a zero round-trip time (0-RTT) and the requirement for all key exchange mechanisms to provide forward secrecy. The risks introduced by the protocol are also detailed and several notable risks exist in the 0-RTT specification. In particular 0-RTT data is not forward secret and is also vulnerable to being replayed by an attacker.

Several flaws have also been documented in the TLS 1.2 protocol over the years. A number of these are due to the support for weaker key exchange algorithms such as RSA.

In the Return Of Bleichenbacher's Oracle Threat (ROBOT) [Böck et al., 2018] paper, Bock, Somorovsky & Youngs detail how they were able to take advantage of weaknesses in the RSA key exchange algorithm. The specific attack type is known as a Padding Oracle attack and was first documented by Bleichenbacher in the 1998 paper [Bleichenbacher, 1998]: Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1.

The Weak Diffie-Hellman and the Logjam Attack paper[Adrian et al., 2015] articulates how a flaw in the TLS implementation of Diffie-Helman can be used by an attacker to downgrade a TLS connection to use a weaker cipher than originally negotiated.

In The CRIME attack paper [Rizzo and Duong, 2012] , Duong and Rizzo detail how they were able to take advantages of flaws in compression algorithms in TLS in order to hijack HTTPS sessions.

3 Implementation

The implementation was completed using a distributed architecture with a server element designed on the Amazon Web Services (AWS) public cloud and a client element deployed on the Ubuntu Linux operating system.

According to the official NIST definition [Mell et al., 2011], "cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."

AWS is as of 2019 the largest of all public cloud providers with a 2018 revenue of 25.65 billion US Dollars [Top, 2019]. AWS provide over one hundred services including compute, database, storage, analytics and machine learning. The AWS service used for this implementation is Amazon Elastic Compute Cloud (Amazon EC2). EC2 is a web service that provides secure, re-sizable compute capacity in the cloud. It is designed to make web-scale cloud computing easier for developers. Nginx a free open-source, high performance web server was installed on the Ubuntu operating system on an EC2 virtual server. Nginx was chosen due to its support for TLS 1.3.

Ubuntu is a free and open-source Linux distribution based on the Debian distribution. It was chosen due to OpenSSL 1.1.1 being installed which has TLS 1.3 support.

3.1 AWS Implementation

A virtual server was configured on Amazon EC2 using the t2.micro instance type which provides 1 virtual CPU

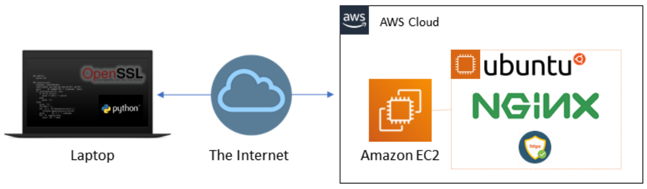
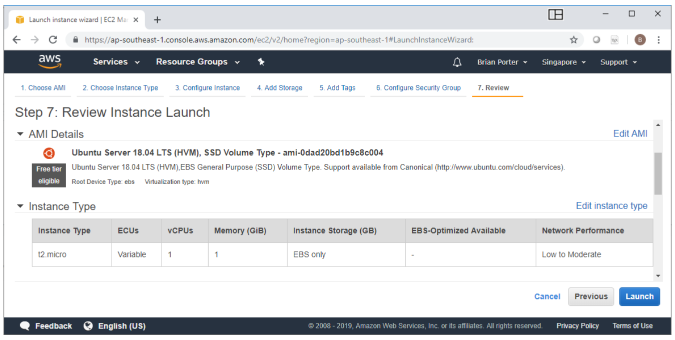


Figure 5: A simplified system architecture diagram

equivalent to a 2.5 GHz, Intel Xeon Family and 1 GB memory.


Launch instance wizard | EC2 Ma
https://ap-southeast-1.console.aws.amazon.com/ec2/v2/home?region=ap-southeast-1#LaunchInstanceWizard:
aws
Services
Resource Groups
Brian Porter
Singapore
Support
1. Choose AMI
2. Choose Instance Type
3. Configure Instance
4. Add Storage
5. Add Tags
6. Configure Security Group
7. Review
Edit AMI
Edit instance type
Cancel
Previous
Launch
Feedback
English (US)
© 2008 - 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Step 7: Review Instance Launch

AMI Details

Ubuntu Server 18.04 LTS (HVM), SSD Volume Type - ami-0dad20bd1b9c8c004

Free tier eligible

Ubuntu Server 18.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (http://www.ubuntu.com/cloud/services).

