Global Information Assurance Certification Paper

SSL/TLS: What's Under the Hood

GIAC (GSEC) Gold Certification

Author: Sally Vandeven, sallyvdv@gmail.com
Advisor: Hamed Khiabani

Accepted:
August 13, 2013

Abstract

Encrypted data, by definition, is obscured data. Most web application authentication happens over HTTPS, which uses SSL/TLS for encryption. Did you ever wonder what that authentication exchange looks like in plaintext? What if you are troubleshooting your HTTPS enabled web application and need to dig deeper down in the OSI model than Firebug or other web developer tools will allow? This paper demonstrates how to easily decrypt and dissect a captured web session without either a proxy middleman or possession of the server's private key. It will walk the reader through the simple steps in a TLS connection in an attempt to reveal the unreasonable mystique surrounding encryption protocols.

1. Introduction

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are both protocols used for the encryption of network data. They use encryption, hash functions or message digests, and digital signatures to provide confidentiality, integrity and authentication for data in transit (Rescorla, 2001, pp. 5-7). Wireshark is a feature-rich, free tool that captures and dissects network traffic (Wireshark Protocol Analyzer, 2013). When data is encrypted using the SSL or TLS protocol, it normally looks like gibberish and until fairly recently, Wireshark was not able to decrypt and dissect such traffic unless it had access to the private key of the web server. If, however, the data is web traffic sent to a browser and Wireshark has access to the appropriate session keys it now has the ability to decrypt that data. Current versions of both the Firefox and Chrome browsers will easily save encryption keys in a file that can then be imported to Wireshark. Since documentation on how to set this up and utilize this feature of Wireshark is sparse, what follows are guidelines to help the reader through the process of capturing and analyzing encrypted web traffic using Wireshark, including a step-by-step reference guide in Appendix A.

This paper will begin with a quick refresher of symmetric and asymmetric key encryption. It will be followed with an explanation of how a TLS secure session is setup between two endpoints and how to capture a TLS session along with its encryption keys on a Linux system. It will then provide a detailed analysis of the SSL/TLS protocols using Wireshark to decrypt and dissect an actual TLS data capture. For a very comprehensive examination of cryptographic protocols and algorithms, the reader is directed to the book *Applied Cryptography: Protocols, Algorithms, and Source Code in C* by Bruce Schneier (Schneier, 1996).

1.1. A Brief History of SSL/TLS

The first publicly released version of SSL was actually SSL 2.0, which was released in 1995. It was quickly updated and replaced by SSL 3.0 in 1996. This was considered a complete redesign of the protocol according to one of the leading experts on the SSL protocol, Eric Rescorla (Rescorla, 2001, p. 49). In 1999, TLS 1.0 was released

as a successor to SSL. TLS 1.0 was based on SSL 3.0 and is defined in RFC 2246 (Dierks & Allen, 1999). TLS is very closely related to SSL 3.0, though it does not provide backward compatibility due to changes in some of the algorithms. The name change to TLS from SSL is political in nature and is beyond the scope of this paper; however, Eric Rescorla has written a very interesting account of the history of SSL/TLS through TLS 1.0 (Rescorla, 2001, p. 47). TLS 1.0 was updated to v1.1 in RFC 4346 in 2006 (Dierks & Rescorla, 2006) and again to v1.2 in RFC 5246 in 2008 (Dierks & Rescorla, 2008). The BEAST attack (Dougherty, 2011) against SSLv3 and TLS v1.0 caused some web servers to force the use of TLS 1.1 or 1.2 that have protection against the attack (Rescorla, 2011). On the client side, however, most browsers do not yet support TLS 1.1/1.2 as of this writing (Ristić, 2013).

Best practice dictates that TLS 1.0 or greater be used wherever possible. SSL 2.0 should no longer be used at all. SSL 2.0 was declared insecure primarily due to its re-use of encryption keys as well as its lack of integrity checking of the SSL handshake sequence (Rescorla, 2001, p. 458). Furthermore, a server using TLS 1.0 should also select cipher RC4 for encryption during the cipher negotiation for HTTPS traffic to mitigate against the BEAST attack. The above recommendations are based on the Qualys Best Practice Guide of 2013 (Ristić, 2013). In addition to the BEAST attack of 2011, other recent notable attacks against SSL/TLS include CRIME (Constantin, 2012), Lucky Thirteen (AlFardon & Paterson, 2013) and a new attack against the RC4 symmetric key cipher (Bernstein, 2013).

The acronyms SSL and TLS are commonly used interchangeably to refer to encryption at the transport layer. Because TLS is the most current this paper will use the term TLS unless otherwise specified.

2. TLS

TLS is encryption for data in transit, not data at rest. That means that the end host or recipient in a TLS connection must be able to decrypt the encrypted traffic sent to it in order to be processed and/or displayed in the web browser. When you capture your own data using an Internet browser from a secure connection that you initiated, you are able to

get at all the pieces involved in this process to examine them regardless of the methods used for encryption.

Bruce Schneier describes the differences between symmetric and asymmetric algorithms in his book Applied Cryptography (Schneier, 1996, Section 1.1). Symmetric algorithms like AES and 3DES use a single key for encryption and decryption. That is, the same key that encrypts data is used to decrypt it. Asymmetric algorithms like RSA use two separate keys, one for encryption and the other for decryption. With asymmetric encryption, the key used to encrypt data cannot be used for decryption. The only mathematically feasible method of decryption is to use the second key in that key pair. Symmetric ciphers are computationally much faster than asymmetric ciphers so symmetric ciphers are preferred over asymmetric ciphers when encrypting/decrypting large amounts of data. The traditional problem of shared key management, however, makes using only symmetric ciphers undesirable because it is difficult to securely transport the key prior to the establishment of an encrypted channel. The method most often used is a combination of both asymmetric and symmetric ciphers. Asymmetric ciphers are used to either exchange or generate the keying material from which symmetric "session" keys are derived. Symmetric keys are often referred to as session keys because they are used for a single session and then discarded. They are the shared secrets used for symmetric encryption/decryption of the bulk of the data. This provides a secure and manageable method for exchanging "secret" keys while also allowing for computationally faster cryptographic operations (Schneier, 1996, Section 10.2).

2.1. Sharing session keys

As noted above, session keys are computed on each side of the connection; however, the components used to generate the session keys are passed back and forth in one of two ways. These are often referred to as "key exchange" and "key generation" methods, respectively. The key exchange method uses asymmetric encryption to send a pre-master secret securely to the server. The key generation method is used to exchange unencrypted components, which both sides will then use to derive symmetric session keys (Rescorla, 2001, p. 58). A more detailed examination of both methods follows.

2.1.1. Key exchange method

The RSA and DSA algorithms are commonly used with the key exchange method (Viega, Messier & Pravir, 2009, ch. 8). The client generates a pre-master secret from which both sides will generate the actual cryptographic keys used to encrypt/decrypt the data transferred over this session. This pre-master secret must remain private to protect the confidentiality of the session. It is the seed from which the final keys will be derived. If an attacker had access to the pre-master secret and was able to intercept the TLS handshake the attacker would be able to generate the session keys that would decrypt all the data from that session (Rescorla, 2001, p.140). To protect the pre-master secret in transit, the client encrypts it with the server's public key (one of the keys in an asymmetric key pair) and sends it to the server. Anyone eavesdropping on the connection cannot determine the pre-master secret because it can only be decrypted with the server's private key. The server uses the parameters exchanged during the TLS handshake and the decrypted pre-master secret and applies an agreed upon pseudo random function (PRF) to produce the master secret. The session keys are then derived from the master secret (Rescorla, 2001, p. 82). After each side has generated the final session keys, data can be sent back and forth between the client and the server securely encrypted with these session keys as shown in figure 1.

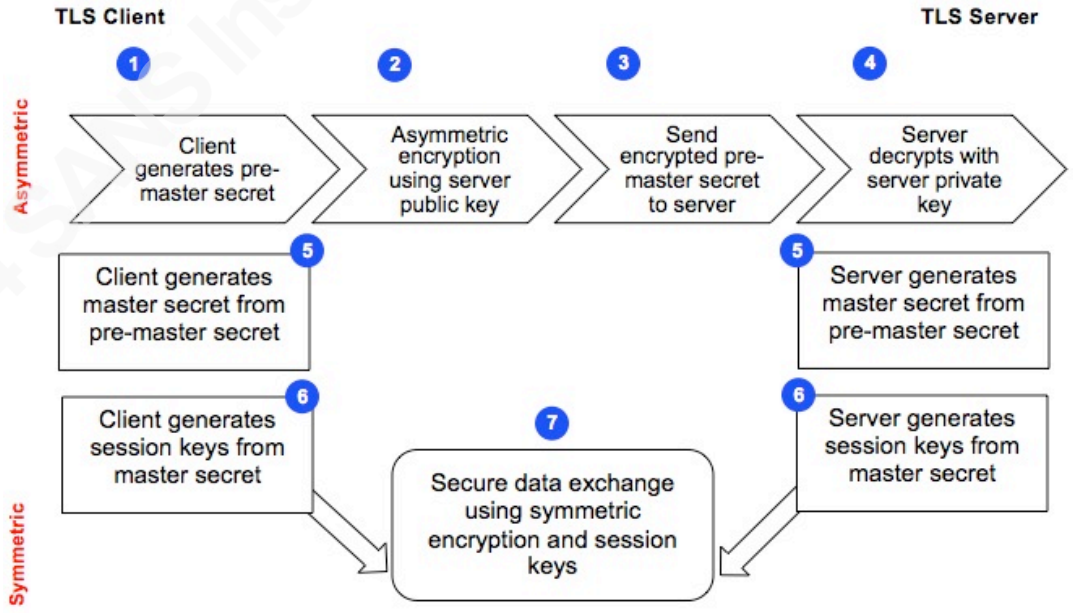

Figure 1. The steps in the RSA key exchange process

2.1.2. Key generation method

A common algorithm used for the key generation method is Diffie-Hellman (DH) (Viega et al., 2009, ch. 8). The client and server independently generate a master secret after an initial exchange of components that are required for that process, all of which can be public and therefore do not require encryption. In addition, each side adds components which must remain private in order to protect the confidentiality of the process, therefore, these are not transmitted across the network but remain private to each side. Intermediate calculations are done on both sides using the private and public components and the results of those calculations are exchanged. These intermediate results may also be public. Similar calculations are done again on each side using the intermediate results. This produces the master secret, which is ultimately converted to session keys for use in symmetric encryption (Rescorla, 1999, p. 1). None of the intermediate private components, the master secret or the final session keys ever travels on the network. They are generated and remain on each side of the connection. The public components that have been exchanged cannot by themselves be used to compromise the Diffie-Hellman key generation process and therefore no encryption is necessary during key generation as shown in figure 2. The important advantage of key generation (DH) over key exchange (RSA) is that if the traffic is intercepted by an attacker during this handshake the attacker does not have enough pieces of the puzzle to compute the master secret and derive the cryptographic keys even if the attacker were to be in possession of the server's private key. Using the key exchange method (RSA), the attacker could independently derive the cryptographic keys and decrypt the exchanged data if she had access to the server's private key.

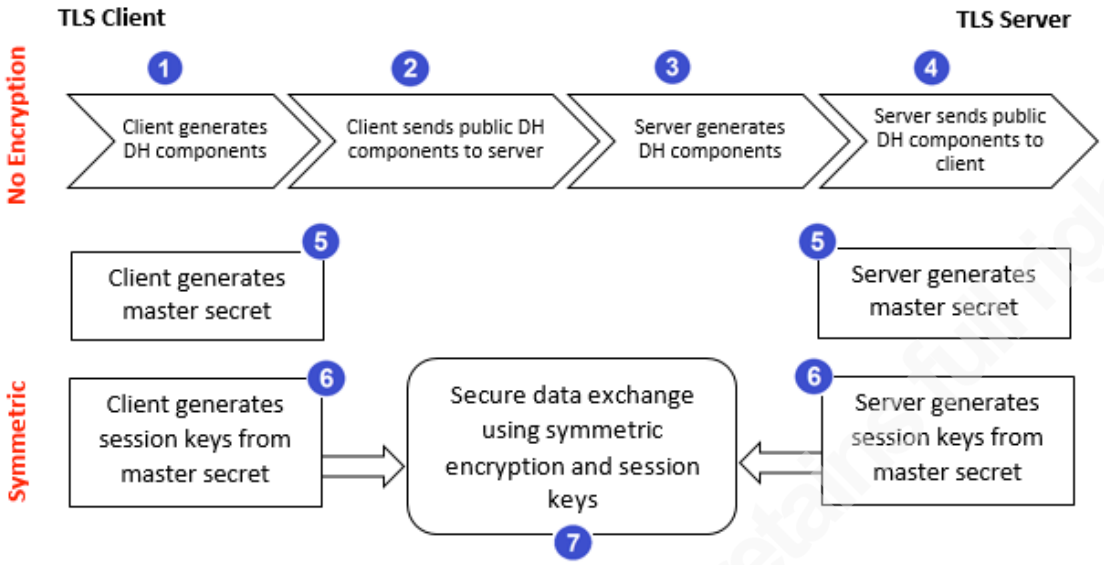

Figure 2. The steps in the DH key generation process

In addition, both key exchange and key generation methods can be ephemeral, meaning they produce "one time use" keys (Barker, Branstad, Chokhani & Smid, 2010). In order to achieve this, each session key is generated using an algorithm that does not include the server's private key in any part of its calculation. The only way to decrypt data that has been encrypted with an ephemeral key is by having possession of the one-time session key. In the case of key exchange methods like RSA, this means that compromise of a session key only compromises a single session. However, since the pre-master secrets are transferred over the network encrypted with the server's private key, a compromise of the server's private key would allow an attacker to intercept the pre-master secret, derive the one-time-use session keys and basically nullify the value of using ephemeral keys. In the case of key generation methods like Diffie-Hellman, pre-master secrets are not used and therefore cannot be intercepted. When using the key generation method, both sides of the connection compute a master key from which the session keys are derived. The only key capable of decrypting a DH session is that session key. Since the components used to derive the session keys do not cross the network, an attacker intercepting network traffic could not discover these keys.

Using the combination of ephemeral keys and the DH key generation method provides what is known as Perfect Forward Secrecy (PFS) (Rescorla, 2001, p. 195). PFS implies that compromise of a server's private key would not compromise past sessions.

The session keys for those past sessions are the **only** keys that will decrypt the session's data. In this case, the server's private key is used exclusively for authentication of the server, not for encryption of any session related keys or data (Rescorla, 2001, p. 147).

2.1.3. Digital Signatures

Bruce Schneier provides an excellent description of digital signatures in his book, *Applied Cryptography* (Schneier, 1996, Sec. 2.6). What follows is a brief summary of his explanation. Digital signatures use the two keys in an asymmetric cipher to provide authentication of a message. The owner of an asymmetric key pair calculates a message digest over a message using a hashing algorithm such as MD5 or SHA1. That message digest is encrypted with the owner's private key and sent over the network along with the message itself as well as the sender's signed public key. The encrypted hash functions as the signature. The receiver can use the owner's public key to decrypt the message hash. The receiver will be able to authenticate the sender by repeating the process and comparing the hashes. That is, the receiver will decrypt the senders encrypted message digest with the sender's public key, independently create a message digest of the received message and compare the message digest it calculated with the message digest it received. If the two match and the receiver trusts the signed public key from the sender then the receiver can be confident that this message came from the sender. An important part of this process is trusting the signed public key of the sender. An organization that issues signed certificates for this purpose is called a Certificate Authority (CA). The CA's function is discussed in more detail in section 2.2.1. It is worth noting here that the certificate trust model has come under attack in the past and is considered by some to be broken. Moxie Marlinspike has written an interesting discussion on CA trust at www.thoughtcrime.org (Marlinspike, 2011). Peter Eckersley has also addressed the trust model at www.eff.org (Eckersley, 2011).

2.2. TLS Handshake

The TLS protocol consists of two sub-protocols, the handshake protocol and the record protocol (Dierks & Rescorla, 2008). The handshake protocol describes the rules used to establish common cryptographic parameters as well as authenticating the server and optionally, the client. The record protocol establishes the rules for breaking up the

data to be transferred, encrypting it and packaging it into "records" such that when it reaches the other side it can be properly decrypted. With respect to the 7 layer OSI networking model, the handshake protocol operates at the session layer and the record protocol operates at the presentation layer.

In order to establish a TLS session using a web browser, a client first contacts a server and declares that it would like to initiate a secure connection with that server. The client does this by sending a TCP packet with the SYN flag set to the port at which the server is listening, usually port 443 for secure web traffic. Assuming the server is configured to support this type of connection, the TCP three-way handshake is completed and the TLS session parameters can be negotiated. This negotiation is referred to as the SSL or TLS handshake (Rescorla, 2001, p. 58). The handshake will authenticate the server and establish the encryption algorithm to be used as well as exchange the components used to generate the cryptographic session keys. The key exchange and key generation methods described in previous sections take place during the TLS handshake. The server may optionally authenticate the client during the handshake as well. Ultimately, session keys will be derived on each side of the TLS connection using the exchanged components and will be used to encrypt and decrypt the data transmitted during the session. Some browsers can capture and log the components used to generate session keys such that captured network traffic can be decrypted either at the time of capture or using previously captured packets. This is possible using both the key exchange and the key generation methods. Wireshark has a built in capability to perform this decryption of TLS traffic if it has access to these components, which will be described in more detail below.

2.2.1. The Handshake Protocol

The TLS handshake begins when the client sends the *ClientHello* Message. The three most important elements in this message are the version of TLS used by the client, a random string that will be used for generating cryptographic keys and the cipher suites that the client supports. The cipher suite consists of a code that represents four parameters: authentication algorithm, key exchange method, encryption cipher and hashing algorithm (Rescorla, 2013). The client lists these ciphers in its preferred order;

in other words, it would prefer to use the cipher listed first if the server supports it. The server replies with a *ServerHello* message, which contains its choice of cipher from the client's list and a random string, again for use later when generating cryptographic keys. Notice that it is the server that ultimately decides on the actual cipher to be used for encryption. If the server is not satisfied with any of the cipher choices offered by the client, it will send a "Handshake Failure" message back to the client and per RFC, sending a TCP packet with the FIN and ACK flags set gracefully terminates the connection (Dierks & Allen, 1999). Next, the server sends its certificate. The certificate is the server's proof of identify so the client can authenticate the server. The client examines the certificate and if it is valid and digitally signed by a CA that the client trusts, the client can be confident that it is talking to the correct server and not an imposter. A more thorough discussion of CA trust follows in section 5.5.3.

A certificate is valid if it meets 3 criteria: the certificate must be signed by a CA that the browser trusts; the domain name on the certificate must match the domain presenting the certificate; and the certificate must not have expired. Additionally, some browsers may confirm that the certificate has not been revoked before declaring it valid (Viega et al., 2009, Sect. 1.2). There are also optional certificate extensions that, if present, should be checked by the client. One important extension is "key usage". This is a field in the certificate that specifies what the certificate may be used for. Examples are digital signature, non-repudiation and data encipherment. A complete description of the key usage option for certificates is described in RFC 2459 (Housley, Ford, Polk & Solo, 1999). If the certificate is self-signed, unsigned or not signed by a trusted third party the browser will usually warn the user that the identity of the server cannot be properly verified. A self-signed certificate is a certificate that has been signed using the server's private key, not the private key of a trusted third party.

