SHARKFEST 2015

WIRESHARK DEVELOPER AND USER CONFERENCE

SSL DOES NOT MEAN SOL

What if you don't have the server keys?

J. Scott Haugdahl
Architect, Blue Cross Blue Shield MN

Robert Bullen
Systems Engineer, Blue Cross Blue Shield MN

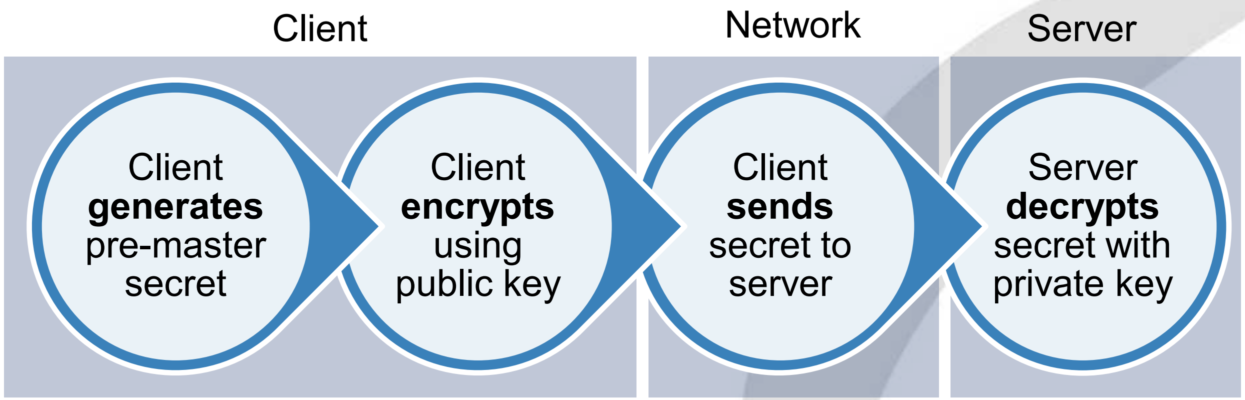
Setting Expectations

- This session is not about..
  - An introduction to SSL encryption
  - How to set up SSL decryption in Wireshark
  - A detailed walk through of the SSL handshake and all the variants
- This session is about…
  - What you can do when you *do not have access to server keys*
    - Calculating server command response time from SSL, even in the cloud
    - Using encrypted data to your advantage
    - Identifying application layer behavior based on SSL patterns
  - Walking through real world examples using Wireshark
  - Focus will be on helping you to analyze application performance more so than security breaches, suspicious activity, etc.

A (very) Brief History of Secure Sockets Layer (SSL)

- Used to encrypt + protect integrity of network data
  - SSL 2.0 was first “public release” in 1995
  - SSL 3.0 released in 1996 forming the foundation for Transport Layer Security (TLS) 1.0 (RFC 2246, 1999)
  - TLS 1.0 is not backward compatible with SSL 3.0!
  - Upgraded to TLS 1.1 (RFC 4346, 2006) and TLS 1.2 (RFC 5346, 2008)
- Supports a wide variety of encryption algorithms
  - RSA and DSA are *asymmetric* (public key encrypts; private key decrypts) – used to <u>exchange and generate key information</u> during the SSL handshake
  - AES and 3DES are *symmetric* algorithms (one key encrypts and decrypts) – used to <u>transfer data</u> (much faster to compute) after the SSL handshake
- TLS 1.0 or higher is recommended practice
  - Many clients & systems now support TLS 1.2 which addresses some vulnerabilities

What’s so Special About the Client Key Exchange?



Both Client and Server generate the master secret from the pre-master to generate the session key.

Therefore, Wireshark needs the server’s private key to decrypt the client pre-master secret to order to generate the master secret to generate the session key to decrypt the SSL packet data!

A Tale of Two Connections

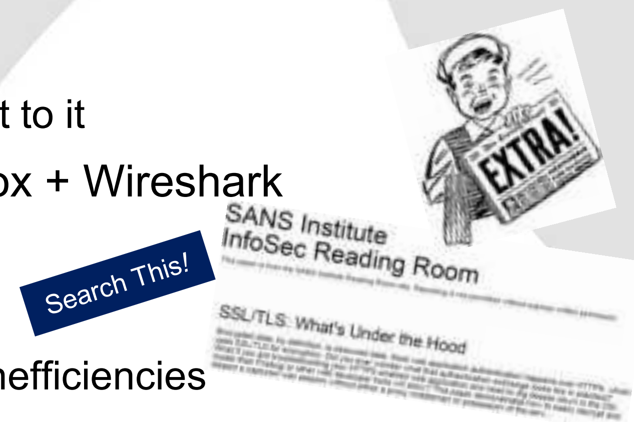
Good, we will get the client key exchange!