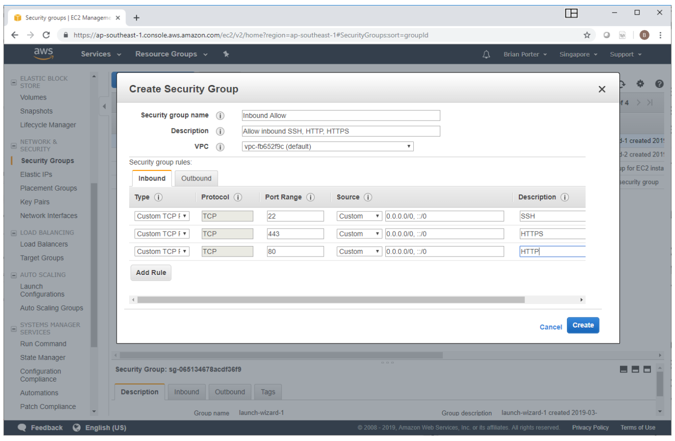
Root Device Type: ebs Virtualization type: hvm

Instance Type

Instance Type	ECUs	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
t2.micro	Variable	1	1	EBS only	-	Low to Moderate

Figure 6: EC2 instance details

AWS security groups were configured which act as a firewall to limit inbound and outbound traffic to/from EC2 instances. The SSH, HTTP and HTTPS protocols were configured to be allowed from any source address to enable this implementation.



Create Security Group

Security group name: Inbound Allow

Description: Allow inbound SSH, HTTP, HTTPS

VPC: vpc-fb652f9c (default)

Security group rules:

Type	Protocol	Port Range	Source		Description
Custom TCP F	TCP	22	Custom	0.0.0.0/0, ::/0	SSH
Custom TCP F	TCP	443	Custom	0.0.0.0/0, ::/0	HTTPS
Custom TCP F	TCP	80	Custom	0.0.0.0/0, ::/0	HTTP

Security Group: sg-065134678acdf36f9

Group name launch-wizard-1 Group description launch-wizard-1 created 2019-03-

Figure 7: EC2 security group configuration

A public private key pair is automatically created by EC2 and the private key is downloaded in the binary .pem format which is a base64 encoded certificate. This private key is used to connect remotely to the EC2 instance in place of a password.

The server was launched and could be connected to using SSH using the following command:

```
ssh −i "NapierTLS.pem" ubuntu@ec2−13−250−36−136.ap−↩
    southeast−1.compute.amazonaws.com
```

Ubuntu version 18.04 is pre-installed on the server and the public key is placed in the ~/.ssh/authorized_keys directory to allow remote management of the instance via SSH.

The Nginx package included in the Ubuntu repository is not compiled with OpenSSL version 1.1.1 which is required for TLS 1.3 support. To ensure TLS 1.3 support, a version of Nginx was installed from a third party repository using the following commands:

```
sudo add−apt−repository ppa:ondrej/nginx
sudo apt update
sudo apt install nginx
```

The version of Nginx installed was confirmed by running the following command:

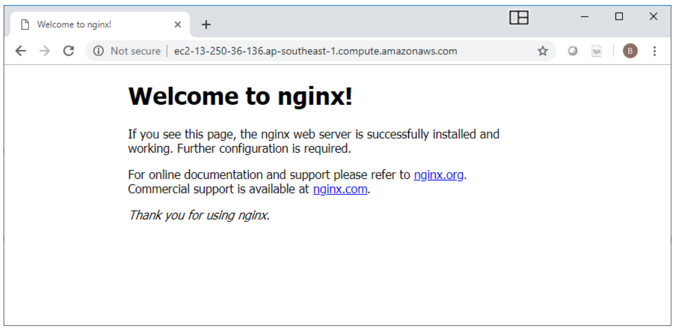
```
sudo nginx −V
```

This returned the following data showing that the correct version of Nginx has been installed and that it has been built with OpenSSL 1.1.1

```
nginx version: nginx/1.14.2
built with OpenSSL 1.1.1  11 Sep 2018 (running with OpenSSL↩
        1.1.1b  26 Feb 2019)
TLS SNI support enabled
```