If the method of key exchange chosen by the server is DH, the server sends the *ServerKeyExchange* message that contains the public components that will be used to generate the master secret (Rescorla, 2001, p. 35). Finally, the server sends the *ServerHelloDone* message that indicates to the client that the server is finished passing parameters.

Once the cipher suite has been agreed upon, the client and server can begin trading more specific parameters. First, the client sends a *ClientKeyExchange* message and a *ChangeCipherSpec* message. The *ClientKeyExchange* message contains the pre-master secret, which is generated by the client. Both the client and the server will use this to generate what is called a master secret from which they both will derive the final cryptographic keys. The client encrypts the pre-master secret with the server's public key so if an eavesdropper were to intercept this bit of data he could not decrypt it. If the cipher chosen for this session was DH, then the *ClientKeyExchange* contains the client's public DH parameters instead of the pre-master secret. The *ChangeCipherSpec* message is just a message from the client to inform the server that all data the client sends from here on will be encrypted using the agreed upon parameters. At this point, both the client and the server have all the components necessary to generate the master secret and then derive cryptographic session keys.

Figure 3. The TLS handshake using the key exchange method

Both sides verify that the handshake has proceeded as planned and that both have generated identical keys by sending an encrypted *Finished* message. First, the client sends an encrypted *Finished* message to the server. This message is a message digest, also called a cryptographic hash, of various components used during the handshake process, not all of which have been transmitted on the network. This is encrypted with the newly generated session key and sent to the server. Next, the server sends a *ChangeCipherSpec* message to the client to tell the client that all data after this message will be encrypted. This is followed by an encrypted *Finished* message from the server to

the client. The final step in the TLS handshake is for each side to decrypt and verify the *Finished* message that it received from the other side. They do this by checking each other's work. Both sides perform a computation using the master-secret and a hash of the handshake messages as inputs. This gets compared to the value received in the *Finished* message from the other side. They will conclude that the handshake has succeeded if the values match. If there was a problem with the verification then a *HandshakeFailure* message is sent and the session is terminated (Rescorla, 2001, p. 81). Figures 3 and 4 show graphical illustrations of TLS handshakes, one using RSA key exchange and the other DH key generation. Notice that the handshake in figure 4 using DH key generation has an extra step. If the server selects a DH cipher from the list of ciphers presented to it from the client, then it must also send the parameters that the client will need for generation of the master secret in the *ServerKeyExchange* message.

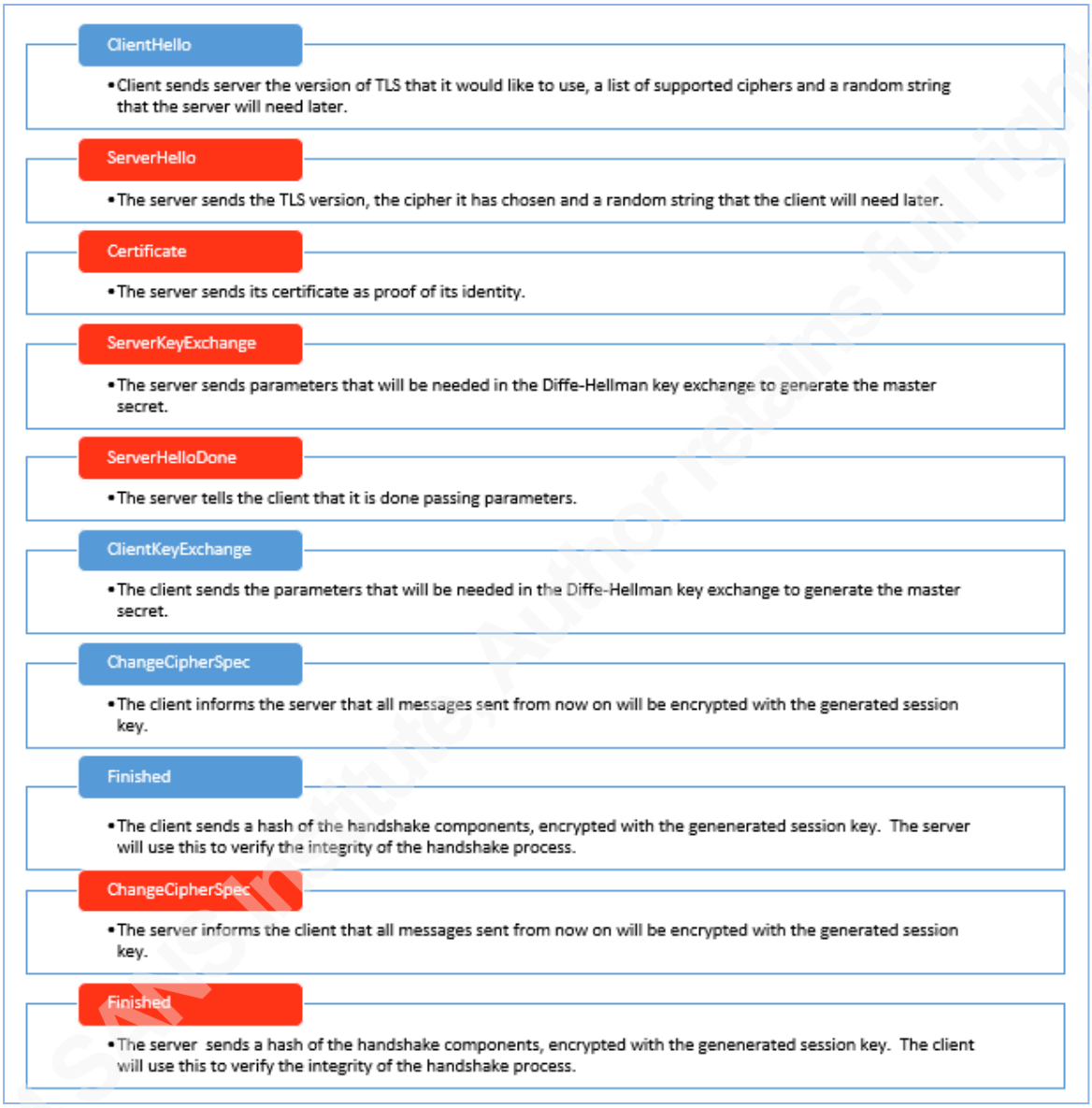

Figure 4. The TLS handshake using the key generation method (Diffie-Hellman).

2.2.2. The Record protocol

Now that the encryption parameters have been established and session keys generated the real purpose of this connection – data transfer can take place. This is where the record protocol comes in. The record protocol is used to break the stream of data into pieces for transmission (Rescorla, 2001, p. 61). Each piece is encrypted and encapsulated with a header letting the other side know the specifics about that particular

piece. Together this header plus data is called a "record". The encrypted records contain the confidential data that is transmitted back and forth. All packets starting with the *Finished* messages of the TLS handshake are encrypted using the newly generated session key. As we will see later, Wireshark is able to reassemble and decrypt these records when given access to the browser's key log file, which contains the values needed to derive the session key.

3. Capturing pre-master secrets/master secrets

Not all browsers support the logging of TLS session components. As of this writing, Firefox 19 and Chrome 24 so support this feature.

Both Firefox and Chrome will check to see if the user has an environment variable set for "SSLKEYLOGFILE" which points to a writable file. If so, the browser will write the secrets to the file identified by the environment variable. This should be a user environment variable, not a system-wide or global variable since these secrets can be used to decrypt captured web traffic which may contain sensitive data.

3.1.1. Environment Variables

On a Linux system user environment variables may be set using any of several files. For example, the users environment variables may be set in ~/.profile, ~/.bash_profile, ~/.bashrc, etc. Typically, the way to create the file and set the environment variable on a Linux system is like this:

```
$ touch /protected/folder/my_session_keys.log
$ chmod  700 /protected/folder/my_session_keys.log
```

Then add the following line to whichever file is executed at login, for example, for user account *student* this might be /home/student/.profile

```
export  SSLKEYLOGFILE=/protected/folder/my_session_keys.log
```

Typing "`echo $SSLKEYLOGFILE`" after a successful login will confirm the value assigned to the variable.

3.1.2. The key log file

Network Security Services (NSS) is an open source set of C libraries, maintained primarily by Mozilla, that contain cryptographic modules. NSS developed modules for key log file operations as well as the format of the key log file (Mozilla Developer Network, 2013). The key log file itself is a flat text file. If the proper environment variable is defined and a file exists that the browser can write to, both Firefox and Chrome utilize the NSS libraries to write pre-master secrets and master secrets to a file. The file format is defined as follows (Combs, 2012):

CLIENT_RANDOM<space>|64 bytes Client Random Values in Hex|<space>|96 bytes Master Secret in Hex|

RSA<space>|16 bytes encrypted pre-master secret in hex|<space>|96 bytes pre-master secret in hex|

```
# SSL/TLS secrets log file, generated by NSS
CLIENT_RANDOM
51a3e7b9041587fb8143cd2c3c212a2c27c40e872c8fc7d10e995484c74d623d
04b1dd60fd0a1a9460ae66d3bdf76eae74f97459352b366b66b3f71150957b7b282
65992c0b0c74d953acd46beea80ac
RSA 0c9b7e07178f02e8
03014fbec62aa7e665957155cb841cbb8f04d4ef48d24160ee40bf1a90aa964c56d4
53711af4abad169edd1f04466282
```

Figure 4a. Sample entries from a populated key log file

The first entry in figure 4a , CLIENT_RANDOM, is the format for TLS sessions using non-RSA cipher suites. The master secret is saved for non-RSA sessions. The second entry is for TLS sessions using an RSA cipher suite. The pre-master secret is saved for RSA negotiated sessions. Wireshark will be able to read in these "secrets" and generate the session keys, which will decrypt the captured data. Because the key log is a flat text file written in ASCII, these are not raw hex values used by the browser, instead they are the ASCII representation for the hex values. For example, the *ClientRandom* value sent as part of the TLS handshake is 32 bytes long. Writing the ASCII representation of a 32-byte value in hexadecimal requires 64 bytes.

4. Capturing Traffic

There are several different tools that will capture network traffic. Two common free tools are tcpdump and Wireshark. Wireshark is an open source GUI based application that runs on many platforms including Linux, Windows and OS X (Wireshark Protocol Analyzer, 2013). Wireshark has a command line version as well, called tshark that is typically installed during a Wireshark install. Tcpdump is a command line application that runs on many Unix and Linux operating systems (Tcpdump & Libpcap, 2013). Windump is the port of tcpdump for Windows (WinPcap, 2013). This paper will address packet capture using tcpdump, Wireshark and tshark on Fedora Linux.

4.1. Capturing network traffic with tcpdump and tshark

Unix–like machines may have tcpdump installed by default. If not, it can be downloaded from www.tcpdump.org. Tcpdump also requires that the network packet capture library, libpcap, be installed. This is also distributed at www.tcpdump.org. For capturing traffic, tshark syntax is very similar to tcpdump; however, tshark has many more options than tcpdump for displaying data from packet captures. Documentation for tshark can be found on the tshark man page (Combs, 2013).

Figure 5 illustrates how to capture packets on network interface "eth0" and write the data to a file called "packets.pcap" in both tcpdump and tshark. The –n option instructs tcpdump not to resolve names while it is capturing data and the -w option indicates that tcpdump should write to a file. The key combination of Control-C will end the packet capture and close the file for both utilities.

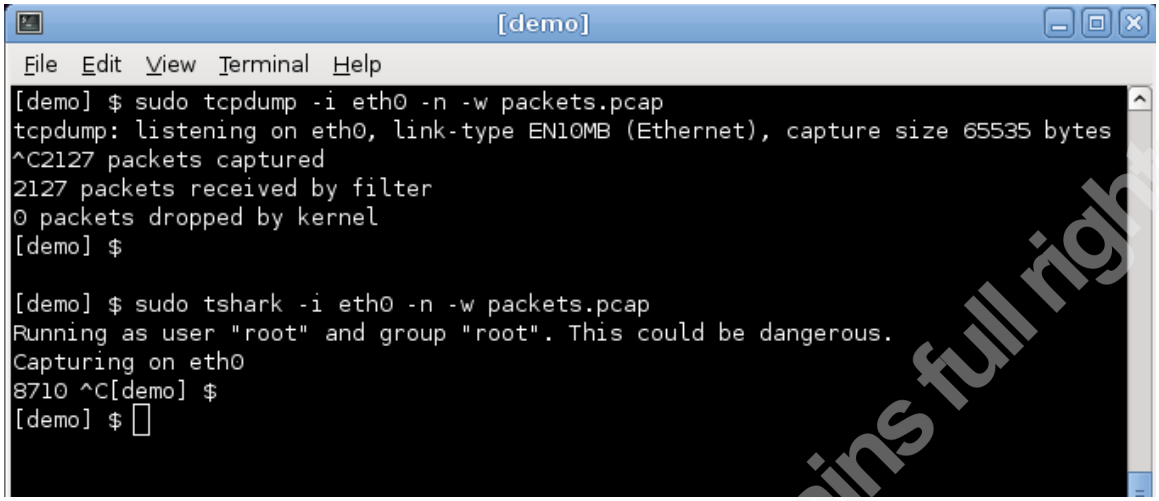

Figure 5. Capturing packets using tcpdump and tshark

4.2. Capturing network traffic with Wireshark

Capturing traffic using Wireshark is very simple. Select "Capture → Options…" to view the capture dialogue. Select the network interface and any other desired options, such as, unchecking the name resolution boxes (Figure 6). Using "promiscuous mode" on all interfaces will capture all traffic that an interface can "see" and is sufficient for this example. Click on the *Start* button to begin the capture. To conclude a packet capture in Wireshark select "Capture → Stop" for the menu bar or click on the *Stop Capture* button on the toolbar. The captured packets are displayed in the Wireshark 3-pane interface.

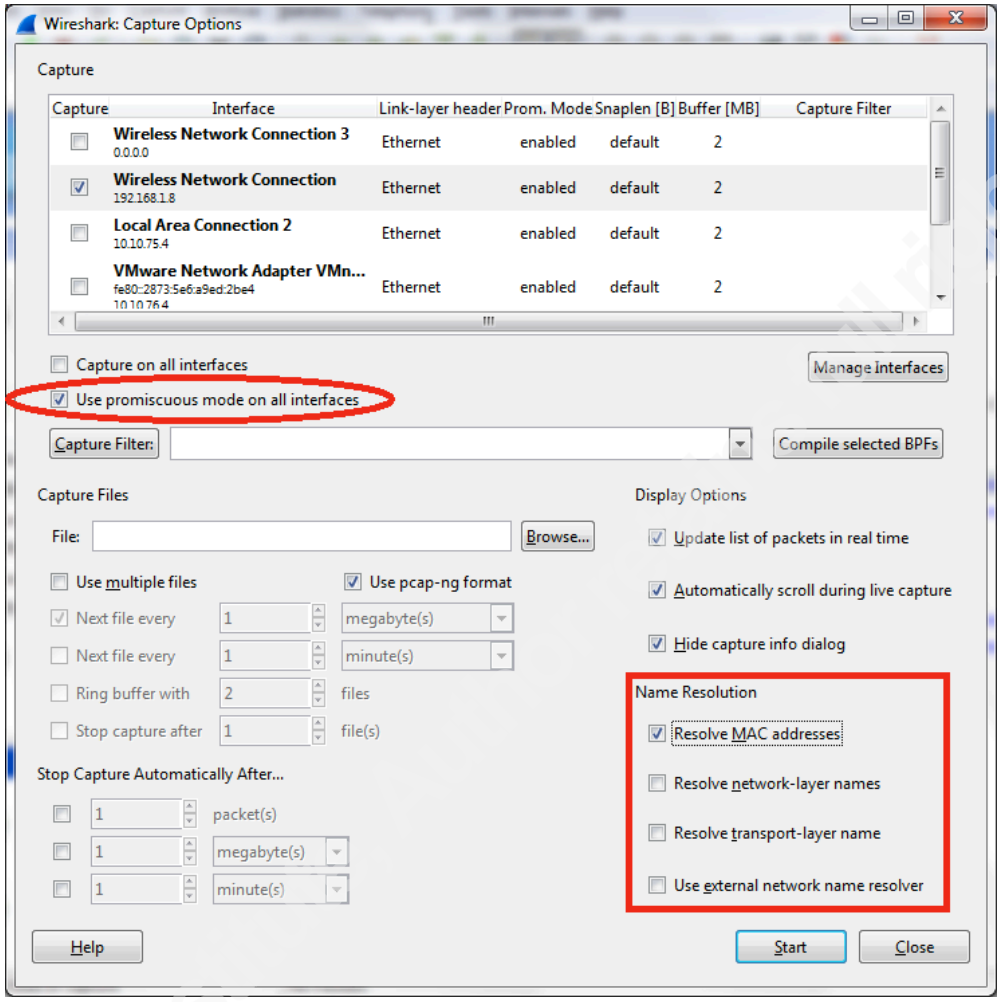

Figure 6. Capturing packets using Wireshark

5. Analyzing Traffic

For the results and discussion that follow, I created a test lab and spent many hours using Wireshark, tshark and tcpdump to view network traffic. This section will illustrate how the TLS handshake works using a captured TLS session establishment and login sequence from a temporary SANS portal account, which has since been removed. The handshake records comprise the TLS session establishment and negotiation of the session parameters. That is followed by "Application data" records that make up the actual login to the SANS account. Appendix A contains a step-by-step guide for this process without the explanations.

5.1. Set environment variables

Prior to capturing the data there are a few things to set up. First, the key log file must exist as well as the environment variable "SSLKEYLOGFILE". This will signal the browser to save TLS pre-master/master secrets and allow decryption of the data outside the browser. The procedure for setting this up is described above in "Capturing pre-master secrets/master secrets", or refer to Appendix A.

5.2. Capture network traffic

Using tcpdump, the following command will start capturing all network traffic coming in to the interface eth0 (-i), DNS names will not be resolved (-n) and captured data will be written (-w) to the file sansLogin.pcap.

```
# tcpdump –i eth0  -n –w  sansLogin.pcap
```

Now that network traffic is being captured, open up Firefox or Chrome and login to a portal account. This experiment has worked with Firefox versions 19 through 22 and Chrome versions 24 through 30. Since this feature is not yet well documented, it is unclear what the future support will be.

This short example includes only a login to the SANS portal account followed immediately by a logout. When the logout sequence completes, terminate the packet capture in tcpdump by entering Control-C. If the packet capture terminates before the TLS session is properly terminated then Wireshark will not have all the packets it needs to properly reassemble the session and will therefore be unable to decrypt the traffic.

At this point, the captured traffic from a login/logout session exists in sansLogin.pcap and the keys for that session have been written to the key log file. To verify that the keys were successfully written to the key log file:

```
# less  $SSLKEYLOGFILE
```

5.3. A look at the traffic in Wireshark

Open the new packet trace file in Wireshark, noting that Wireshark does not require administrative privileges to open a previously captured trace file because the default permissions that it assigns allow anyone to read.

```
$ wireshark –r  sansLogin.pcap
```

In Wireshark select Analyze → Conversations from the menu bar at the top of the screen. Click on the tab for TCP. This will display a summary of the different TCP conversations in this trace file (Figure 7).