No.	Length	Source	Destination	Protocol	Stream index	Sess ID Len	Info
155	62	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	9		54785□9119 [SYN] Seq=312491139 Win=8192 Len=0 MSS=1460 SACK_PERM=1
156	62	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	9		9119□54785 [SYN, ACK] Seq=523685150 Ack=312491140 Win=14600 Len=0 MS
157	54	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	9		54785□9119 [ACK] Seq=312491140 Ack=523685151 Win=64240 Len=0
159	62	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	10		54789□9119 [SYN] Seq=2035020690 Win=8192 Len=0 MSS=1460 SACK_PERM=1
161	62	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	10		9119□54789 [SYN, ACK] Seq=3851424993 Ack=2035020691 Win=14600 Len=0
162	54	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	10		54789□9119 [ACK] Seq=2035020691 Ack=3851424994 Win=64240 Len=0
170	292	BCBSMN911 [illegible]	proxy-mb. [illegible]	HTTP	9		CONNECT encrypted-tbn3.gstatic.com:443 HTTP/1.1
171	270	BCBSMN911 [illegible]	proxy-mb. [illegible]	HTTP	10		CONNECT ssl.gstatic.com:443 HTTP/1.1
172	60	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	9		9119□54785 [ACK] Seq=523685151 Ack=312491378 Win=15544 Len=0
174	60	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	10		9119□54789 [ACK] Seq=3851424994 Ack=2035020907 Win=15544 Len=0
194	192	proxy-mb. [illegible]	BCBSMN911 [illegible]	HTTP	10		HTTP/1.1 200 Connection established
195	268	BCBSMN911 [illegible]	proxy-mb. [illegible]	TLSv1.2	10	0	Client Hello
196	192	proxy-mb. [illegible]	BCBSMN911 [illegible]	HTTP	9		HTTP/1.1 200 Connection established
197	571	BCBSMN911 [illegible]	proxy-mb. [illegible]	TLSv1.2	9	32	Client Hello
210	1514	proxy-mb. [illegible]	BCBSMN911 [illegible]	TLSv1.2	10		Server Hello
211	504	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	10		[TCP segment of a reassembled PDU]
212	54	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	10		54789□9119 [ACK] Seq=2035021121 Ack=3851427042 Win=64240 Len=0
215	1514	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	10		[TCP segment of a reassembled PDU]
216	598	proxy-mb. [illegible]	BCBSMN911 [illegible]	TLSv1.2	10		Certificate
217	54	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	10		54789□9119 [ACK] Seq=2035021121 Ack=3851429046 Win=64240 Len=0
219	60	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	9		9119□54785 [ACK] Seq=523685289 Ack=312491895 Win=16616 Len=0
222	207	proxy-mb. [illegible]	BCBSMN911 [illegible]	TLSv1.2	9	32	Server Hello, Change Cipher Spec, Hello Request, Hello Request
224	316	BCBSMN911 [illegible]	proxy-mb. [illegible]	TLSv1.2	10		Client Key Exchange, Change Cipher Spec, Hello Request, Hello Request
226	241	BCBSMN911 [illegible]	proxy-mb. [illegible]	TLSv1.2	9		Change Cipher Spec, Hello Request, Hello Request
227	60	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	9		9119□54785 [ACK] Seq=523685442 Ack=312492082 Win=17688 Len=0
238	60	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	10		9119□54789 [ACK] Seq=3851429046 Ack=2035021383 Win=17688 Len=0
241	98	proxy-mb. [illegible]	BCBSMN911 [illegible]	TCP	10		[TCP segment of a reassembled PDU]
242	152	proxy-mb. [illegible]	BCBSMN911 [illegible]	TLSv1.2	9		Application Data, Application Data
243	288	proxy-mb. [illegible]	BCBSMN911 [illegible]	TLSv1.2	10		New Session Ticket
244	54	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	10		54789□9119 [ACK] Seq=2035021383 Ack=3851429422 Win=63864 Len=0
252	54	BCBSMN911 [illegible]	proxy-mb. [illegible]	TCP	9		54785□9119 [ACK] Seq=312492082 Ack=523685540 Win=63851 Len=0

Rats, the client is reusing a previous session ID and the server accepts.