Nginx was then started using the following command and a test connection made to the HTTP service as shown in Figure 8

```
sudo systemctl start nginx
```


Welcome to nginx!
Not secure | ec2-13-250-36-136.ap-southeast-1.compute.amazonaws.com

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Figure 8: Test connection to Nginx over HTTP

Once Nginx was confirmed to be functioning as expected for HTTP, the configuration for HTTPS was completed. Three things needed to be done in order for HTTPS to work:

- x509 certificates created.
- Diffie Hellman Ephemeral parameters created.
- Nginx configuration file updated.

A self-signed x509 certificate was created using the following command:

```
sudo openssl req −x509 −nodes −days 365 −newkey rsa:2048 −↩
      keyout /etc/nginx/nginx−selfsigned.key −out /etc/nginx/↩
      nginx−selfsigned.crt
```

Next a new Diffie-Helman Ephemeral parameter was created using the following command. This isn't strictly necessary for TLS to function as the OpenSSL defaults will be used, however, as the default groups are 1024 bits they could be broken by a nation state as articulated in the Imperfect Forward Secrecy [Adrian et al., 2015] paper which describes the Logjam attack on Diffie-Helman key exchange.

```
sudo openssl dhparam −out /etc/ssl/certs/dhparam.pem 2048
```

The Nginx configuration file /etc/nginx/nginx.conf was updated. Two version of this were created, one with TLS1.2 enabled and one with TLS 1.3 enabled.

A server section was added to the file to instruct the server to listen on port 443 for TLS.

```
    server {
        listen 443 ssl;
        listen [::]:443 ssl;
        server_name default;
        root /var/www/html;
        index index.html index.htm index.nginx−debian.html;
    }
```

Finally TLS related parameters were set. The ssl_protocols parameter is used to instruct the server what versions of SSL and TLS to use. This is the only parameter that was changed between configurations.

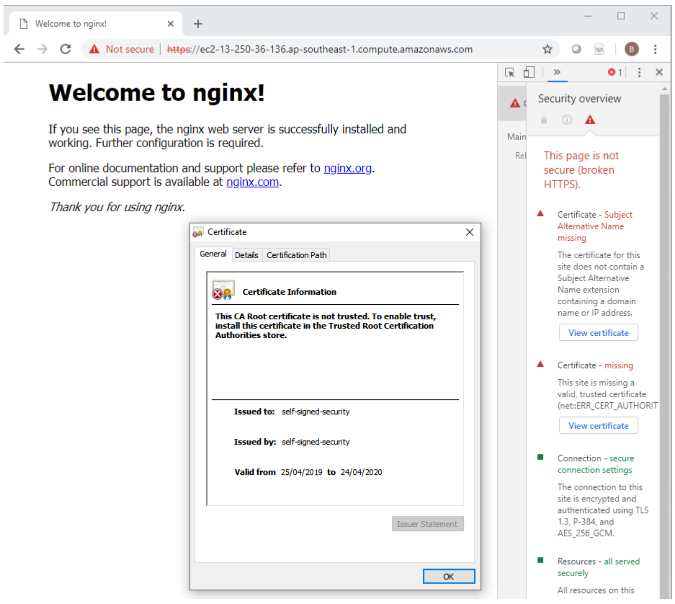
```
    ssl_protocols TLSv1.3;
    ssl_prefer_server_ciphers on;
    ssl_ecdh_curve secp384r1;
    ssl_session_timeout  10m;
    ssl_session_cache shared:SSL:10m;
    ssl_session_tickets off;
    ssl_stapling on;
    ssl_stapling_verify on;
    resolver 8.8.8.8 8.8.4.4 valid=300s;
    resolver_timeout 5s;
    ssl_certificate /etc/nginx/nginx−selfsigned.crt;
    ssl_certificate_key /etc/nginx/nginx−selfsigned.key;
    ssl_dhparam /etc/ssl/certs/dhparam.pem;
```

A connection using HTTPS was then attempted. This resulted in a security warning from the web browser about the use of the self-signed certificate. Once this warning was acknowledged, the site was proven to work correctly and was using a TLS 1.3 connection as shown in Figure 9.