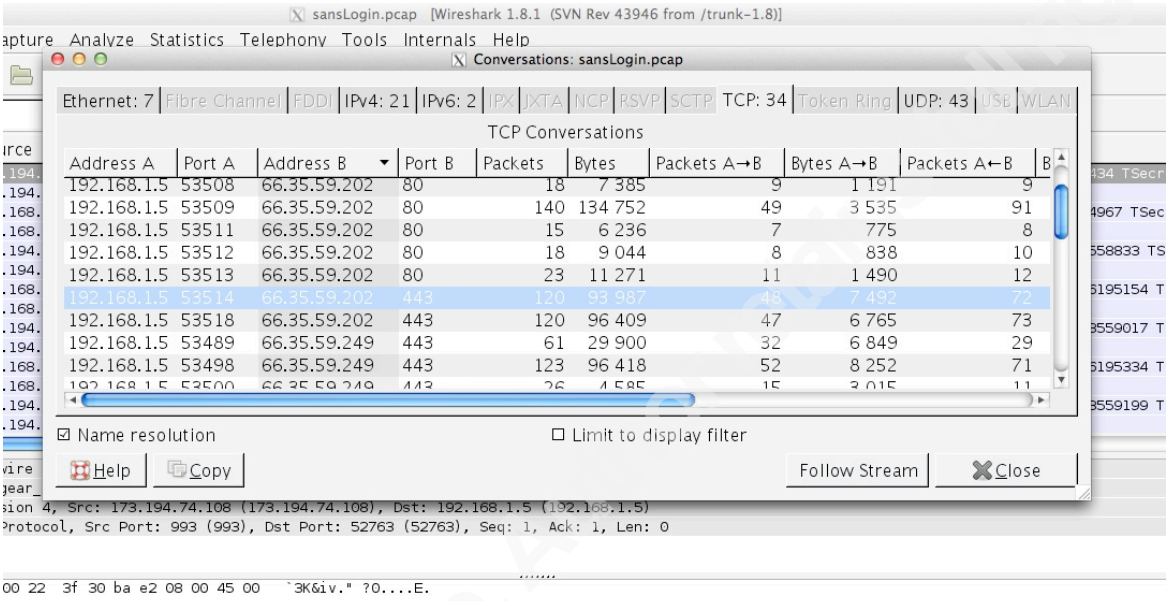

Figure 7. Wireshark's "Analyze Conversations" window

The IP address resolution for www.sans.org at the time of this capture was 66.35.59.202 and the port for the HTTPS connection was 443. The highlighted line represents the conversation for the actual login session. Select "Follow Stream" and Wireshark will filter on all packets related to that conversation only. The actual meat of the traffic, the payloads without the TCP handshake packets and packet headers, are arranged neatly in a pop up window that is in two colors, one color for the client to server payloads and the other color for the server to client payloads. This is often red for client initiated packets and blue for server initiated packets but the default can vary per platform.

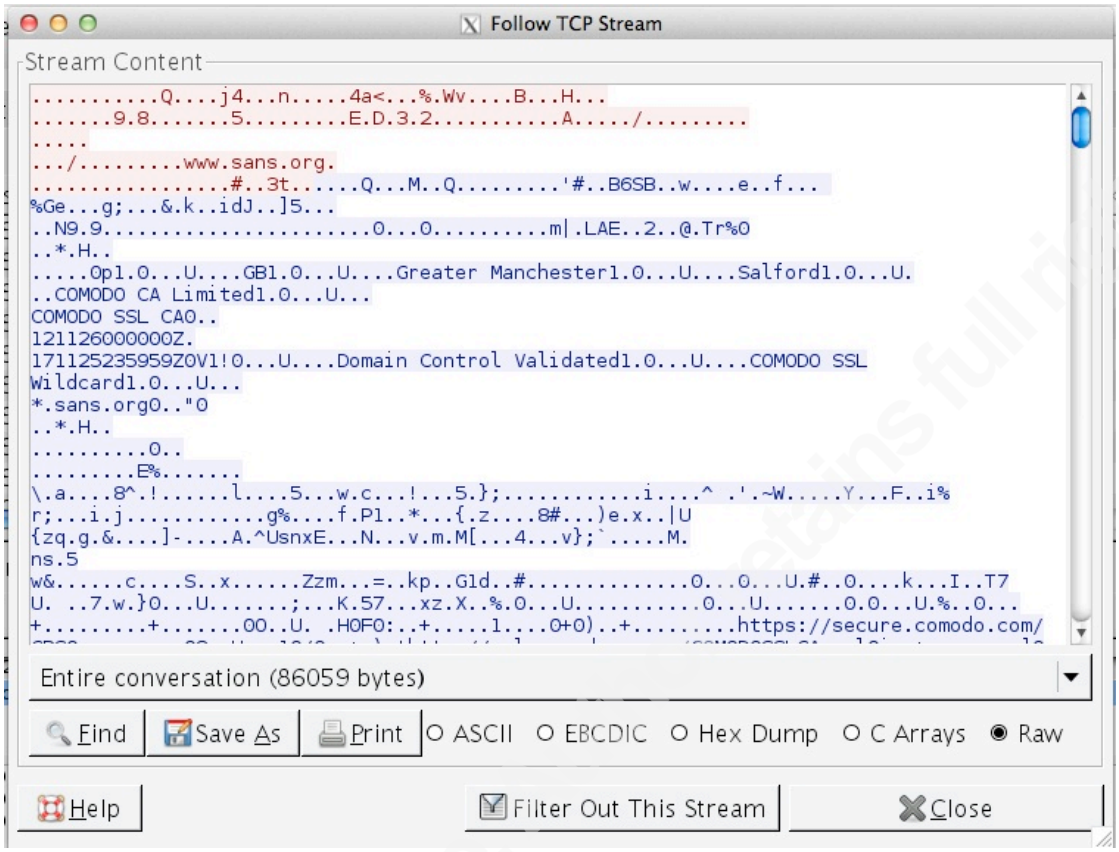

Figure 8. "Follow Stream" shows the payloads for an entire TCP stream.

The data shown in figure 8 is the raw data from the TCP payloads in this stream. The first portion of the SSL handshake, including the transmission of the server's certificate can be seen in figure 8. Further down in the stream, after the completion of the TLS handshake, the data would be encrypted and therefore unreadable. Close the "Follow TCP Stream" window and look at the top pane in Wireshark. The display filter shows that only the packets associated with this single TCP stream are being displayed. The first 3 packets establish the TCP connection between the client and the server using the sequence of TCP flags [SYN], [SYN, ACK] and [ACK]. What follows the TCP three-way handshake is the handshake to establish the TLS session parameters. After the TLS handshake comes the actual data that is being transmitted, which Wireshark labels "Application Data".

5.4. Dissecting the TLS handshake

The first step in the TLS handshake, the *ClientHello*, follows the TCP handshake and is underlined in figure 9.

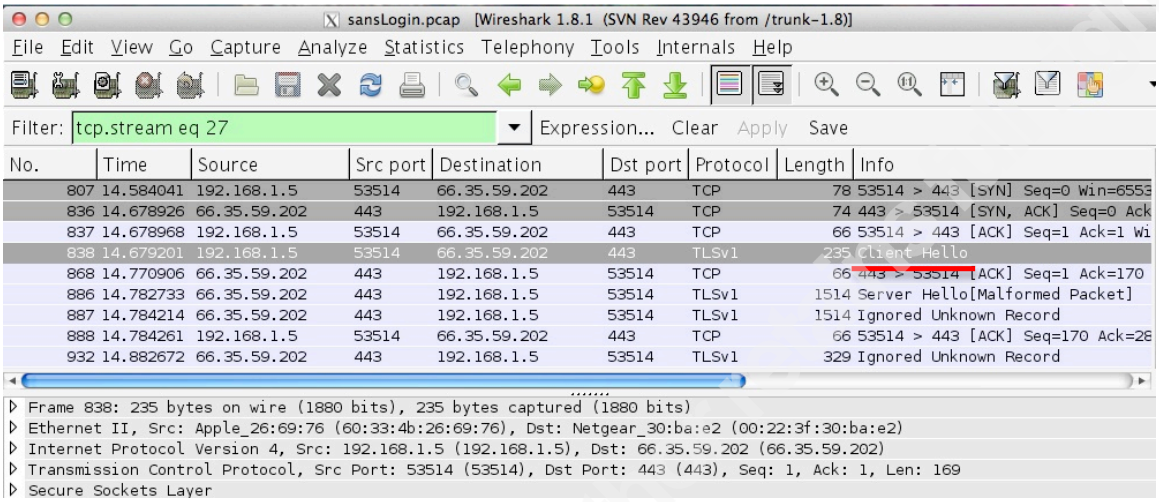

Figure 9. *ClientHello* portion of handshake

To filter for only the packets that comprise the TLS handshake, add the following to the display filter: "`&& ssl.handshake`" and *Apply* (Figure 10). This will narrow the selection and display only the handshake packets for this particular conversation.

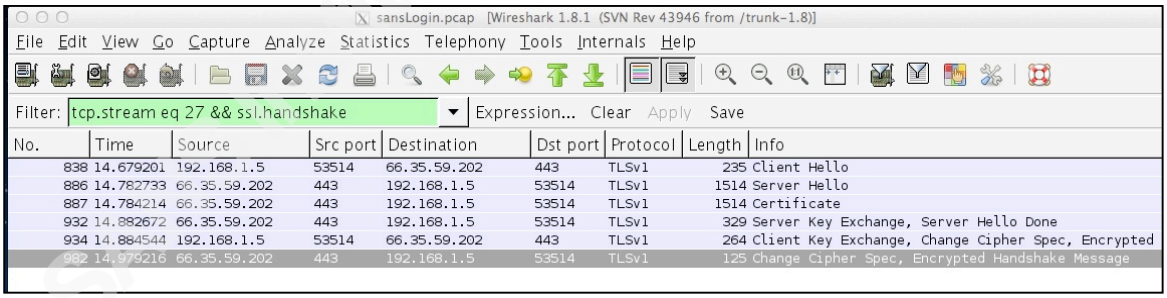

Figure 10. Display filter showing single SSL handshake sequence

The handshake consists of 10 distinct steps and there is an extra step when using the DH key generation method. The TLS session captured in figure 10 shows the steps of the TLS handshake transmitted in 6 packets. This is because the entire TLS stream, starting with the handshake records, is broken up at the application layer into the maximum size pieces that may be transmitted at a given time over the network to reduce overhead and improve performance (Rescorla, 2001, p. 179). In other words, some of the handshake records are combined and sent in one packet. Other records, like the

certificate are too long to fit into one packet and so are split up and sent in two packets. The TLS data is the embedded protocol data in the TCP packets. Each packet will contain a frame header, and IP header and a TCP header. Following the TCP header is the TLS data.

RFC 2246 defines the format for the TLS records (Dierks & Allen, 1999). There are specific TLS header values to identify what type of content is contained in each record. Wireshark uses these specifications to parse the data and display it properly. Each record can consist of one of four "Content Types", *Alert*, *ApplicationData*, *ChangeCipherSpec* or *Handshake*. *Alert* messages are to signal warning or error messages, *ApplicationData* contains the encrypted data being sent back and forth, *ChangeCipherSpec* is the message informing the other side that encryption will begin and the *Handshake* content type is used while setting up the session. Because there are different fields for each Content Type, there is a formal definition for these header fields, defined in RFC 2246 (Dierks & Allen, 1999, pp. 48-50). Header fields can take the following values:

Content Type

ChangeCipherSpec = 0x14
Alert = 0x15
Handshake = 0x16
ApplicationData = 0x17

Protocol Version

SSL v3 = 0x0300
TLS v1.0 = 0x0301
TLS v1.1 = 0x0302
TLS v1.2 = 0x0303

Handshake Message Type

ClientHello = 0x01
ServerHello = 0x02
Certificate = 0x0B
ServerKeyExchange = 0x0C
ServerHelloDone = 0x0E
ClientKeyExchange = 0x10
Finished = 0x14

Alert Message Type

close_notify 0x00	certificate_unknown 0x2E
unexpected_message 0x0A	illegal_parameter 0x2F
bad_record_mac 0x14	unknown_ca 0x30
decryption_failed 0x15	access_denied 0x31
record_overflow 0x16	decode_error 0x32
decompression_failure 0x1E	decrypt_error 0x33
handshake_failure 0x28	export_restriction 0x3C
no_certificate 0x29 SSLv3 only	protocol_version 0x46
bad_certificate 0x2A	insufficient_security 0x47
unsupported_certificate 0x2B	internal_error 0x50
certificate_revoked 0x2C	user_canceled 0x5A
certificate_expired 0x2D	no_renegotiation 0x64

Figure 10a. TLS record header field values

Below are the formats of the header fields for the four different record types: Handshake (Figure 11) , ChangeCipherSpec (Figure 12) , ApplicationData (Figure 13) and Alert (Figure 14).

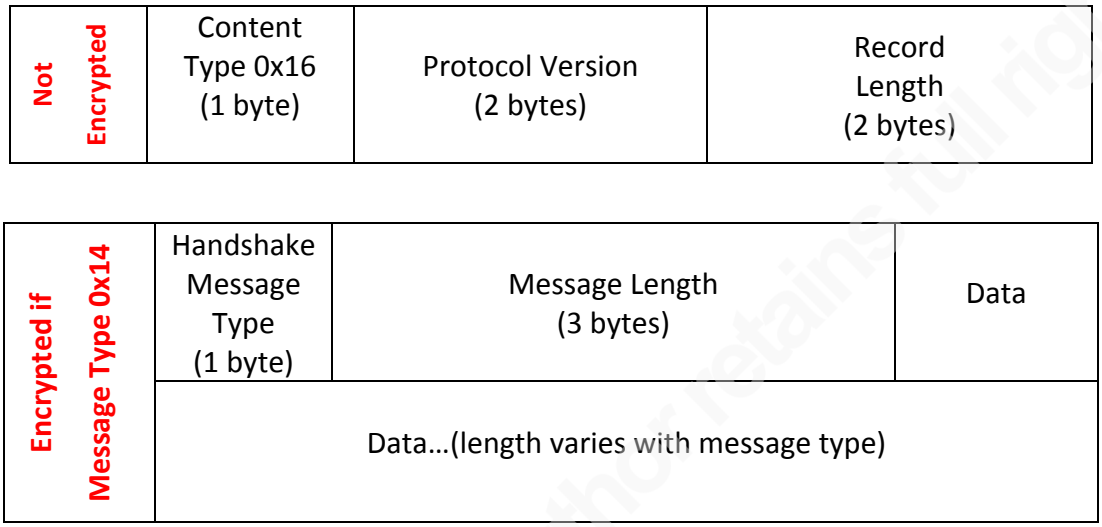

Figure 11. Handshake Record Layout

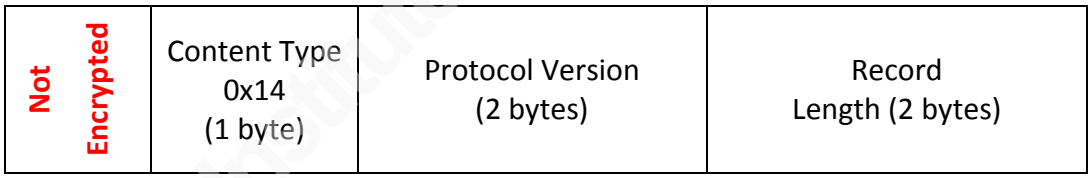

Figure 12. ChangeCipherSpec Record Layout

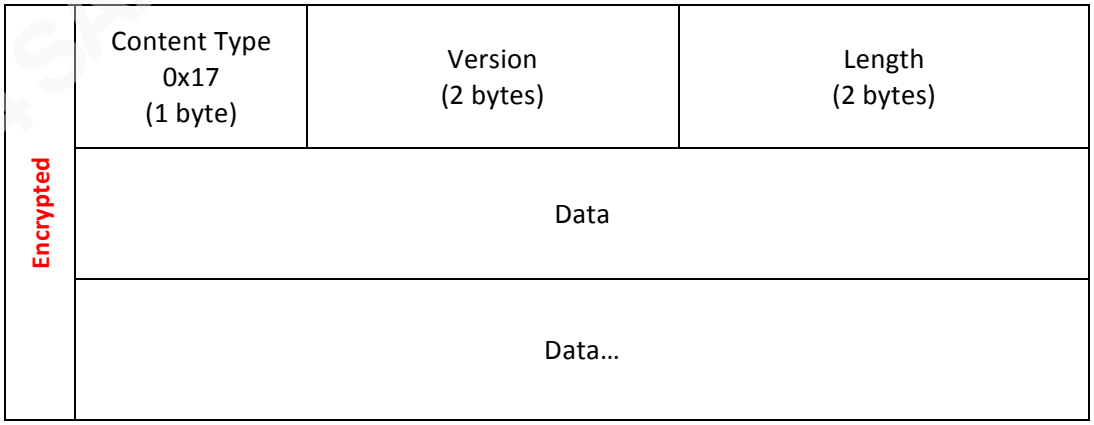

Figure 13. ApplicationData Record Layout

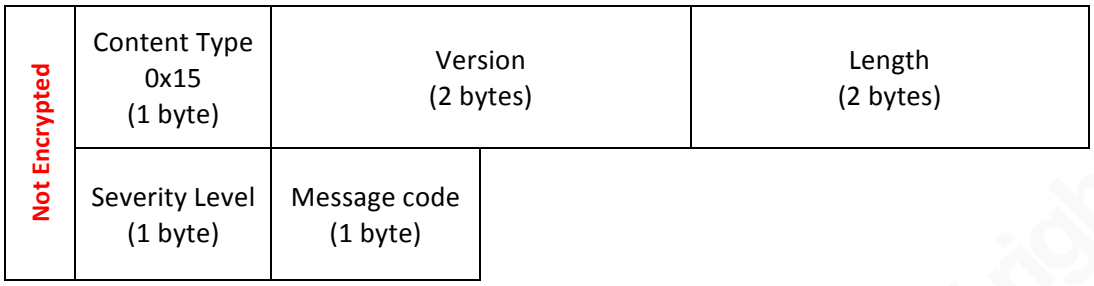

Figure 14. Alert Record Layout

5.5. Handshake records in detail

5.5.1. ClientHello

Since the client initiated this TCP connection, it will also start the negotiation with the server regarding TLS parameters, making suggestions that the server must either agree to or suggest a suitable alternative in order for the connection to be successful. The client presents the parameters shown in figure 15. The handshake header fields are outlined in red and contain handshake type, handshake length and TLS version. Following the header fields, are the content type specific fields. This particular record is a *ClientHello* record and contains the random data that the client will use when establishing keys later, the cipher suites that the client supports, the server name it would like to authenticate, etc. By clicking on an individual item in the middle pane, Wireshark will highlight the corresponding hex value for that item in the lower pane. Conversely, clicking on any value in the lower pane in Wireshark will show the corresponding parsed value with its identifier in the middle pane. The TLS record header is outlined in red in figure 15 and can be mapped to the header fields shown in figure 11. These five bytes define the Content Type (0x16 is a handshake record), TLS version (0x0301 is TLS version 1.0), and the Length of the record following the header fields (0x00A4 or 164 decimal).