What if we don’t have the client key exchange*?

- If your SSL session reused the Session ID…
  - Try to find a trace containing the original handshake containing the key exchange and pre-pend it
- Use Fiddler or similar
  - As a proxy that runs on the client
  - As a proxy on another workstation & point the remote client to it
- Use client pre-master secret logged by Chrome or Firefox + Wireshark
  - This is cool ‘cuz we don’t need the server key to decrypt it
- When all else fails…
  - Use knowledge of TCP & SSL segmentation to watch for inefficiencies
    - SSL payload size (small is probably ok for SSH but not FTP)!
  - Identify unlike flows across firewalls using encrypted data pattern matching
  - Look for other factors that throttle throughput in other sessions



*Or the client key exchange uses Diffie-Hellman in which we are  even if we posses the server key.

Diffie-Hellman

- Described in a 1976 White Paper by Whitfield Diffie and Martin Hellman
- Protects against long-term key compromise (i.e. server keys!)
- Is not SSL specific, can be used for any secret information exchange
- Client generates a random number, as does server
  - Thus forms a way for the client to encrypt the pre-master (already encrypted with the server’s public key) back to the server

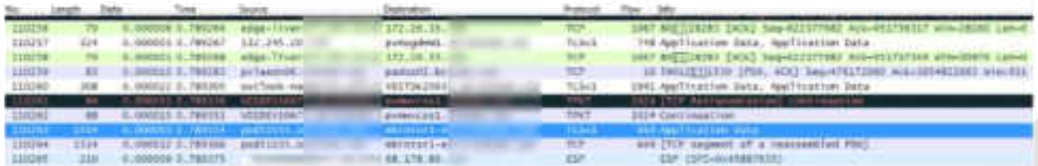
Diffie-Hellman

Use Case: Firewall Pattern Matching

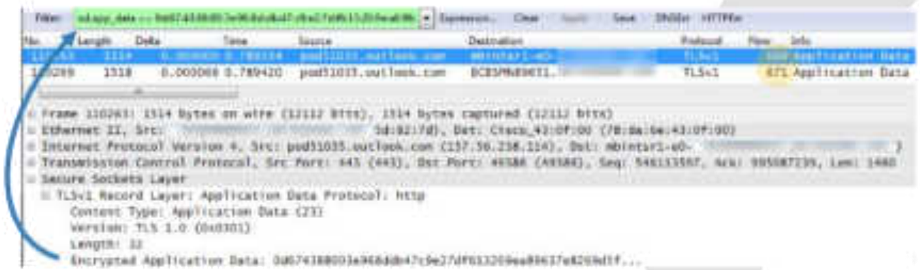
- Perimeter firewalls NAT from private to public IP
  - Terminates TCP but maintain SSL session data
  - Unfortunately, we cannot say the same for proxy servers, load balancers, or anything else that terminates SSL connections
- Simply grab some binary data (i.e. encrypted) from SSL on one side of the firewall and filter on it to find the other side
- Once you have a match, you can then filter on the TCP streams and determine the firewall delay and other characteristics
  - Do not use SPANs nor multiple sniffers due to delays and timestamp synchronization
  - Best practice is to use taps above and below the firewall that feed a common sniffer or are combined via a visibility fabric (Apcon, Big Switch, Gigamon, Ixia, VSS, etc.)
- Also works great for following encrypted VMWare VDI streams (filter on UDP payload) across multiple tiers