A bash script was written to generate one thousand random files of size 2KB. The script pipes data from the /dev/urandom file which generates a psuedorandom bytestream. This was done to ensure that caching wouldn't take place on the client or server, ensuring each TLS connection made is unique and is completed in full when running the evaluation.

Listing 1: Bash script to create 1000 random files

```
#!/bin/bash
for i in {1..1000}
do
  head −c 2k </dev/urandom > file$i
done
```



Welcome to nginx!

Not secure | https://ec2-13-250-36-136.ap-southeast-1.compute.amazonaws.com

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

Certificate

General | Details | Certification Path

Certificate Information

This CA Root certificate is not trusted. To enable trust, install this certificate in the Trusted Root Certification Authorities store.

Issued to: self-signed-security

Issued by: self-signed-security

Valid from 25/04/2019 **to** 24/04/2020

Issuer Statement

OK

Security overview

This page is not secure (broken HTTPS).

Certificate - Subject Alternative Name missing

The certificate for this site does not contain a Subject Alternative Name extension containing a domain name or IP address.

View certificate

Certificate - missing

This site is missing a valid, trusted certificate (net::ERR_CERT_AUTHORIT

View certificate

Connection - secure connection settings

The connection to this site is encrypted and authenticated using TLS 1.3, P-384, and AES_256_GCM.

Resources - all served securely

All resources on this

Figure 9: A successful TLS 1.3 connection

3.2 Client Implementation

A python script was written to make requests to the server whilst measuring the execution time of each request. The Python Requests library was used to perform HTTPS requests. This library was chosen due to the simplicity of generating HTTPS Requests. The library uses OpenSSL as the underlying method of making TLS connections.

The Ubuntu 19.04 operating system was chosen due to it having OpenSSL 1.1.1 pre-installed which is the version required to support TLS 1.3. This was installed and run from a virtual machine on my laptop.

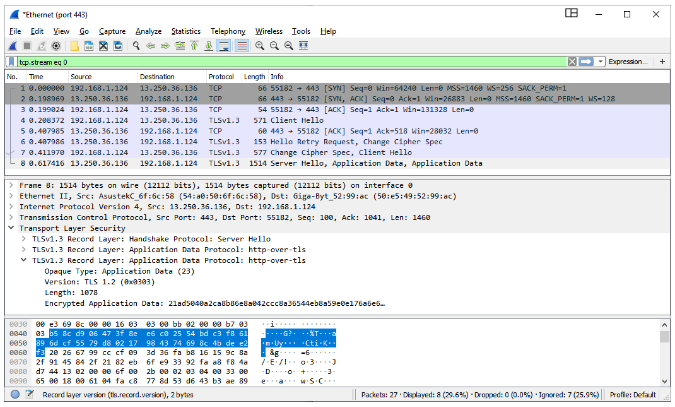
The python requests library was installed using the following command:

```
pip install requests
```

Listing 2: Python script to make 1000 TLS requests and measure the execution time of each request

```python
import requests
import time

iterations = 1000
average = 0

for x in range(1, iterations):
    start = time.time()
    requests.get('https://ec2-13-250-36-136.ap-southeast-1.compute.amazonaws.com/file' + str(x), verify=False)
    end = time.time()
    print(end - start)
    average = average + (end - start)

average = average / iterations

print('average: ' + str(average))
```