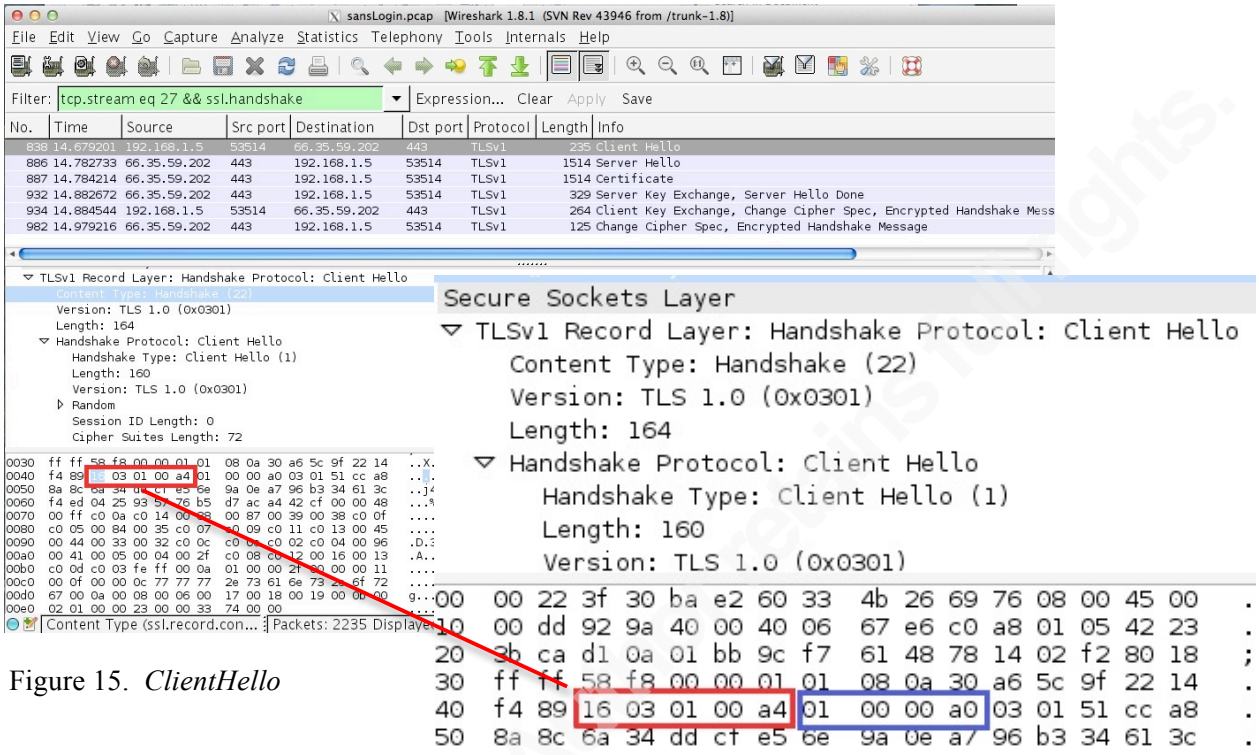

Figure 15. *ClientHello*

The handshake message header fields follow the TLS header fields and are outlined in blue. Figure 15 shows a *ClientHello* message, identified by the 0x01 in the handshake message type field. This is followed by the 3-byte message length field (0x0000A0 or 160 decimal) and then the data associated with this *Handshake* message type that is of variable length. Most of relates to the negotiation of cryptographic parameters, such as, which cipher to use, whether or not to use compression on the data, etc.

Highlighting and expanding the Cipher Suites field in the middle pane shows the numeric representations of all the cipher suites that the client will support. Each cipher suite is represented by a 2-byte value. Wireshark resolves this 2-byte value to the RFC defined ciphers. A full listing of these values is maintained by IANA and can be found at the IANA web site (Rescorla, 2013).

Figure 16 shows the ciphers supported by Firefox 21, which was used for this test.

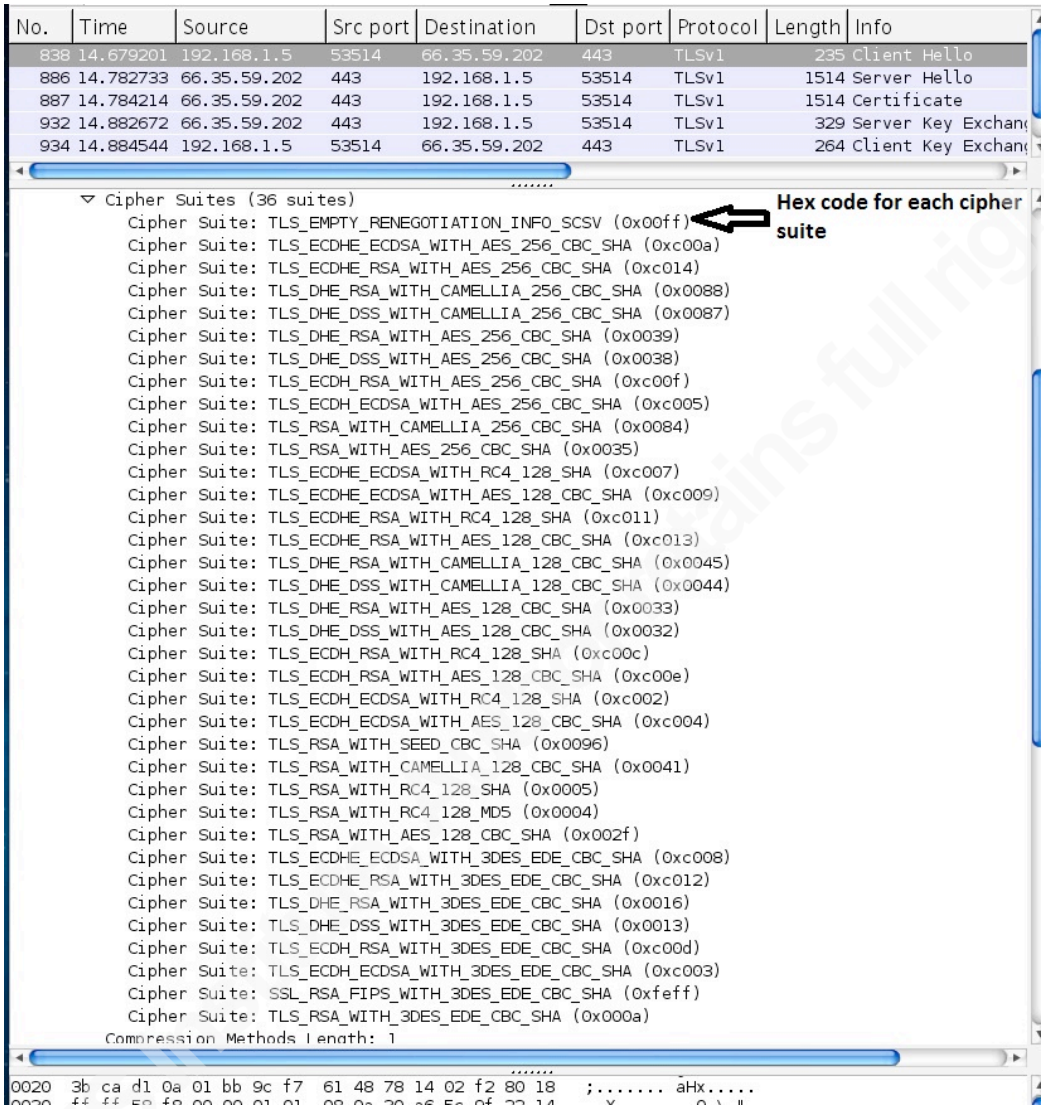

Figure 16. Cipher Suites supported by Firefox 21

5.5.2. ServerHello

Once the client is finished sending its proposed parameters to the server, the server will respond with a *ServerHello* message (Figure 17). This portion of the TLS handshake will inform the client which of the ciphers it has chosen from the list presented by the client in the *ClientHello* message. It also passes its "random" values for key generation later, confirms the version of SSL/TLS used and confirms which compression method, if any, will be used on the encrypted data. For this particular session, no compression will be used as shown by the Compression Method of "null".

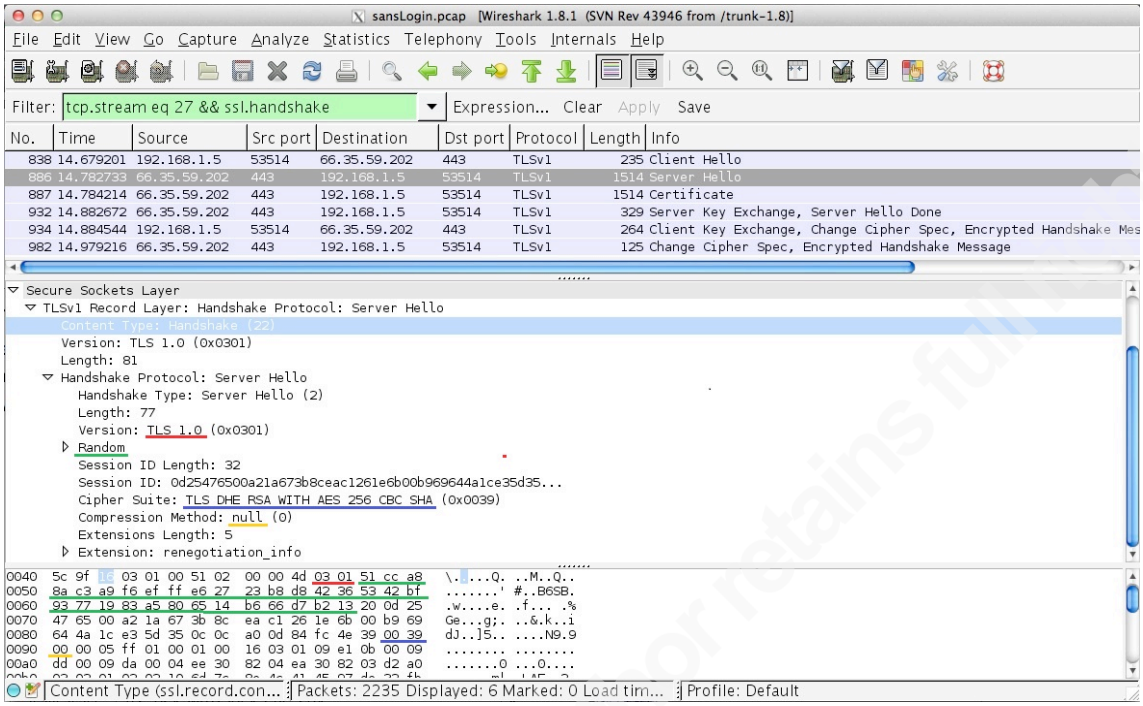

Figure 17. *ServerHello* message

5.5.3. Certificate

The server's certificate is its credential for authentication. The server sends the client its certificate that also includes the server's public key. Either an intermediate Certificate Authority or a root Certificate Authority digitally signs the server's certificate. If the server's certificate is signed by an intermediate CA it must attach that certificate as well. The intermediate CA's certificate may be signed be another intermediate CA or a root CA. All intermediate CA certificates must be attached in what is referred to as a "certificate chain". The certificate chain is complete when a root CA has digitally signed a certificate. Browsers contain a store of trusted root CA's. If a certificate from an intermediate CA has been signed by a root CA in the browsers trusted store, then according to this trust model the authenticating server can also be trusted. The client verifies three things: an authority that the browser trusts has digitally signed the certificate; the domain name on the certificate matches the domain that is presenting the certificate; and the certificate has not expired (Schneier, 1996, Sec. 2.6).

If the certificate checks out, the client will proceed. If the cipher suite negotiated was not a DH cipher then the server's public key will be used to encrypt the pre-master secret at a later stage in the handshake. If the cipher suite negotiated is a DH cipher, then

the server's public key is not needed for anything else. There are many good resources online for further information about certificates and Public Key Infrastructure (PKI) including the already referenced book by Bruce Schneier, (Schneier, 1996). In the example here, the length of the certificate is more than can be sent in 1514 byte datagram so it is split up. The first portion is sent in the *ServerHello* message because that packet was not near its maximum length. Wireshark shows the certificate in the following handshake packet because that is the packet that reassembles the two portions of the certificate. Figure 18 shows the certificate bytes beginning immediately following the *ServerHello*. The Content Type of 0x16 and the *Handshake* Message Type of 0x0B indicates that a certificate follows.

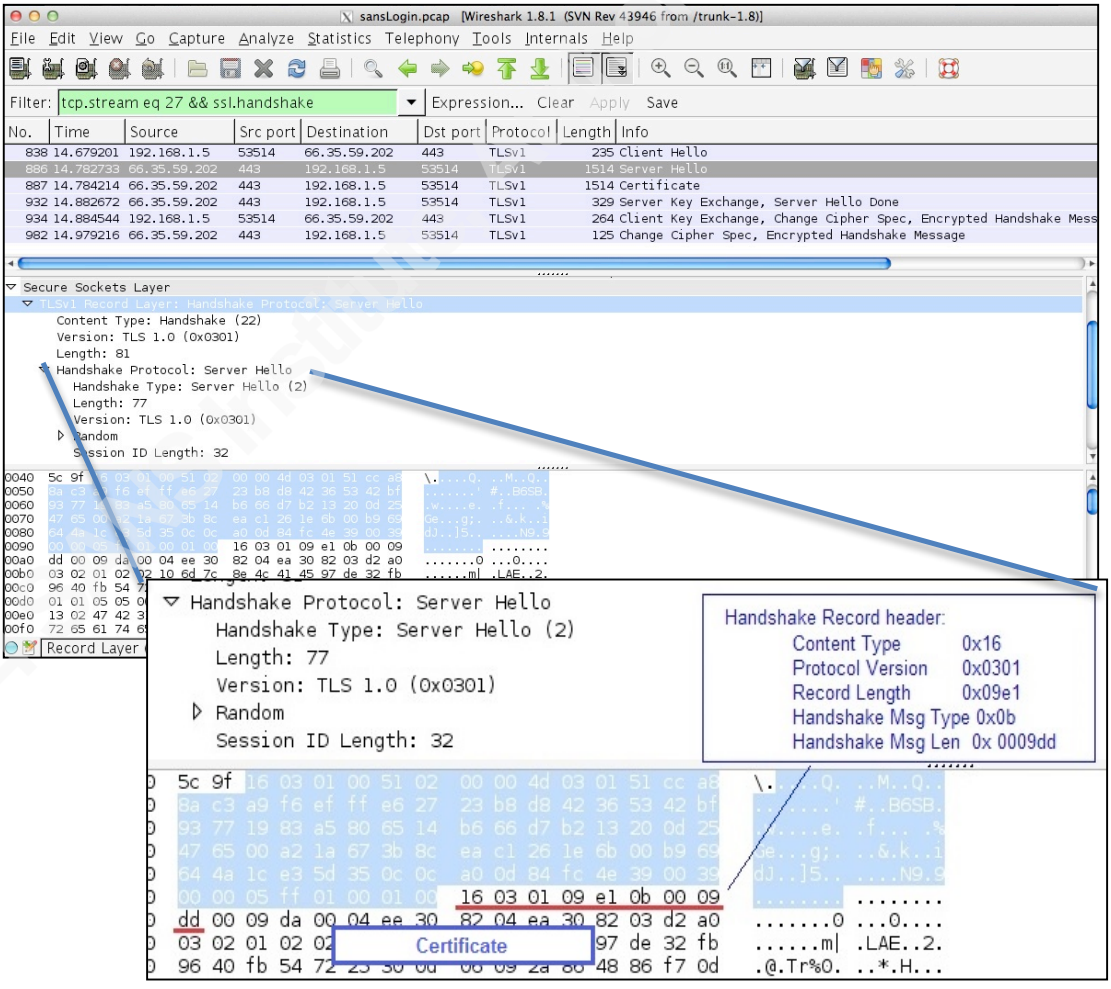

Figure 18. *ServerHello* Message with first section of server certificate

The next record in the Wireshark display is the *Certificate* message. The second and complete portion of the server certificate is transmitted in this packet. Expanding the *Certificate* section in the middle pane in Wireshark will identify the various components of the certificate (Figure 19).

Click on any component to see the corresponding highlighted value for that field.

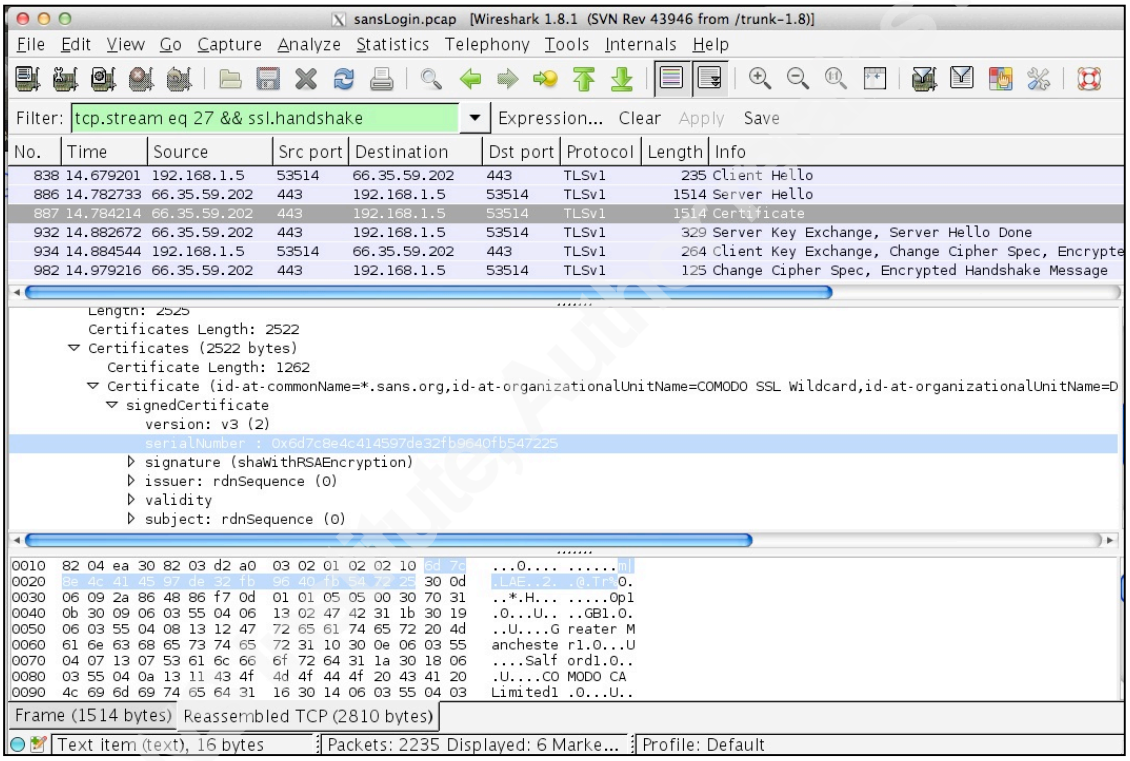

Figure 19. Server certificate

5.5.4. Server to client handshake messages

The next packet contains two of the TLS handshake messages. They are *ServerKeyExchange* and *ServerHelloDone*. The cipher chosen by the server in this example session will use the DH key generation method. The *ServerKeyExchange* message contains numeric components that will be used for the generation of the session keys in the DH algorithm (Rescorla, 2001, p. 35). The *ServerHelloDone* message is a one byte code to indicate that the server is finished with its side of the session negotiation (Figure 20).