Using Wireshark to Find NATed SSL Flows

❶ Start with a pool of packets captures inside and outside of the firewall…

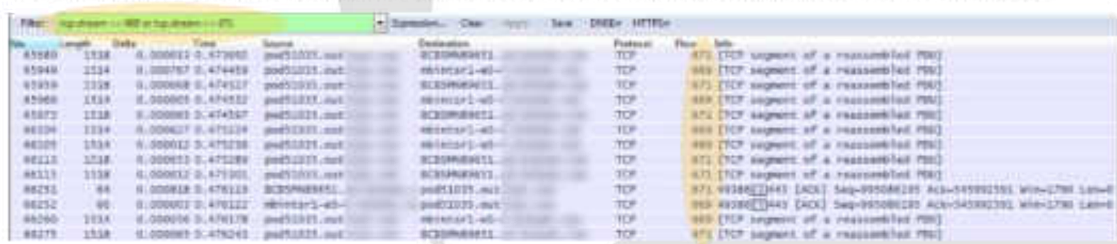


❷ Filter on some SSL data from the flow of interest into the firewall…



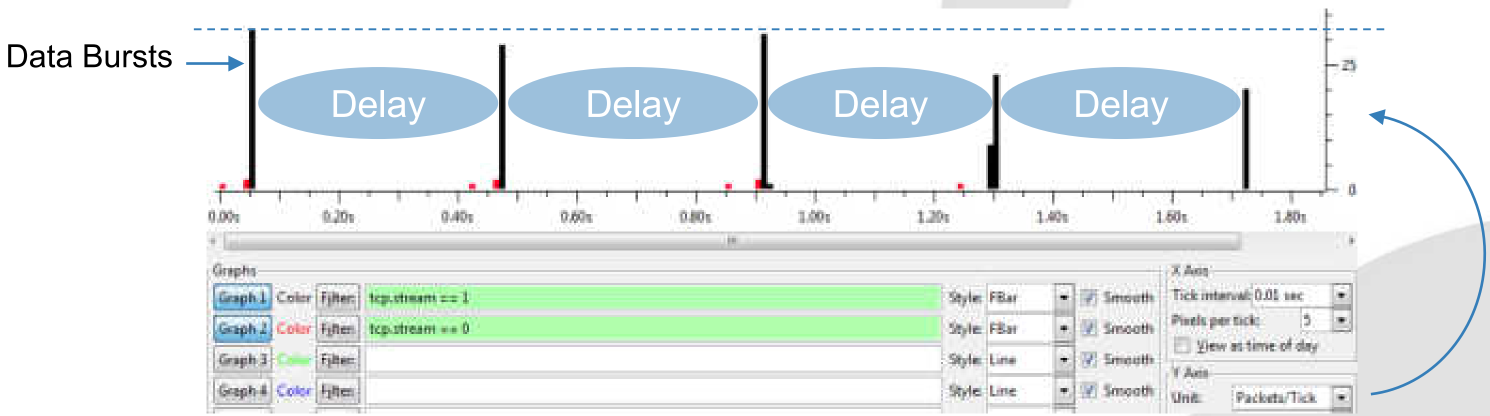
Which picks up the matching flow on the other side of the firewall.

❸ We now have our two flows either side of the firewall for focused analysis



Use Case: Slow eMail Migration

- Migrating user’s mailboxes from internal Lotus Notes servers to Microsoft Office 365 in the Cloud
  - Typical mailbox size was 50 GB
  - Throughput varied from 200-500 kbps over a 1 gig Internet pipe
  - 4k users @ 1 hour per user = 166 days!
- Subsequent web proxy bypass did not help nor did moving to DMZ
- Graphing the I/O revealed a potential problem area



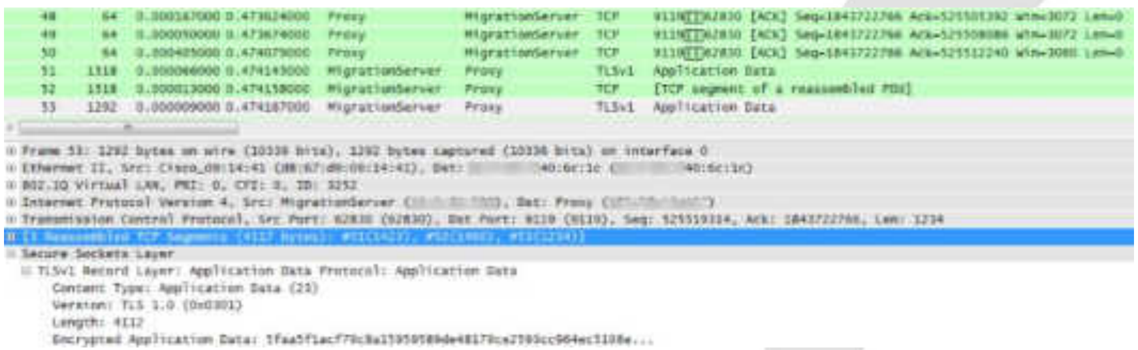
Use Case: Slow eMail Migration

- A pattern emerged when walking through the SSL & checking neighboring flows
  - *A **second flow** (in red below) running was clearly controlling the throughput*
  - The throttling was set to approximate three bursts or blocks of data per second
  - Properties could not be changed, i.e. they are controlled by the (MS) cloud server