A single HTTPS request was executed from the python command line and the traffic viewed in Wireshark to confirm that the python requests library was indeed using TLS 1.3. The result of this test can be viewed in the following figure.



```
*Ethernet (port 443)
File  Edit  View  Go  Capture  Analyze  Statistics  Telephony  Wireless  Tools  Help
tcp.stream eq 0                                                        Expression...

No. Time     Source        Destination   Protocol Length Info
  1 0.000000 192.168.1.124 13.250.36.136 TCP        66 55182 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
  2 0.198969 13.250.36.136 192.168.1.124 TCP        66 443 → 55182 [SYN, ACK] Seq=0 Ack=1 Win=26883 Len=0 MSS=1460 SACK_PERM=1 WS=128
  3 0.199024 192.168.1.124 13.250.36.136 TCP        54 55182 → 443 [ACK] Seq=1 Ack=1 Win=131328 Len=0
  4 0.208372 192.168.1.124 13.250.36.136 TLSv1.3   571 Client Hello
  5 0.407985 13.250.36.136 192.168.1.124 TCP        60 443 → 55182 [ACK] Seq=1 Ack=518 Win=28032 Len=0
  6 0.407986 13.250.36.136 192.168.1.124 TLSv1.3   153 Hello Retry Request, Change Cipher Spec
  7 0.411970 192.168.1.124 13.250.36.136 TLSv1.3   577 Change Cipher Spec, Client Hello
  8 0.617416 13.250.36.136 192.168.1.124 TLSv1.3  1514 Server Hello, Application Data, Application Data

> Frame 8: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
> Ethernet II, Src: AsustekC_6f:6c:58 (54:a0:50:6f:6c:58), Dst: Giga-Byt_52:99:ac (50:e5:49:52:99:ac)
> Internet Protocol Version 4, Src: 13.250.36.136, Dst: 192.168.1.124
> Transmission Control Protocol, Src Port: 443, Dst Port: 55182, Seq: 100, Ack: 1041, Len: 1460
v Transport Layer Security
  > TLSv1.3 Record Layer: Handshake Protocol: Server Hello
  > TLSv1.3 Record Layer: Application Data Protocol: http-over-tls
  v TLSv1.3 Record Layer: Application Data Protocol: http-over-tls
      Opaque Type: Application Data (23)
      Version: TLS 1.2 (0x0303)
      Length: 1078
      Encrypted Application Data: 21ad5040a2ca8b86e8a042ccc8a36544eb8a59e0e176a6e6…

0030  00 e3 69 8c 00 00 16 03  03 00 bb 02 00 00 b7 03   ··i····· ········
0040  03 b5 8c d9 06 47 3f 8e  e6 c0 25 54 bd c3 f8 61   ·····G?· ··%T···a
0050  89 6d cf 55 79 d8 02 17  98 43 74 69 8c 4b de e2   ·m·Uy··· ·Cti·K··
0060  f3 20 26 67 99 cc cf 09  3d 36 fa b8 16 15 9c 8a   · &g···· =6······
0070  2f 91 45 84 2f 21 82 eb  6f e9 33 92 fa a8 f8 4a   /·E·/!·· o·3····J
0080  d7 44 13 02 00 00 6f 00  2b 00 02 03 04 00 33 00   ·D····o· +·····3·
0090  65 00 18 00 61 04 fa c8  77 8d 53 d6 43 b3 ae 89   e···a··· w·S·C···

Record layer version (tls.record.version), 2 bytes || Packets: 27 · Displayed: 8 (29.6%) · Dropped: 0 (0.0%) · Ignored: 7 (25.9%) || Profile: Default
```

Figure 10: A successful TLS 1.3 connection

4 Evaluation

To evaluate the performance of the TLS 1.2 protocol against the performance of the TLS 1.3 protocol, the python script on the client was run three times for TLS 1.3 and another three times for TLS 1.2. The script made 1000 TLS connections each time it was run and measured the average time of each connection. Each of these 1000 connections downloaded a unique 2 kilobyte file to mitigate the effect of caching on the client or server side. The same setup was used for both TLS 1.2 and TLS 1.3 meaning that the effects of system performance and network performance should be equivalent. The Nginx server side component was restarted each time the configuration changed from TLS 1.2 to TLS 1.3 and vice versa.

The results of this evaluation can be found in table 1.

The performance of TLS 1.3 was not found to be measurably better than TLS 1.2 in real-world conditions. However, the protocol does offer additional security when compared to TLS 1.2 and this has been proven not to come at the cost of performance.

One reason for TLS 1.3 not clearly outperforming TLS 1.2 could be due to the lack of TLS early data in the implementation. TLS Early data allows data to be sent in the first round trip of a TLS connection between client and server as described in [Usi,].

Furthermore, there were several other factors that could have influenced the results of this evaluation. The Nginx server was using a self-signed certificate which generated errors in the python requests library and these errors had to be suppressed.

The python requests library itself may have been a factor as it abstracts away the complexity of HTTPS requests and there is little opportunity to influence the raw TLS connection being made. It is possible that if using a library that offers more control of the TLS connection, then performance could have been optimized.