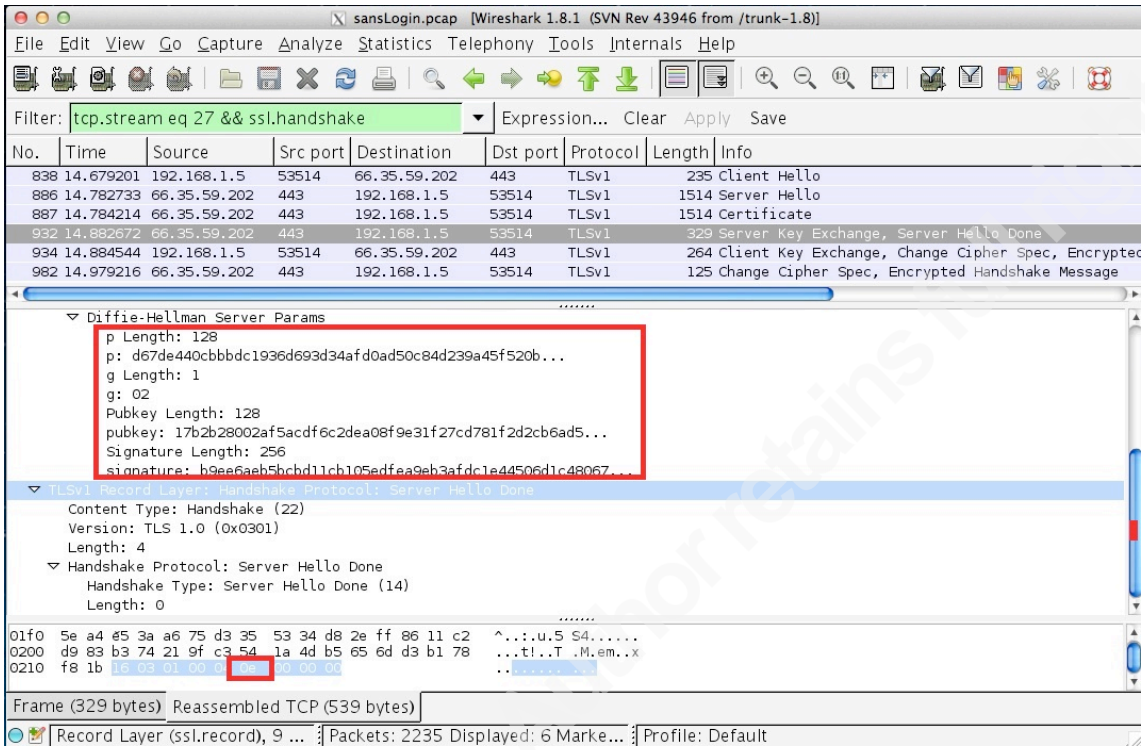

Figure 20. Diffie-Hellman public values from server in *ServerKeyExchange* message and the *ServerHelloDone* Message

5.5.5. Client to server final handshake messages

The client then responds similarly, sending a *ClientKeyExchange*, a *ChangeCipherSpec* and a *Finished* message, however, the Finished message is still encrypted so Wireshark labels this "*Encrypted Handshake Message*". When Wireshark is pointed at the SSL key log file that stores the session keys and is able to perform decryption, it will label this message *Finished*.

Since the client and server have decided to use the DH key generation method, the client sends DH parameters in the *ClientKeyExchange* message just as the server did above. This is followed by the *ChangeCipherSpec* message, a one byte code stating that the all further transmissions will have an encrypted payload. Lastly, the client completes its side of the TLS handshake by using the newly generated session key to encrypt a hash of the handshake components, creating the *Finished* Message. The client and server can

each derive session keys at this point because they have exchanged all the necessary information, including the DH parameters.

5.5.6. Server to client final handshake messages

The final step in the handshake is for the server to also send a *ChangeCipherSpec*, followed by an encrypted hash of the handshake components so that the client can also verify the integrity of the process. If both the client and server determine that the other's *Encrypted Handshake Message* is correct the connection will remain open and data will be sent in encrypted *ApplicationData* records. If a problem is detected on either side during the verification then the session is terminated and an alert record will be sent with a message to indicate the problem.

At this point, removing just the "ssl.handshake" portion of the Wireshark display filter will show all packets in the stream including the *ApplicationData* Packets. These are all encrypted, so it is impossible to determine what data is being exchanged.

Wireshark needs to have access to the key log file in order to decrypt the *Finished* messages from the handshake and the *ApplicationData* records. Navigate to the Wireshark Preferences page using either Edit → Preferences or the key combination Shift-Ctrl-P. Expand "Protocols" in the leftmost column and select SSL.

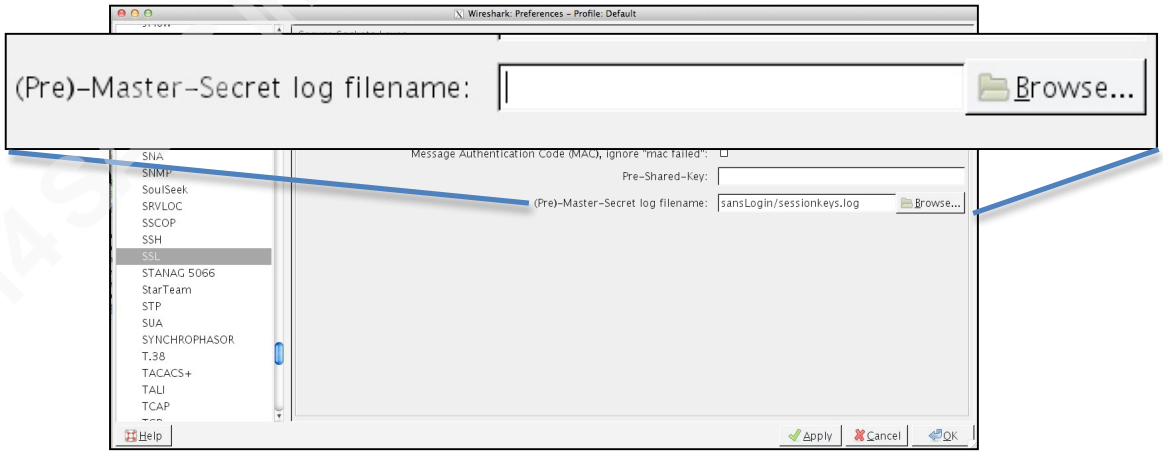

Figure 21. Configuring Wireshark to use saved session keys

Enter the path to the log file as defined by the SSLKEYLOGFILE variable, which Wireshark calls the "(Pre)-Master-Secret log file" and click *Apply* and *OK* (Figure 21).

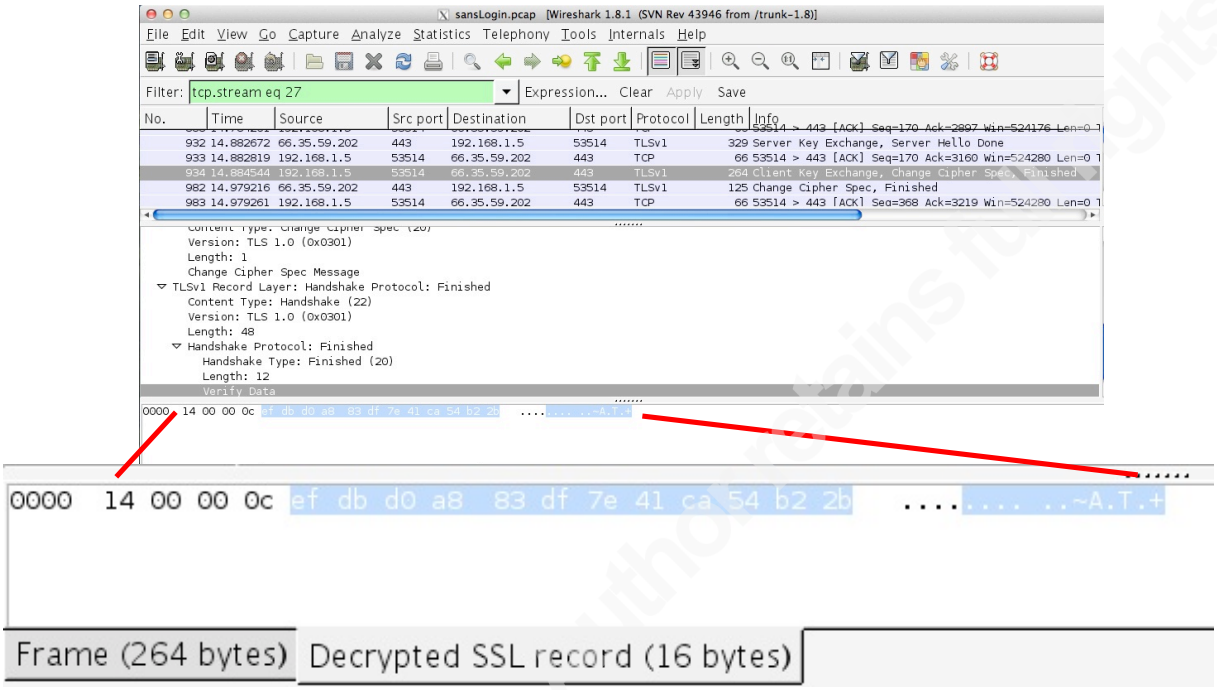

Figure 22. The decrypted tab in Wireshark

The decrypted *Finished* message shown in figure 22 was called "*Encrypted Handshake Message*" by Wireshark before it had access to the key log file. Now that Wireshark can decrypt the data, it can read all the header fields, in this case, the handshake type field is 0x14 defining this as a *Finished* message. The *Finished* message content is a hash or digest over the master secret and all handshake messages. Expand the record in the middle pane to view the decrypted *Finished* message from the client and compare it to the decrypted *Finished* message from the server. The two messages are different. This is because these messages contain a hash of the concatenation of all handshake messages thus far so the server's *Finished* message will be hashed over a slightly different data set than the client's *Finished* message. The *Finished* message is encrypted with the newly generated session key. In the Decrypted tab in Wireshark, is the handshake message type of 0x14 (20 decimal), followed by 3 bytes for the message length field. This is followed by 0xC (12 decimal) bytes of data that represents the hash used to verify the successful completion of the handshake. Each side makes a quick calculation of what it believes the hash should be and compares it to what the other sent.

If those values do not match the session will not proceed. To understand how this protects the integrity of the session, suppose that an attacker was able to intercept and change some part of the handshake that the client sent to the server (remember, most of the handshake is unencrypted). During the handshake verification process, the handshake data used to calculate the hash for the *Finished* message consists of the data that it sent to the server and the data that it received from the server. When the server verifies this hash, it does its own computation of the fields that it received from the client as well as the fields that it sent to the client. If an attacker had changed any of these fields while in transit then those datasets would no longer match and the hash of each would be different, causing the session to end in error (Rescorla, 2001, p. 81).

5.6. Dissecting the Application Data

The data that follows the handshake is the *ApplicationData*. Adding "and ssl.segments" to the Wireshark display filter will display only the decrypted packets from that stream. Because it is decrypted, Wireshark can apply protocol decoders and parse the contents that are now identified and labeled as HTTP and highlighted with the default color of green. Figure 23 shows the unencrypted *ApplicationData* packets and figure 24 shows the properly decrypted and decoded HTTP data.

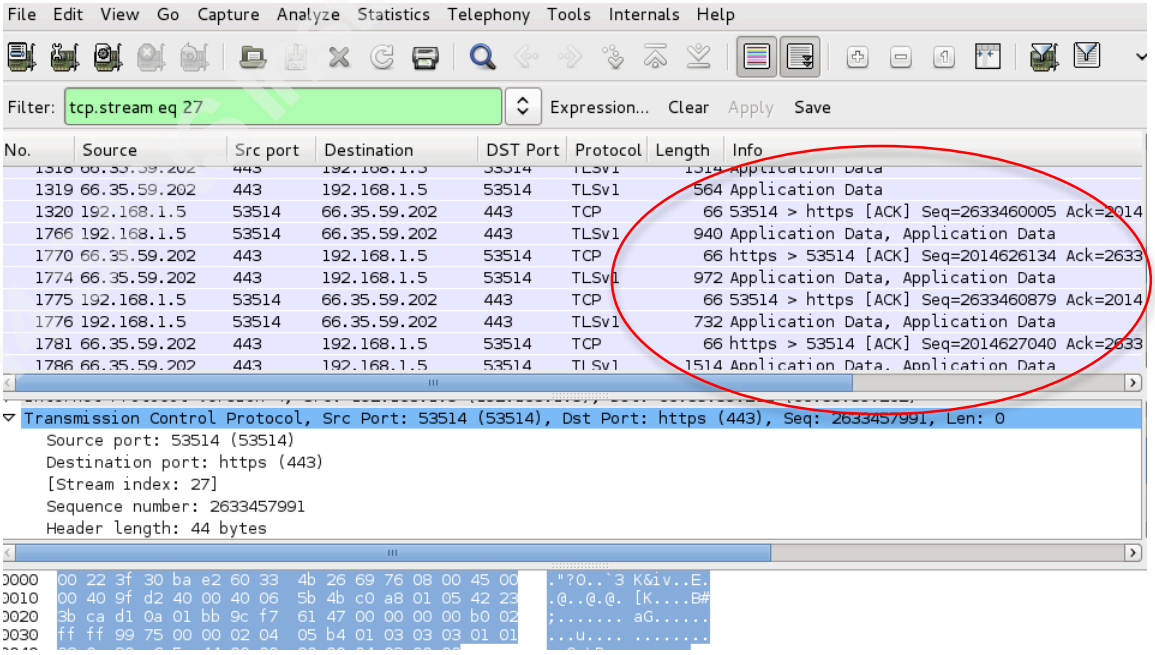

Figure 23. Application Data records before decryption

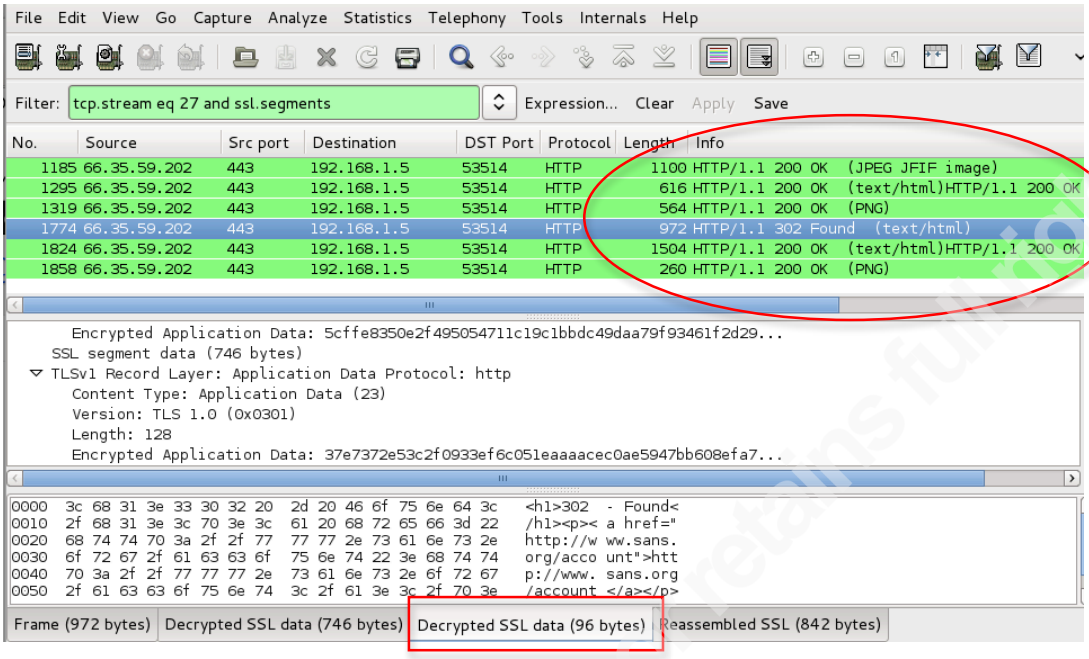

Figure 24. Application Data records after decryption

Note the tabs at the bottom in Figure 24. Because TLS breaks up the data for encryption at the application layer and then TCP may have to further break up encrypted streams at the transport layer, there may be several tabs for a given packet where data has been decrypted and/or reassembled. Selecting the "Decrypted SSL data" tab will show the decrypted data in the bottom pane of Wireshark. Alternatively, by right-clicking on any HTTP, SSL or TLS packet in the top or middle pane and selecting *Follow SSL stream* and assuming Wireshark has access to the key log file, Wireshark will pop up a box containing the TLS Application data without the packet headers and decrypted.

Wireshark is displaying data that has already been decrypted in a browser in this example. What is Wireshark showing us that we could not see with the browser? Any code, including scripts or values that was sent to the browser encrypted and used by the browser to display pages has been decrypted by the browser but it has not necessarily all been displayed. The browser does, however, have a "View Source" option that will show the raw HTML revealing some of these scripts and values. What is not normally displayed is the data being sent back from the browser to the server, that is, unless we have set up a proxy such as Fiddler or Paros. This data may include things like cookies, hidden form fields or data the user has entered into a web form such as usernames and

passwords. Often this is data used by the web application to help with session tracking and as such is not normally displayed to the user.

Most of us know that usernames and passwords should never be sent in cleartext because tools like Wireshark can capture that traffic and compromise those credentials. Many websites use TLS to encrypt a session so that sensitive credentials are protected. For example, a server login session normally starts by setting up a TLS connection with the server after which the server sends the browser a login page. The user enters his username and password and it is encrypted with the established session keys and sent back to the server. For a session using the RSA key exchange method, if an attacker was intercepting that traffic, she would also need the server's private key in order to discover the pre-master secret. The pre-master secret is required to generate the master secret that, in turn, is used to generate the session keys. The session keys ultimately decrypt the captured data. For a session using the DH key generation method, an attacker intercepting traffic would also need the DH private components or she would not have enough information to generate the master secret and then the session keys. Since those private components do not cross the network, such an attack is unlikely to succeed. In contrast, Wireshark is able to decrypt the session because it has access to the pre-master and master secrets that the browser saved and it has captured the client and server random values from the *ClientHello* and *ServerHello* messages, or the DH parameters from the *ClientKeyExchange* and *ServerKeyExchange* messages.

We already know our username and password so this may not seem very interesting, however, this method can be helpful when troubleshooting TLS connection problems or debugging web applications. It also provides a way to view exactly what is sent back and forth during an encrypted session that a browser does not normally display. From a defensive standpoint, this is not a bad method to "look under the hood" and get a better view of how an application works in order to decide whether or not to perform transactions with a given online entity, for example, a bank. This is not the only method to view this decrypted data, but it provides another way. In addition, it helps to understand how the TLS protocol works by breaking down the steps in an organized fashion. To further enhance an understanding of TLS, Wireshark offers the option to turn

on debugging mode. With this enabled, every step that Wireshark takes in the decryption process is documented in a debug file.

5.6.1. Help from Wireshark's debug feature

The final part of this examination of TLS traffic will demonstrate the debug capabilities that are built into Wireshark using the same trace file, sansLogin.pcap with logged pre-master and master secrets in the sessionkeys.log file. The first step in this process is to create an empty debug file so that Wireshark can write the details of its actions when reading in the trace. The file can be located anywhere but it is important to remember that all decrypted data will be written to this file so if the data is sensitive, it should be properly protected.

```
$ touch /tmp/sansLogin.debug.txt
```

This can be set up on Wireshark preferences page using either Edit → Preferences or the key combination Shift-Ctrl-P. Expand "Protocols" in the leftmost column and select SSL.