	No.	Length	Delta	Time	Source	Destination	Protocol	Info
	27	64	0.000033000	0.054483000	Proxy	MigrationServer	TCP	9119□□62830 [ACK] Seq=1843722766 Ack=525495198 Win=3006 Len=0
	28	1518	0.000078000	0.054561000	MigrationServer	Proxy	TLSv1	Application Data
	29	1518	0.000013000	0.054574000	MigrationServer	Proxy	TCP	[TCP segment of a reassembled PDU]
	30	1292	0.000009000	0.054583000	MigrationServer	Proxy	TLSv1	Application Data
	31	64	0.000159000	0.054742000	Proxy	MigrationServer	TCP	9119□□62830 [ACK] Seq=1843722766 Ack=525499352 Win=2987 Len=0
	32	484	0.000070000	0.054812000	MigrationServer	Proxy	TLSv1	Application Data, Application Data
	33	64	0.000630000	0.055442000	Proxy	MigrationServer	TCP	9119□□62830 [ACK] Seq=1843722766 Ack=525500812 Win=2979 Len=0
	34	64	0.000046000	0.055488000	Proxy	MigrationServer	TCP	9119□□62830 [ACK] Seq=1843722766 Ack=525503506 Win=2967 Len=0
Stop →	35	64	0.000193000	0.055681000	Proxy	MigrationServer	TCP	9119□□62830 [ACK] Seq=1843722766 Ack=525503832 Win=2964 Len=0
	36	196	0.368240000	0.423921000	Proxy	MigrationServer	TLSv1	Application Data, Application Data
Wait	37	292	0.045276000	0.469167000	Proxy	MigrationServer	TLSv1	Application Data, Application Data
	38	64	0.000134000	0.469531000	MigrationServer	Proxy	TCP	62831□□9119 [ACK] Seq=653341697 Ack=3420581597 Win=511 Len=0
Start →	39	1518	0.003211000	0.472742000	MigrationServer	Proxy	TLSv1	Application Data
	40	1518	0.000011000	0.472753000	MigrationServer	Proxy	TCP	[TCP segment of a reassembled PDU]
	41	1292	0.000009000	0.472762000	MigrationServer	Proxy	TLSv1	Application Data
	42	1518	0.000394000	0.473156000	MigrationServer	Proxy	TLSv1	Application Data
	⋮							
Stop →	67	64	0.000183000	0.478359000	Proxy	MigrationServer	TCP	9119□□62830 [ACK] Seq=1843722766 Ack=525529250 Win=2980 Len=0
	68	356	0.380602000	0.858961000	Proxy	MigrationServer	TLSv1	Application Data, Application Data, Application Data
Wait	69	132	0.052061000	0.909022000	Proxy	MigrationServer	TLSv1	Application Data
	70	64	0.000460000	0.909482000	MigrationServer	Proxy	TCP	62831□□9119 [ACK] Seq=653341697 Ack=3420581909 Win=510 Len=0
Start →	71	1518	0.006197000	0.915679000	MigrationServer	Proxy	TLSv1	Application Data

Use Case: Slow eMail Migration

- Each data stream was equated to one piece of mail
  - Due to control channel, conversion rate was approximately three emails per second(!)
  - Another potential optimization was to increase the application layer block size to greater than 12k (which we derived from the SSL segment size of 4112 bytes x 3 per turn)



- Solution was to run multiple severs simultaneously with multiple mailbox migrations per server to the cloud, which is per MS recommendation
  - We were running up to 40 migrations in parallel at the peak
  - All mailboxes were migrated in under 30 days

Wrapping it Up

- First gain a solid understanding of the general application layer command-response characteristics in the unencrypted world (HTTP, SQL, mail, etc.)
  - Pretend that the SSL layer *is* the application layer and apply those characteristics
- Figure out who is the client and who provides the data
  - Usually the client opens the connection, but not always!
- Breakdown the TCP segmentation and the SSL segmentation
  - Ensure that the SSL segment size makes sense for the application (SSH vs. HTTPS for instance)
- Identifying network from back-end response time is easier but must use patterns and neighboring flows for more complex cases

Thank You!

Contact us!

scott.haugdahl@bluecrossmn.com

robert.bullen@bluecrossmn.com