The AWS EC2 server was mistakenly created in the Singapore region which adds latency when connecting from the UK. With TLS 1.2 taking an extra round trip, it was expected that this would be a contributing factor to per-

TLS 1.2	TLS 1.3
816.13 ms	813.25 ms
824.83 ms	821.25 ms
805.42 ms	840.90 ms

Table 1: Average time in milliseconds of TLS connections

formance. A server closer to the UK would result in higher performance due to the reduced latency.

The Nginx server may not have been optimally configured. For example, a list of preferred ciphers was not configured which may have contributed to performance inefficiencies in the protocol setup. The server and client may not have been able to agree on suitable ciphers at the first attempt, leading to the latency of additional requests.

Even though steps were taken to mitigate the effects of caching, it's possible that caching was being applied somewhere in the system with the results being skewed by this.

5 Conclusion

On paper, TLS 1.3 offers significant security benefits over it's predecessor, TLS 1.2. Lessons have been learned from previous TLS implementations and TLS 1.3 offers significantly enhanced security when compared with TLS 1.2. Weaker cryptography is no longer supported and stronger modern cryptography methods are supported. Forward secrecy ensures that the impact of a breach of a private key is minimised.

While the performance enhancements of TLS 1.3 appear clear on paper, the real-world implementation tested in this paper found that the performance of TLS 1.3 over TLS 1.2 when used in the HTTPS protocol is at best minor. The same tests were repeated three time and TLS 1.3 was on average 3 milliseconds faster then TLS 1.2 during two of the tests. In the final test it was found that TLS 1.2 outperformed TLS 1.3 by 35ms on average. This inconsistency may be due to confounding factors such as network bandwidth and CPU utilization as the testing was performed in real world conditions. In a more controlled test the results may have been different.

Even though testing of the performance enhancements were found to be inconclusive, TLS 1.3 does provide significantly enhanced security for little or no performance overhead. As such, this evaluation finds no reason not to use TLS 1.3 given the security enhancements over previous versions.

References

[hel,] Alexa top 1 million analysis - february 2019. https://scotthelme.co.uk/alexa-top-1-million-analysis-february-2019/.

[Lda,] Ldapwiki: Serverkeyexchange. https://ldapwiki.com/wiki/ServerKeyExchange.

[Usi,] Using early data in http. https://tools.ietf.org/id/draft-thomson-http-replay-01.html.

[Wha,] What's new in tls 1.3? — cloudflare. https://www.cloudflare.com/learning-resources/tls-1-3/.

[Top, 2019] (2019). Top cloud providers 2019: Aws, microsoft azure, google cloud. https://tinyurl.com/y526vneg.

[Adrian et al., 2015] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J. A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., and Zimmermann, P. (2015). Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *22nd ACM Conference on Computer and Communications Security*.

[Bhargavan et al.,] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P., Zanella-Beguelin, S., Zinzindohoue, J., and Beurdouche, B. Freak: Factoring rsa export keys, 2015.

[Bleichenbacher, 1998] Bleichenbacher, D. (1998). Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs# 1. In *Annual International Cryptology Conference*, pages 1–12. Springer.

[Böck et al., 2018] Böck, H., Somorovsky, J., and Young, C. (2018). Return of bleichenbacher's oracle threat ({ROBOT}). In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 817–849.

[Bradner, 1996] Bradner, S. (1996). The internet standards process–revision 3. Technical report.

[Dierks and Allen, 1999] Dierks, T. and Allen, C. (1999). Ietf rfc 2246: the tls protocol version 1.0.

[Dierks and Rescorla, 2008] Dierks, T. and Rescorla, E. (2008). The transport layer security (tls) protocol version 1.2. Technical report.

[Mell et al., 2011] Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing.

[Rescorla, 2000] Rescorla, E. (2000). Http over tls. Technical report.

[Rescorla, 2018] Rescorla, E. (2018). The transport layer security (tls) protocol version 1.3. Technical report.

[Rescorla et al., 2010] Rescorla, E., Ray, M., Dispensa, S., and Oskov, N. (2010). Transport layer security (tls) renegotiation indication extension. Technical report.

[Rizzo and Duong, 2012] Rizzo, J. and Duong, T. (2012). The crime attack. In *EKOparty Security Conference*, volume 2012.