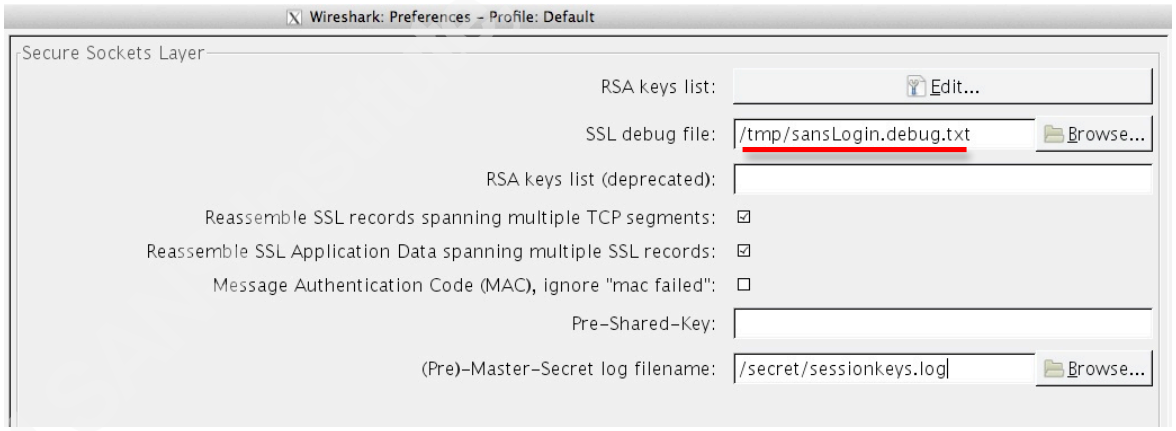

Figure 25. Configuring the debug file in Wireshark

Enter the path to the newly created SSL debug file and click *Apply* (Figure 25). When the trace file is opened in Wireshark, the debug file will be populated with verbose information about the steps taken by Wireshark to read in the file, reassemble packets and decrypt the data. If the trace file is already open, the debug file will be populated immediately after clicking *Apply* or *OK* to finish the dialogue. The debug file will continue to be populated every time Wireshark performs any action. This file can grow very large, very quickly. For our purposes here, loading the file once to populate the

debug file will provide adequate data for analysis. To stop the debugging, remove the entry for the debug file in Wireshark's preferences and click *Apply* or *OK*.

When Wireshark encounters an encrypted packet and it has been given a filename where pre-master and master secrets have been stored it will search through the file, looking for the correct key. There are entries that start with RSA and entries that start with CLIENT-RANDOM. Wireshark takes a brute force approach and checks each line to see if the index field matches the corresponding field in the packet. For example, if a line in the key log file starts with RSA it will compare the next 16 bytes from that line in the file to the first 16 bytes of the encrypted pre-master secret field from the *ClientKeyExchange* message. If the client and server have negotiated a DH key generation method then that particular session will not use an encrypted pre-master secret so Wireshark will move to the next line in the file. When an entry in the key log file starts with CLIENT_RANDOM Wireshark will compare the next 96 bytes from that line in the file with the Random field in the *ClientHello* message of the TLS handshake for that session. This process continues until a match is found or the end of the file is reached. The debug file will have an entry for each line it checks, as well as, whether or not it found a match. To see which cipher suite was chosen for a particular TLS session in Wireshark enter the following display filter:

```
ssl.handshake.ciphersuite && tcp.port == xxx
```

(xxx = the client's TCP port number for that particular session)

Two packets are returned for this display filter, the first is the *ClientHello* message, which is the client's initial proposal to the server stating which cipher's it supports and the second is the *ServerHello* message stating the cipher that it has chosen for the session.

Each cipher suite consists of four elements: 1) Authentication method 2) Key exchange method 3) Encryption algorithm and 4) Hash algorithm.

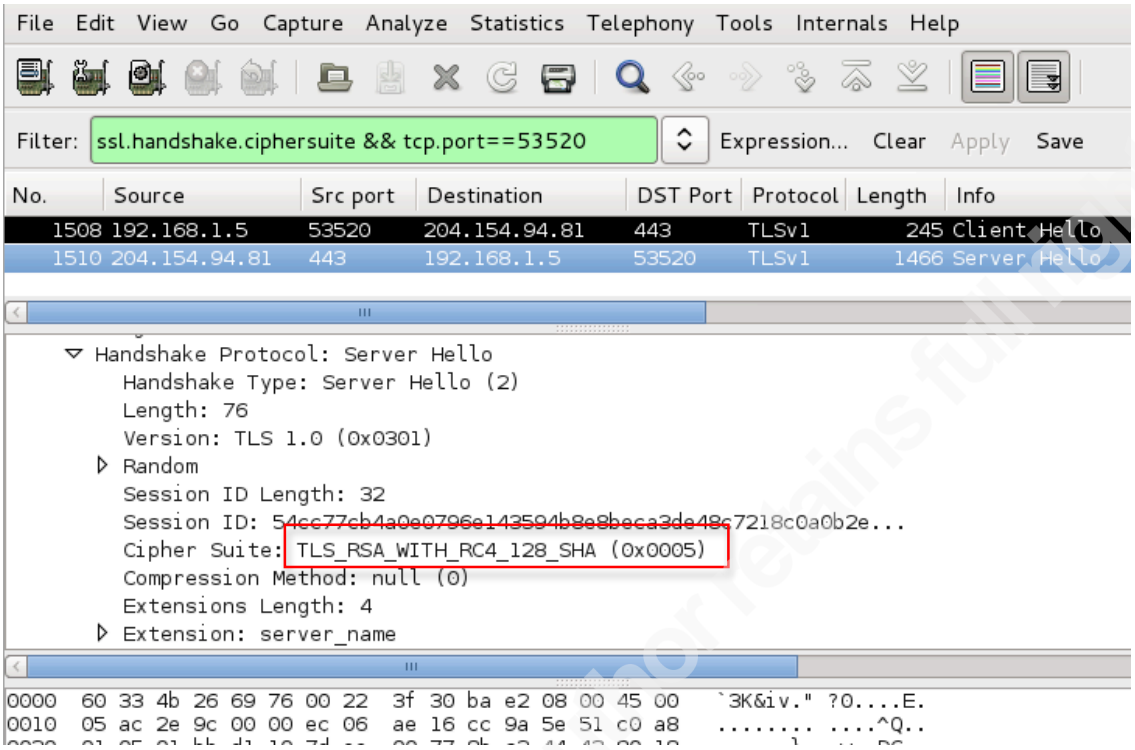

Figure 26. Chosen cipher suite for a TLS session

Figure 26 shows a *ServerHello* message with the chosen cipher suite highlighted. The server sends the value 0x0005 to the client to identify the cipher suite and Wireshark translates that according to the assignments at IANA (Rescorla, 2013).

For this cipher, TLS_RSA_WITH_RC4_128_SHA, the TLS session will use RSA Key Exchange, RSA Authentication, RC4 Encryption using a 128 bit key length and SHA1 hash algorithm (Rescorla, 2001, p. 74).

```
# SSL/TLS secrets log file, generated by NSS
RSA 2273f99ea182a552 0301121d681ed9412175d564ba404f2ffb9cb99f8c9df7af006fdab054520b5cc8b491
CLIENT_RANDOM 51cca88386fca93031d2b4d3ded55a8a953c74d5da1938840c26276eeaef5753 3e2204d13185
RSA 47222a2a2f69baf7 0301a080d05fbd620930e3e0d4352db9b3fc9c25a64cb5902ea5652bf4a2646fc5ee9a
CLIENT_RANDOM 51cca88357857a9d540b6b185d5db9466de05056e52ee5b36f40d697647b2c48 1d7d6bf0c062
CLIENT_RANDOM 51cca885df6d530d56cc035b73d449b18f78be973014ce9739300eac7b074dd6 3e2204d13185
CLIENT_RANDOM 51cca885d403811039b25ad8ee8b8f9f934aa0ce889a4bc7503c374afa040865 9c54ae691e84
CLIENT_RANDOM 51cca885da15d694946fb795e974fb84e4c786ad5ad52066f14e3414beaf581b b72a47c8c098
CLIENT_RANDOM 51cca885b06e0651d6630a5b047635d7491974eef6e4be811c2d493d510db1a1 3e2204d13185
CLIENT_RANDOM 51cca88574cee70b302dbf54b77b03bc45abd565e2bd6ceade0dff5abb16eba7 3e2204d13185
CLIENT_RANDOM 51cca88517e544deb14e2a06637a8d6bd41ab8bf23a59ffde18b14b0b09c08d7 3e2204d13185
CLIENT_RANDOM 51cca885a3553c93955556741e439ee359f7020e2e0de63d8376d236034d8a35 3e2204d13185
CLIENT_RANDOM 51cca886a6229b929199ef25dd6f046e859fd9937c7d549bf60bc2932c7011b5 78c68b869cb6
CLIENT_RANDOM 51cca88713e45b212ec775ee506ca9888254875a1d5800fe2a2caab82d9d72d8 e2f608c44b09
CLIENT_RANDOM 51cca88a8c6a34ddcfe56e9a0ea796b334613cf4ed0425935776b5d7aca442cf 7604597b988f
CLIENT_RANDOM 51cca88b4b69945c7d627f1a6cc8989ec3b4b7499119ffe50796c9fcf6f607a8 7604597b988f
```

Figure 27. Sample from a key log file showing entries for RSA and Diffie-Hellman Key Exchange methods.

The format of the TLS key log file shown in Figure 27 is documented at the Mozilla Developer's Network website (Combs, 2012). See Appendix B for a more thorough examination of the contents a Wireshark debug file.

6. Securing packet capture and debug files

Since TLS is often used for sensitive transactions, it naturally follows that when capturing this traffic along with the keys used to decrypt it, security of these files is paramount. Of primary concern is the key log file, which should be readable only to the owner of the file. In addition, when Wireshark writes to a debug file it writes TLS session keys, ciphertext and the corresponding decrypted plaintext to the debug file as described in the previous section and so may also require access restrictions. Securing these files is documented in Appendix A.

7. Conclusion

This paper has outlined how to capture, decrypt and analyze TLS sessions using the built in capabilities of Wireshark. The TLS handshake was examined in depth both using Wireshark display filters as well as the debug capability that Wireshark offers. Based on experiments done for this paper, the browsers that currently will support export to a key log file for this type of operation are Firefox versions 19 -22 and Chrome versions 24 - 30. This type of analysis is useful for those who wish to understand the TLS protocol in greater depth. Web application developers may find these tools useful when troubleshooting an application that uses TLS. Penetration testers may find value in examining all the data transferred back and forth during an encrypted session with targets of their test without using a proxy.

8. References

AlFardon, N., & Paterson, K. (2013, February 4). *Lucky thirteen: Breaking the TLS and DTLS record protocols*. Retrieved July 5, 2013 from http://www.isg.rhul.ac.uk/tls/Lucky13.html

Barker, E., Branstad, D., Chokhani, S., & Smid, M. U.S. Department of Commerce, NIST. (2010). *A framework for designing cryptographic key management systems* (800-130). Retrieved from website: http://csrc.nist.gov/publications/drafts/800-130/draft-sp800-130_june2010.pdf

Bernstein, D. J. (2013, March 12). *Failures of secret key cryptography*. Retrieved June 18, 2013 from http://cr.yp.to/talks/2013.03.12/slides.pdf

Combs, G. (2012, June 4). *NSS key log format*. Retrieved July 21, 2013 from https://developer.mozilla.org/en-US/docs/NSS_Key_Log_Format

Combs, G. (2013, July 4). *Tshark - dump and analyze network traffic*. Retrieved July 21, 2013 from http://www.wireshark.org/docs/man-pages/tshark.html

Constantin, L. (2012, September 13). *'CRIME' attack abuses SSL/TLS data compression feature to hijack HTTPS sessions*. Retrieved July 21, 2013 from http://www.pcworld.com/article/262307/crime_attack_abuses_ssltls_data_compression_feature_to_hijack_https_sessions.html

Dierks, T. & Allen, C. (1999) *The TLS protocol version 1.0.* (Request for comments: 2246) Retrieved May 15, 2013 from IETF: http://www.ietf.org/rfc/rfc2246.txt

Dierks, T. & Rescorla, E. (2006) *The transport layer security (TLS) protocol version 1.1.* (Request for comments: 4346) Retrieved May 13, 2013 from IETF: http://www.ietf.org/rfc/rfc4346

Dierks, T. & Rescorla, E. (2008) *The transport layer security (TLS) protocol version 1.2.* (Request for comments: 5246) Retrieved May 15, 2013 from IETF: http://tools.ietf.org/html/rfc5246

Dougherty, C. (2011, September 27). *SSL 3.0 and TLS 1.0 allow chosen plaintext attack in CBC modes*. Retrieved June 18, 2013 from http://www.kb.cert.org/vuls/id/864643

Eckersley, P. (2011, October 25). *How secure is HTTPS today? How often is it attacked?*. Retrieved from https://www.eff.org/deeplinks/2011/10/how-secure-https-today

Housley, R., Ford, W., Polk, W., & Solo , D. (1999, January). *Internet X.509 public key infrastructure certificate and CRL profile*. Retrieved from http://www.ietf.org/rfc/rfc2459.txt

Marlinspike, M. (2011, April 11). *SSL and the future of authenticity*. Retrieved from http://www.thoughtcrime.org/blog/ssl-and-the-future-of-authenticity/

Mozilla Developer Network. (n.d.). *Network security services*. Retrieved July 27, 2013 from https://developer.mozilla.org/en-US/docs/NSS

Rescorla, E. (1999) *Diffie-Hellman key agreement method.* (Request for comments: 2631) Retrieved May 15, 2013 from IETF: http://www.ietf.org/rfc/rfc2631.txt

Rescorla, E. (2001). *SSL and TLS: Designing and building secure systems*. (1st ed.). Montreal, Canada: Addison-Wesley Professional.

Rescorla, E. (2011, September 23). *Security impact of the Rizzo/Duong CBC "BEAST" attack*. Retrieved June 28, 2013 from http://www.educatedguesswork.org/2011/09/security_impact_of_the_rizzodu.html

Rescorla, E. (2013, May 8). *Transport layer security (TLS) parameters*. Retrieved July 21, 2013 from http://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

Ristić, I. (2013, May 14). *SSL/TLS deployment best practices*. Retrieved June 23, 2013 from https://www.ssllabs.com/downloads/SSL_TLS_Deployment_Best_Practices_1.2.pdf

Schneier, B. (1996). *Applied cryptography: protocols, algorithms, and source code in C.* 2nd ed. New York: Wiley. Retrieved from Safari: http://proquest.safaribooksonline.com

Tcpdump & Libpcap. (2013, July 10). Retrieved July 19, 2013 from http://www.tcpdump.org/

Viega, J., Messier, M., & Pravir, C. (2009). *Network security with openssl*. (Kindle Edition ed.). Reilly Media. Retrieved from http://www.amazon.com

Winpcap: Windows packet capture library. (2013, July 10). Retrieved July 24, 2013 from http://www.winpcap.org/

Wireshark Protocol Analyzer. (2013). Retrieved June 14, 2013 from http://www.wireshark.org

Appendix A
Decrypt TLS Traffic Using Wireshark: Step-by-Step

1. Create a key log file and set environment variable

Add the following to either ~/.profile, ~/.bash_profile, or ~/.bashrc

```
export  SSLKEYLOGFILE=/protected/folder/my_session_keys.log
```

Create the empty log file

```
$ touch /protected/folder/my_session_keys.log
$ chmod  700 /protected/folder/my_session_keys.log
```

Check the variable

```
$ ls –l  $SSLKEYLOGFILE
```

2. Capture traffic

2.1.1. Tcpdump or tshark

Figure A-1 shows how to capture traffic using tcpdump and tshark respectively. The command will capture all traffic on interface "eth0" (-i), IP addresses will not be resolved to domain names (-n) and the packets will be written (-w) to a file called packets.pcap in the current directory.

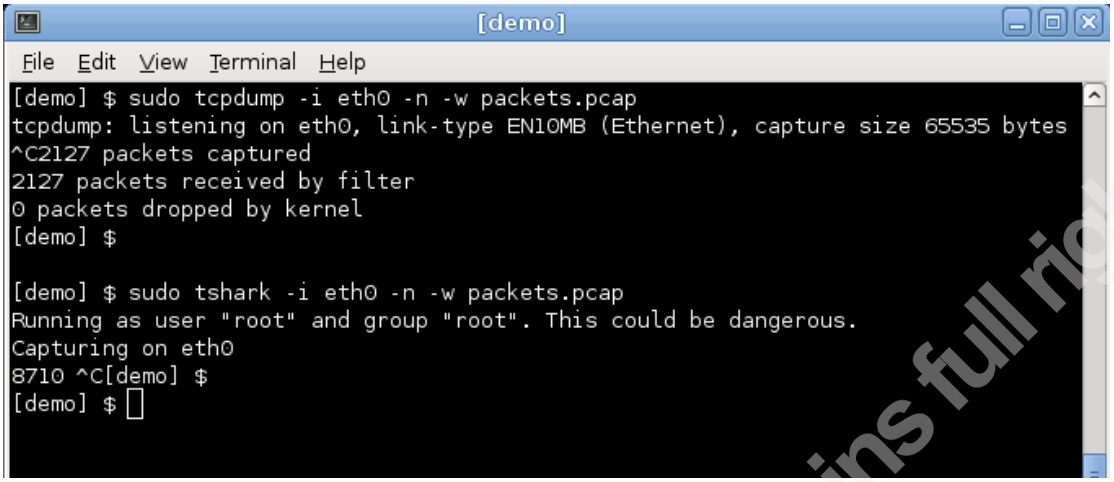

Figure A-1. Capturing network traffic with tcpdump and tshark

2.1.2. Wireshark

Select "Capture → Options…" to view the capture dialogue. Select the network interface and any other desired options, such as, unchecking the name resolution boxes (figure A-2). Using "promiscuous mode" on all interfaces will capture all traffic that an interface can "see". Typically, this includes all traffic to and from that interface and all broadcast traffic. This may vary if an interface is connected to a spanned switch port.

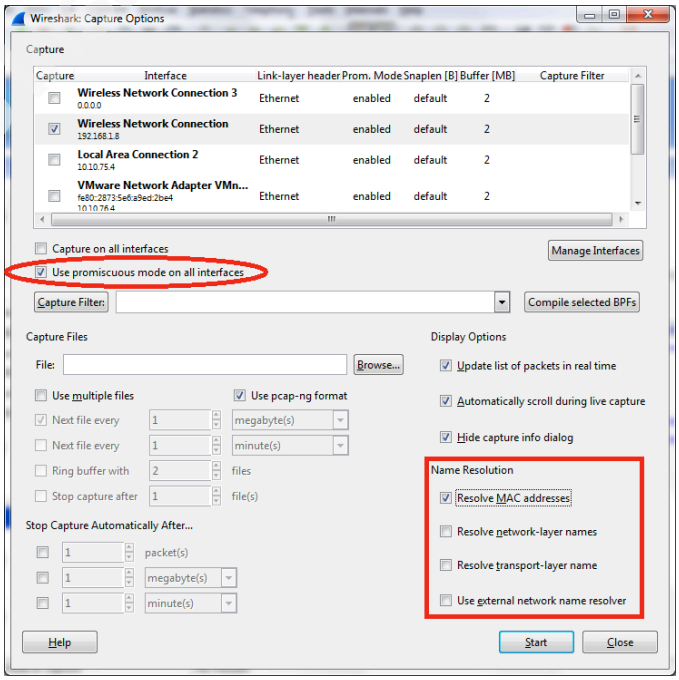

Figure A-2. Wireshark capture options

To end the capture, click on the *Stop capture* button from the Wireshark toolbar (figure A-3).

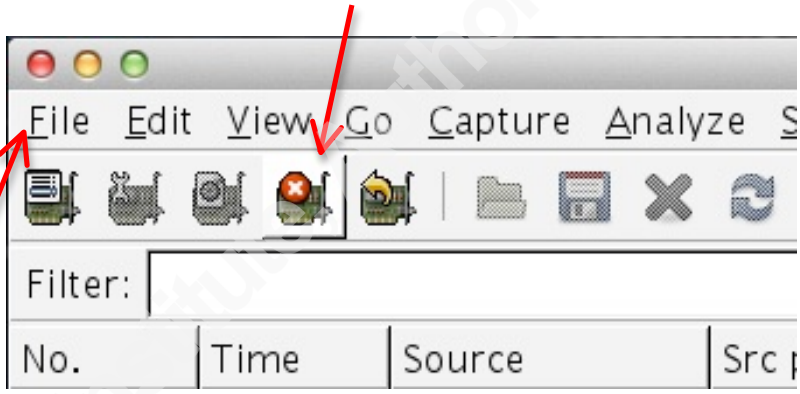

Figure A-3. Wireshark stop capture and "Save As…" options

Save the captured packets to a file by selecting "File" from the Wireshark toolbar and then "Save As …" to select a location.

2.2. Open a packet capture file in Wireshark for analysis

Open the Wireshark application and click on "File → Open…" from the toolbar. Navigate to the desired file.

3. TLS Analysis steps

3.1.1. Filter on TLS packets

From the Wireshark toolbar select Analyze → Conversations and select the TCP tab.

Select an encrypted conversation form the populated list. A server port of 443 can most likely identify encrypted conversations (figure A-4).

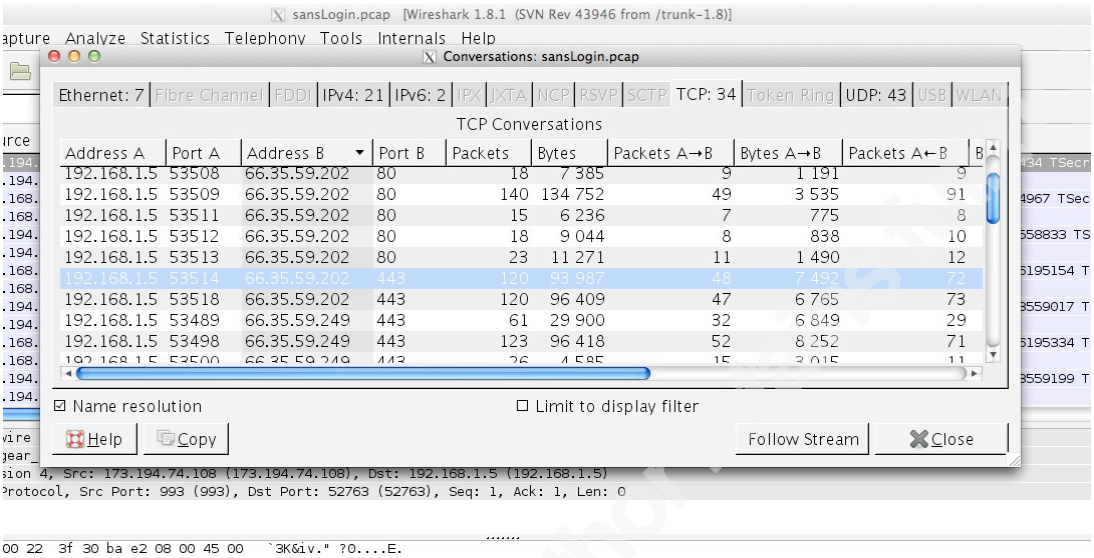

Figure A-4. Conversation list displayed by Wireshark

Highlight a conversation and click on *Follow Stream* (figure A-5).

Figure A-5. TCP stream display in Wireshark

Close "Follow TCP Stream" window to view just that conversation that Wireshark has now filtered on (figure A-6).

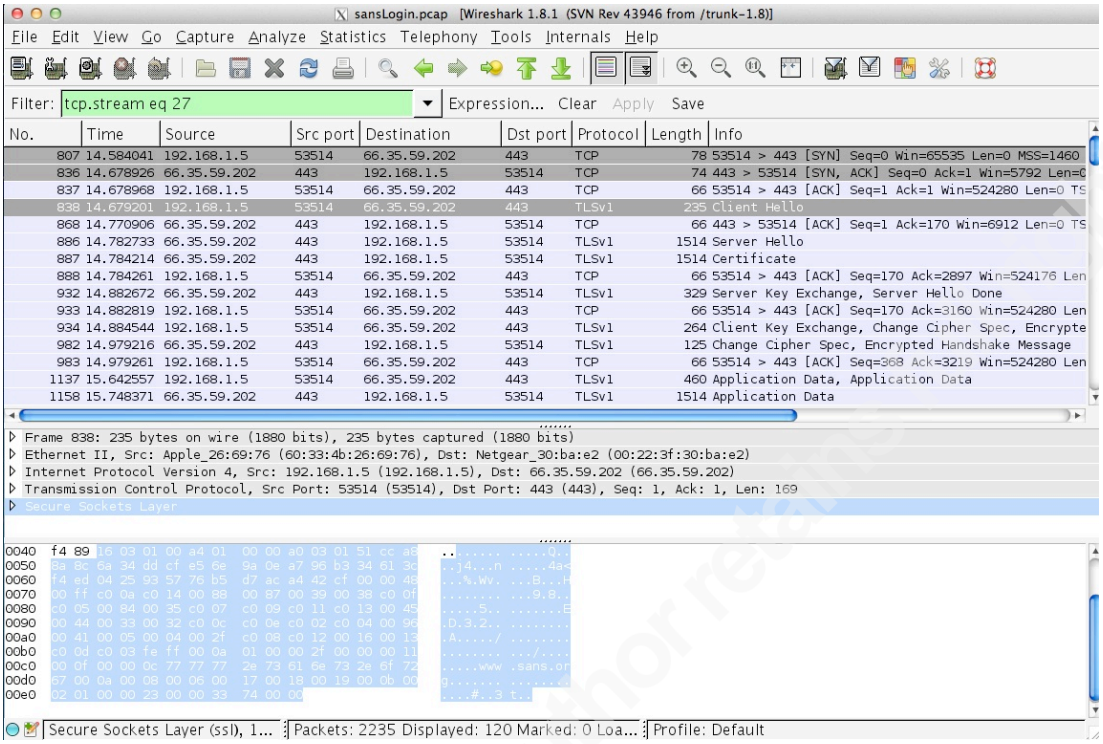

Figure A-6. Filtering on a single TCP stream in Wireshark

3.1.2. Filter on the TLS handshake

To view just the TLS handshake from this conversation, add "`&& ssl.handshake`" to the display filter (figure A-7).

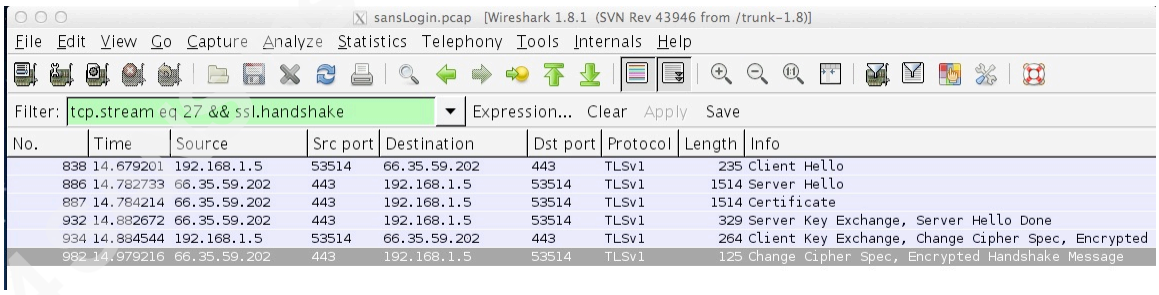

Figure A-7. Isolate an SSL handshake in Wireshark

Click on the *ServerHello* handshake record in the top pane in Wireshark (figure A-8).

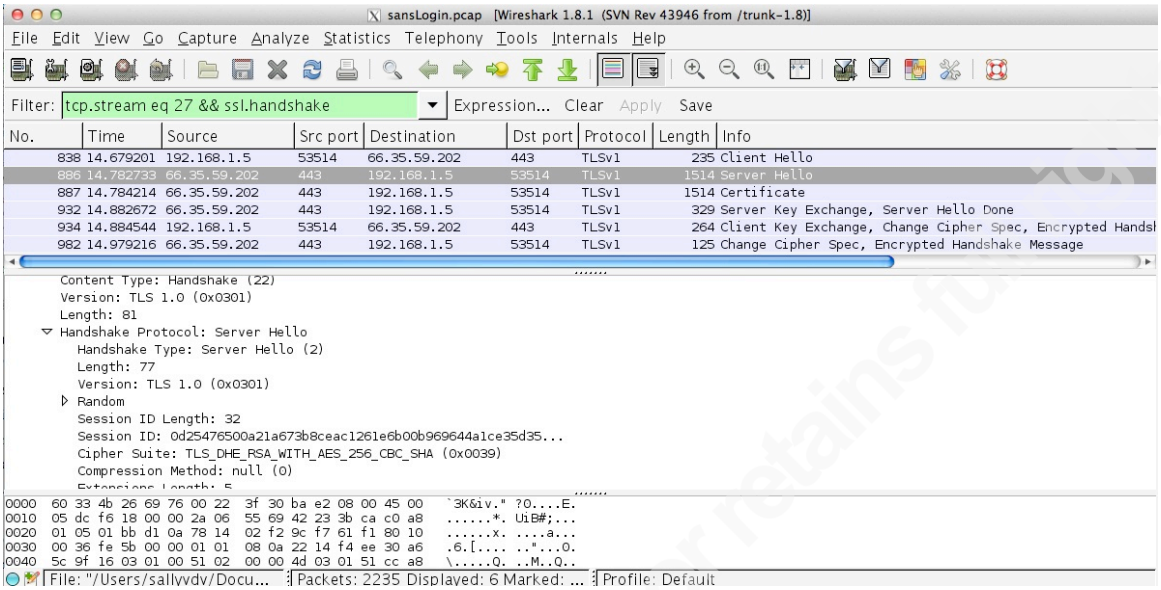

Figure A-8. *ServerHello* message

Expand any sections in the middle pane to view how Wireshark identifies the various components.

3.1.3. TLS Decryption

To determine the location of the key log file type one of the following commands at the command line

On Linux or OS X:

```
$ echo $SSLKEYLOGFILE
```

On Windows:

```
C:\> echo  %SSLKEYLOGFILE%
```

Configure Wireshark to use the session keys collected by the browser in order to decrypt the data. From the toolbar select "Edit → Preferences → Protocols → SSL" (figure A-9).

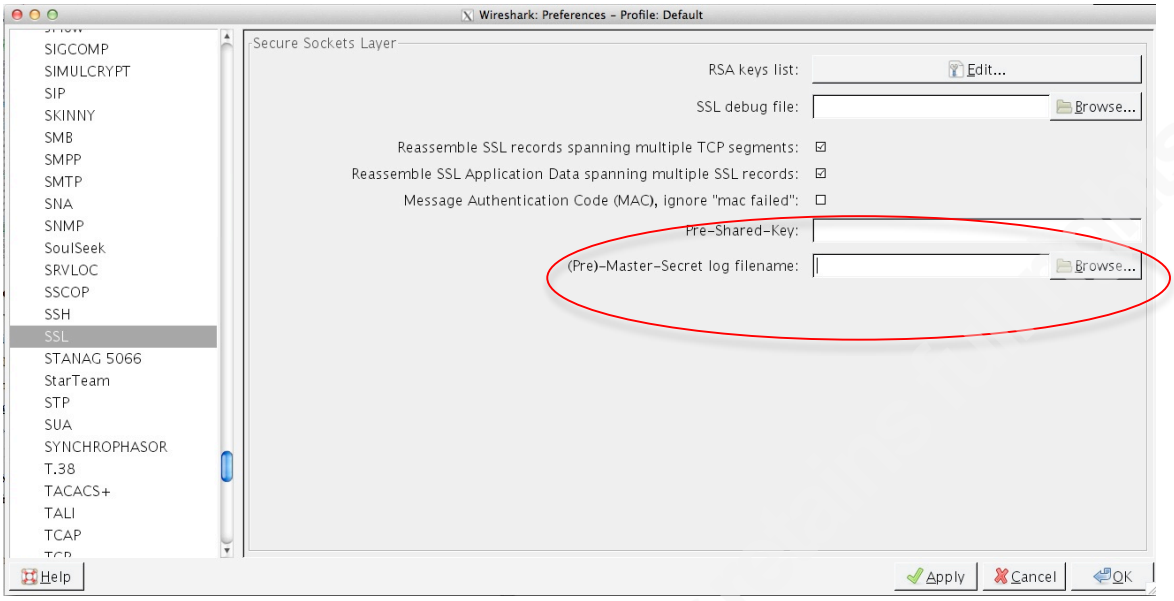

Figure A-9. Wireshark's SSL preferences dialogue box

In the Secure Sockets Layer configuration box, browse to the file containing the session keys and click *OK* (figure A-10).

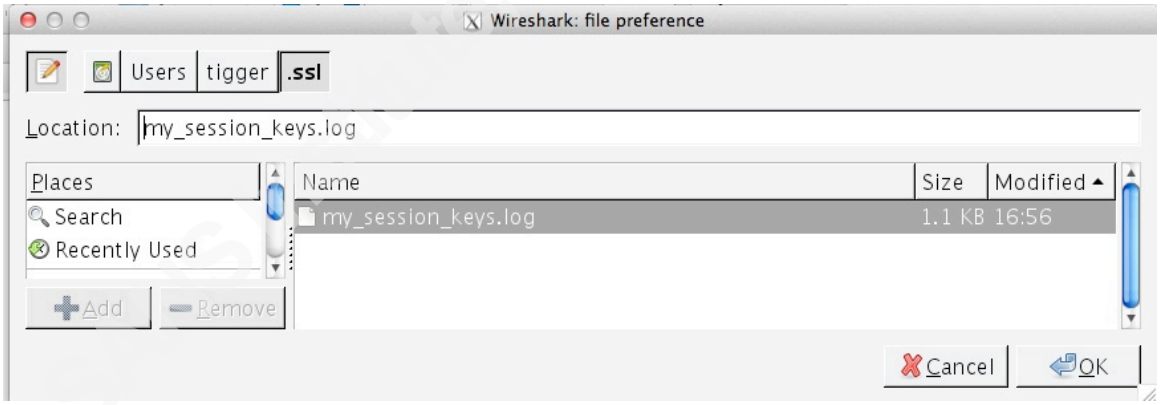

Figure A-10. Directing Wireshark to the key log file

Enter the display filter "`tcp.port==443 && http`" to show the packets that Wireshark was able to decrypt (figure A-11).

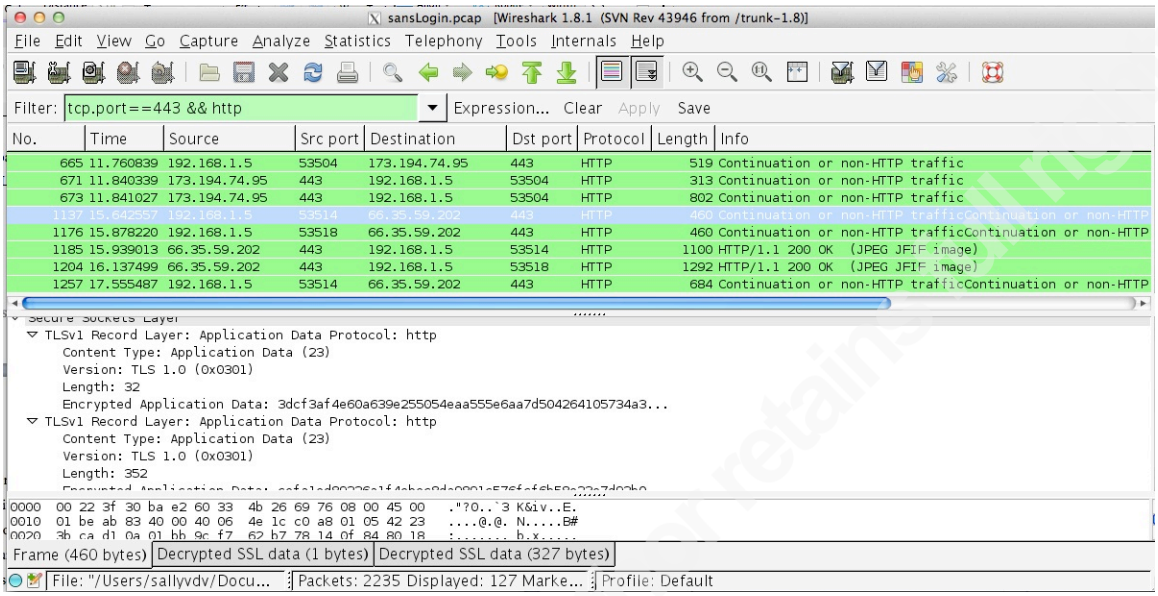

Figure A-11. Display decrypted HTTPS packets

3.1.4. Use the SSL debug feature in Wireshark

Create an empty debug file for Wireshark to write its detailed decryption operations.

```
$ touch /path/to/protected/folder/my_debug_file.log
$ chmod  700 /path/to/protected/folder/my_debug_file.log
```

Configure Wireshark to use the debug file. From the toolbar select "Edit → Preferences → Protocols → SSL" (figure A-12).

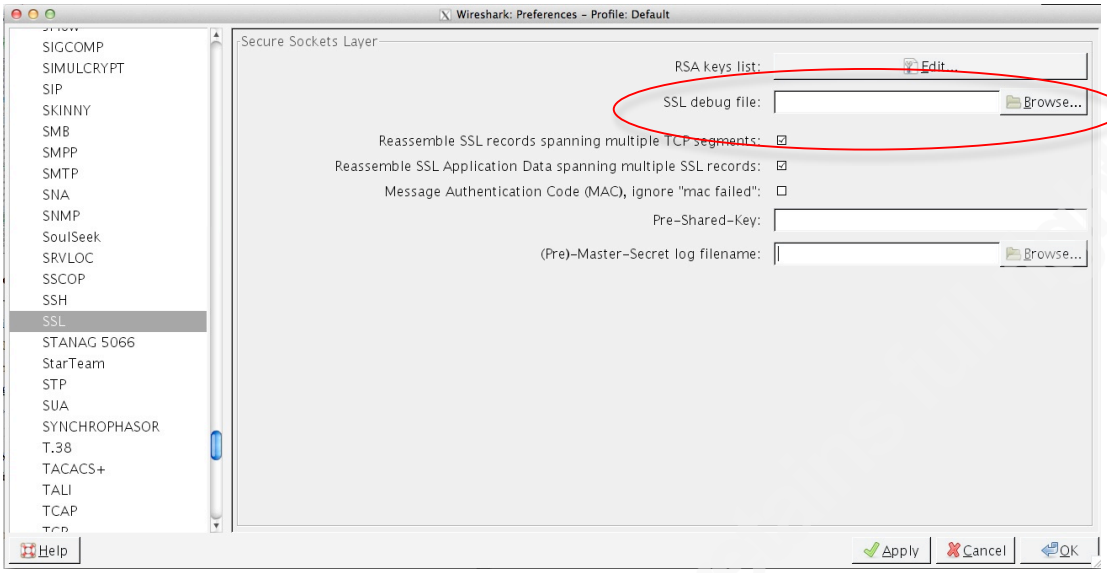

Figure A-12. Wireshark's SSL preferences dialogue box

In the Secure Sockets Layer configuration box, browse to the newly created debug file and click *OK* (figure A-13).

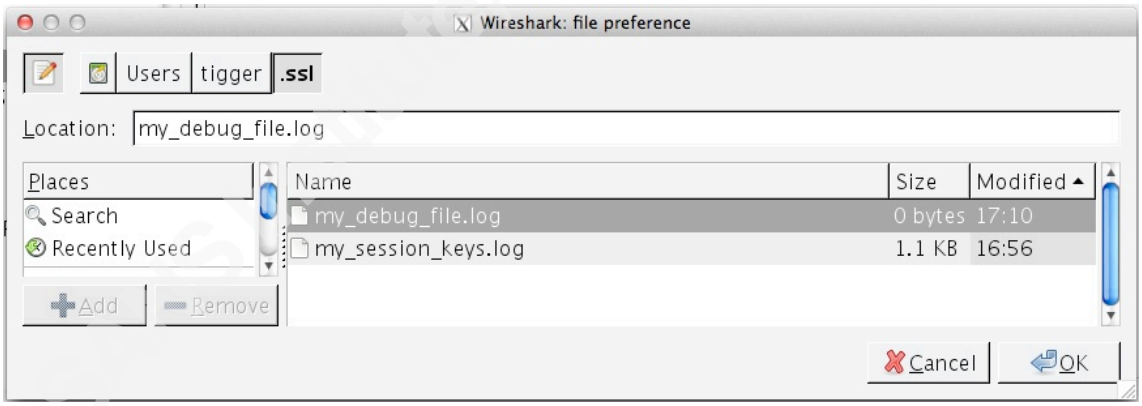

Figure A-13. Directing Wireshark to the debug file

Reload the capture file to populate the debug file. The reload button is on the main toolbar in Wireshark (figure A-14).

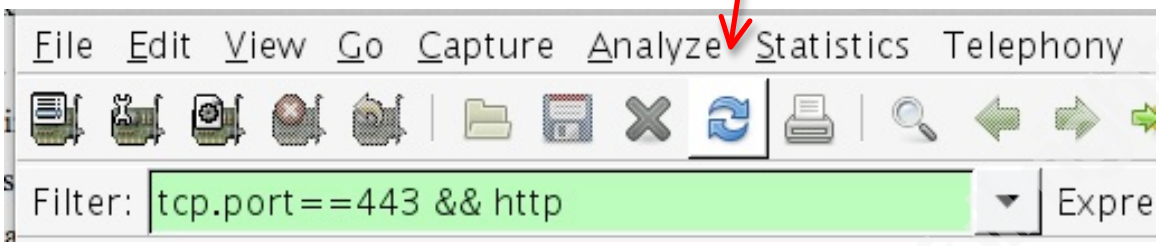

Figure A-14. The reload packet capture button

Return to the SSL Preferences and remove the entry for the debug file to prevent this file from growing very large.

View the debug file with any text editor to see verbose output from Wireshark regarding its decryption operations. Appendix B shows an example debug file.

Appendix B
The Wireshark Debug File

Below is a section of the debug file (with many lines snipped for brevity) that illustrates Wireshark's process for finding the correct entry in the key log file and computing session keys. This is an abbreviated entry for a single packet. Wireshark creates similar entries for each packet that is part of a TLS conversation.

```
dissect_ssl enter frame #208 (first time)
conversation = 0x10A573110, ssl_session = 0x10A573CF0
record: offset = 0, reported_length_remaining = 314
dissect_ssl3_record: content_type 22 Handshake
  <snip>
dissect_ssl3_handshake iteration 1 type 16 offset 5 length 258 bytes,
remaining 267
trying to use SSL key log in /sansLogin/sessionkeys.log
checking keylog line: # SSL/TLS secrets log file, generated by NSS
line does not match
checking keylog line: RSA 2273f99ea182a552
0301121d681ed9412175d564ba404f2ffb9cb99f8c9df7af006fdab05452
0b5cc8b491c718dd041d0cff9e97565b19a7
line does not match encrypted pre-master secret    line does not match
checking keylog line: CLIENT_RANDOM
51cca88386fca93031d2b4d3ded55a8a953c74d5da1938840c26276eeae
f57533e2204d1318530e0e1faade7c8bde696528a4b5373f0fa53ef59bc7
5ea6ad55f7b10904810876fdb740645ff067c588b
line does not match client random
line does not match
checking keylog line: RSA 47222a2a2f69baf7
0301a080d05fbd620930e3e0d4352db9b3fc9c25a64cb5902ea5652bf4a2
646fc5ee9af2b1fb9f43886871cfe8bc97da
found pre-master secret in key log
ssl_generate_keyring_material:PRF(pre_master_secret)
```

Reading in $SSLKEYLOGFILE

```
pre master secret[48]:
03 01 a0 80 d0 5f bd 62 09 30 e3 e0 d4 35 2d b9
b3 fc 9c 25 a6 4c b5 90 2e a5 65 2b f4 a2 64 6f
c5 ee 9a f2 b1 fb 9f 43 88 68 71 cf e8 bc 97 da
client random[32]:
51 cc a8 83 57 85 7a 9d 54 0b 6b 18 5d 5d b9 46
6d e0 50 56 e5 2e e5 b3 6f 40 d6 97 64 7b 2c 48
server random[32]:
bd 3c b3 42 b7 1f 6f 36 71 7f d1 1f 3b 25 bc 6c
91 be 7d ac a3 c0 54 9e e9 fc 3d 9d 58 83 cb 50
tls_prf: tls_hash(md5 secret_len 24 seed_len 77 )
tls_hash: hash secret[24]:
03 01 a0 80 ....

      <big snip>

                                ....66 5a 28 92 71 25 b7
2d 9e 68 c1 5b be c0 d2 cb c9 de b9 42 b6 29 d7
master secret[48]:
1d 7d 6b f0 c0 62 3f 02 ec 83 73 0b f0 29 80 74
75 5e 06 59 e6 5c bc 33 8c 66 5a 28 92 71 25 b7
2d 9e 68 c1 5b be c0 d2 cb c9 de b9 42 b6 29 d7
ssl_generate_keyring_material sess key generation
tls_prf: tls_hash(md5 secret_len 24 seed_len 77 )

      <snip>

key expansion[72]:
eb dd 99 67 09 fb bd 42 29 40 74 ea 9d 8a 58 0f
e0 62 c1 60 88 fa 98 9a ab b9 b1 0d c9 98 cd 34
45 5b e3 f9 9d cd 00 e2 a7 91 d2 e9 a2 5a 27 8f
```

Correct entry from key log file containing pre-master secret and random values. These will be inputs to pseudo random function (PRF) to generate master secret.

Master secret computation

Session key generation, continued to next page…

```
1d d5 68 d7 d2 80 87 da 5b e3 c9 d5 cd 89 ba 4c
2a 5c 71 d5 99 8b 3d 9a
Client MAC key[20]:
eb dd 99 67 09 fb bd 42 29 40 74 ea 9d 8a 58 0f
e0 62 c1 60
Server MAC key[20]:
88 fa 98 9a ab b9 b1 0d c9 98 cd 34 45 5b e3 f9
9d cd 00 e2
Client Write key[16]:
a7 91 d2 e9 a2 5a 27 8f 1d d5 68 d7 d2 80 87 da
Server Write key[16]:
5b e3 c9 d5 cd 89 ba 4c 2a 5c 71 d5 99 8b 3d 9a
Client Write IV[8]:
50 ce bf 5f ff 7f 00 00
Server Write IV[8]:
cc cb bf 5f ff 7f 00 00
ssl_generate_keyring_material ssl_create_decoder(client)
ssl_create_decoder CIPHER: ARCFOUR
decoder initialized (digest len 20)
ssl_generate_keyring_material ssl_create_decoder(server)
ssl_create_decoder CIPHER: ARCFOUR
decoder initialized (digest len 20)
ssl_generate_keyring_material: client seq 0, server seq 0
ssl_save_session stored session id[32]:
bd 3c b3 42 b7 1f 6f 36 71 7f d1 1f 3b 25 bc 6c
91 be 7d ac a3 c0 54 9e e9 fc 3d 9d 58 83 cb 50
ssl_save_session stored master secret[48]:
1d 7d 6b f0 c0 62 3f 02 ec 83 73 0b f0 29 80 74
75 5e 06 59 e6 5c bc 33 8c 66 5a 28 92 71 25 b7
2d 9e 68 c1 5b be c0 d2 cb c9 de b9 42 b6 29 d7
dissect_ssl3_handshake session keys successfully generated
```

Session key generation finished. Client/Server Write keys used for application data encryption/decryption.

Session key generation success message

```
 record: offset = 267, reported_length_remaining = 47
dissect_ssl3_record: content_type 20 Change Cipher Spec
dissect_ssl3_change_cipher_spec
packet_from_server: is from server - FALSE
ssl_change_cipher CLIENT
 record: offset = 273, reported_length_remaining = 41
dissect_ssl3_record: content_type 22 Handshake
decrypt_ssl3_record: app_data len 36, ssl state 0x3F
packet_from_server: is from server - FALSE
decrypt_ssl3_record: using client decoder
ssl_decrypt_record ciphertext len 36
Ciphertext[36]:
24 44 16 c1 8b b3 66 f4 e0 e9 87 c9 6b 99 b6 b8
77 bf d8 08 2f 9e 92 d1 06 89 47 14 87 c2 65 4b
3d 84 a5 33
Plaintext[36]:
14 00 00 0c b2 60 f2 5b 73 c4 3a 8d cb d7 67 f7
bb 98 a9 92 50 c9 cf 8f 5d 8b 00 fe f0 1a 09 94
8e 33 b2 ec
checking mac (len 16, version 301, ct 22 seq 0)
tls_check_mac mac type:SHA1 md 2
Mac[20]:
bb 98 a9 92 50 c9 cf 8f 5d 8b 00 fe f0 1a 09 94
8e 33 b2 ec
ssl_decrypt_record: mac ok
dissect_ssl3_handshake iteration 1 type 20 offset  0,
 length 12 bytes, remaining 16
```

Example of decrypted client *Finished* message.

Finished header fields:
Handshake message type =0x14 (*Finished*)

Message length = 0x00000c (12 decimal)

Verify Data = b2 60 ..... (This is the decrypted hash used to verify integrity of TLS handshake)

This fragment from a debug file shows frame number 208. It is a single packet containing the *ClientKeyExchange*, *ChangeCipherSpec* and *Finished* elements of an TLS handshake.

When Wireshark encounters a TLS packet, it searches the key log file for a corresponding entry. If it finds a matching entry in the key log file, it is able to generate "Keying material". This is the "Key expansion" section. The "Key expansion" is a stream divided up to create several different session keys, two of which are client and server write keys. For a complete explanation of the various keys derived from the key stream for a single TLS session see page 20 of RFC 2246 (Dierks & Allen, 1999).

The session keys remain in memory for the duration of the session. When an ApplicationData message is encountered for the session, Wireshark writes the ciphertext to the debug file and then uses the corresponding session key to decrypt the ciphertext without recalculating the keys. It writes the hex encoded plaintext to the debug file as well and for some embedded protocols, like HTTP, it will decode the hex encoded plaintext to the corresponding readable characters or "decrypted app data fragment" as shown in figure B-1.

```
dissect_ssl3_record decrypted len 804
decrypted app data fragment: POST /account/login HTTP/1.1

Host: www.sans.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:21.0)
Gecko/20100101 Firefox/21.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: https://www.sans.org/account/login
Cookie: __utma=157178169.777277707.1372366987.1372366987.1372366987.1;
__utmb=157178169.4.10.1372366987;
  __utmc=157178169;
__utmz=157178169.1372366987.1.1.utmcsr=(direct)|utmccn=(direct)|utmcmd=(none);
  sans=24aa05918bbfbd8db40691fea5f012f6
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 132
_xsrf_token=7fe998ee86a6ee93aeefa4f7f5a6b5d8f46f0539&

email=graceomalley1603%40yahoo.com&password=maytheroadriseuptomeetyou& website
```

Figure B-1. Conversion of hex encoded plaintext to readable text in the debug file, called "decrypted app data fragment"

Wireshark works through the packets, frame by frame, decrypting packets when possible and recording its actions in the debug file.

The display filters available in Wireshark provide a very granular and convenient way to view particular fields in a given packet. This makes it relatively easy to find records in a trace file containing values found in the debug file. For example, to search for the encrypted pre-master secrets of an RSA key exchange use the display filters:

For RSA key exchange:

```
ssl.handshake.epms == 4722.2a2a.2f69.baf7     or
ssl.handshake.epms contains 4722
```

Figure B-2 shows the results of the filter in Wireshark.

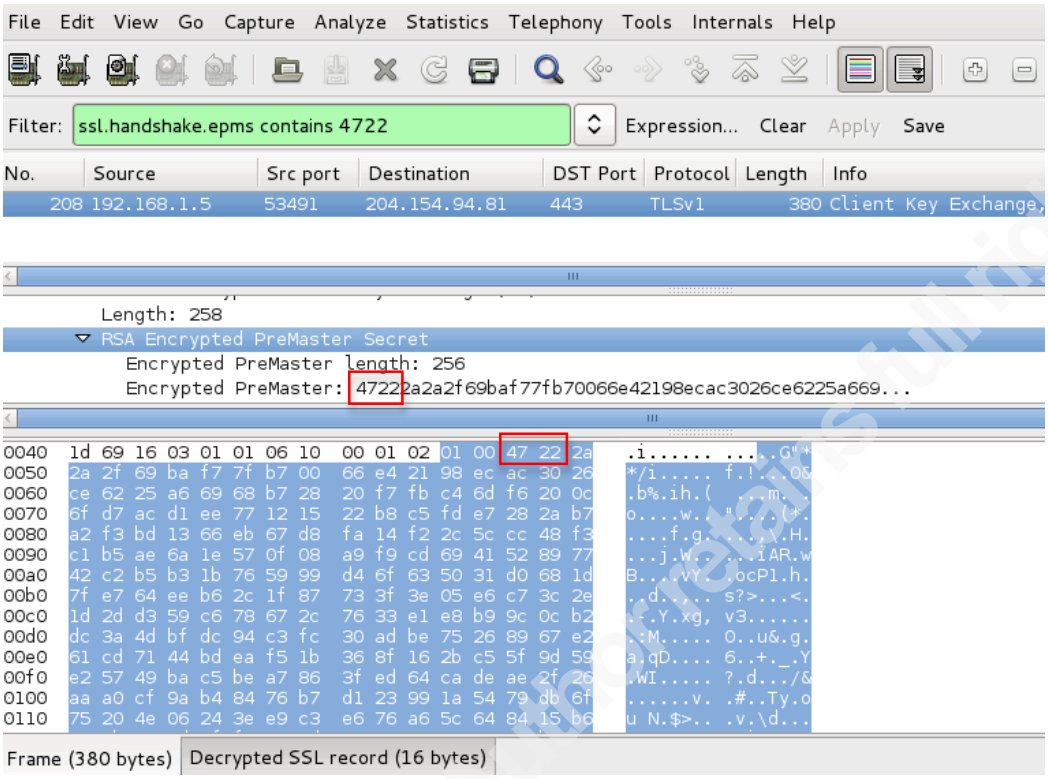

Figure B-2. Using Wireshark display filter to find encrypted pre-master secret from debug file

To view the handshake random values for key generation use the display filter:

```
ssl.handshake.random_bytes == 51cc.a885….<48 bytes
snipped>....fa04.0865
          or
ssl.handshake.random_bytes contains  fa04.0865
```

For a complete listing of Wireshark's display filters is maintained at http://www.wireshark.org/docs/dfref/.

Figure B-3 shows the results of the random_bytes filter in Wireshark.

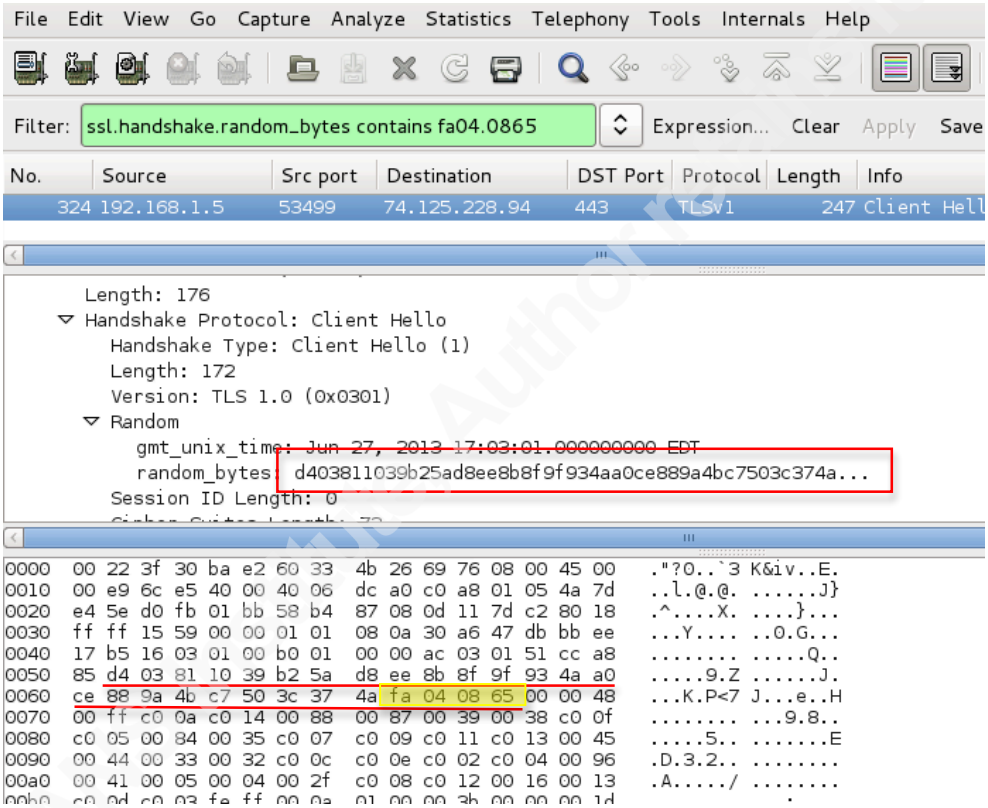

Figure B-3. Using Wireshark display filter to find a Client Random Bytes field from the debug file