Speed, Privacy, and Control on the Web

TLS 1.3 Handling in the Web Security Appliance

Introduction

More than ever before, the discussion of network security is intersecting with the interests of business, government, and popular media. The Transport Layer Security (TLS) protocol is perhaps the best modern example of this. There are stakeholders on all sides who demand both a secure end-to-end connection for clients, as well as the tools necessary to inspect and control the content of traffic in their private networks. As a leader in content security, Cisco must meet both of these requirements by supporting the latest Internet security standards while at the same time enabling businesses to protect their networks from malicious traffic and enforce their usage policies. TLS version 1.3 represents the latest iteration of secure Internet transport and brings with it special challenges when meeting these needs. The Cisco® Web Security Appliance (WSA) is uniquely positioned in the market to meet these needs while incorporating the benefits of TLS 1.3. This paper will explain some of the more relevant points of TLS 1.3 as they pertain to the WSA.

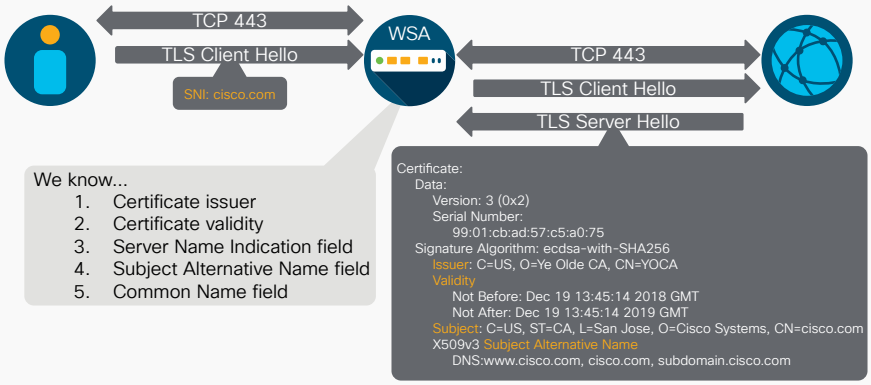
How the WSA handles TLS 1.3 today

The Web Security Appliance accepts both explicit and transparent TLS connections from clients to be proxied to web servers. In almost all cases, the WSA will negotiate an independent TLS connection with the web server in order to validate the Common Name, Subject Alternative Name, and validity of the web server certificate, and the issuer. The WSA will cache the certificate to increase the performance of certificate validation. The only case in which the WSA forgoes the certificate validation is when the **HTTP CONNECT** request (in an explicit deployment) or **Server Name Indication** (SNI) field (in a transparent deployment) contains an IP address or host name that is present in a custom category, which is set to **passthrough**.

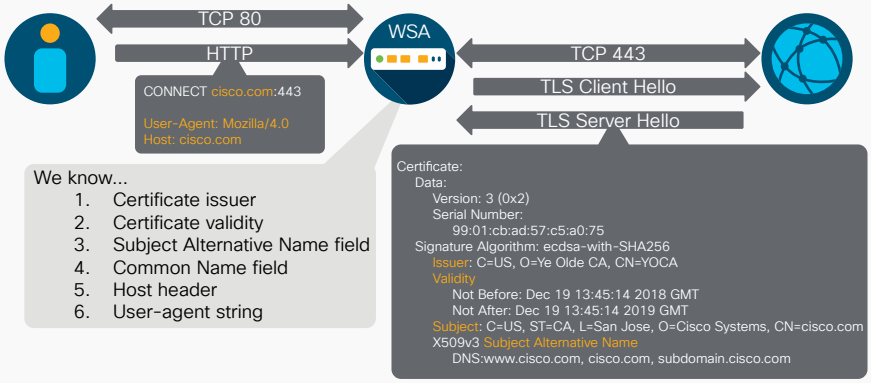
Contents

The following section illustrates the connection flow in both explicit and transparent scenarios, where the policy is **decrypt**.

Transparent HTTPS - What do we know?

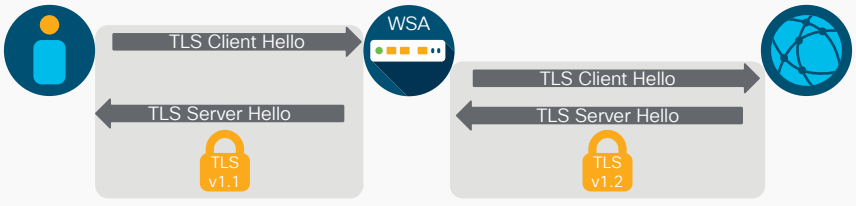


Explicit HTTPS - What do we know?



After these checks are done, if the policy decision is **passthrough**, then the WSA will send a TCP FIN packet in order to close the server-side connection. A new TCP connection will be established with the server, and the client will be allowed to negotiate a connection directly with the web server. If the policy decision is to decrypt the connection, a separate TLS session will be maintained on the server side. Because the two connections are independent, they may use different TLS versions, different encryption parameters, and so on.

Negotiating two TLS versions



There is one other possible scenario in a transparent deployment, which is known as **bypass**. If the deployment uses the **Web Control Cache Protocol** (WCCP) to redirect traffic to the WSA, the **bypass settings** list can be used to route traffic back to the WCCP device (switch, router, or firewall) to be routed directly to the web server. In this case, the WSA does not evaluate the request at all, and the client is free to negotiate any version of TLS with the web server without any interaction with the WSA.

Because every TLS 1.3 web server today also supports TLS 1.2, if the connection is decrypted, the WSA will simply negotiate a TLS 1.2 connection to the server. Because there are two independent connections, the client and server are not aware that each may be capable of a higher TLS version. This is how all decrypted TLS 1.3 connections are handled in the WSA today; they are downgraded to TLS 1.2. If the decision is **passthrough**, the client is permitted to negotiate directly with the server and set up a TLS 1.3 connection if both support it. Connections that are passed through via web category or custom category can both support TLS 1.3. This is true for both explicit and transparent connections.

In the future, the WSA will support TLS 1.3 in both modes, **decrypt** and **passthrough**.

Encrypted certificates

Digital certificates are used to prove the authenticity of endpoints. In the majority of cases, this is done on the server side only to ensure that the client is connected to the intended host and not an imposter. In all previous versions of TLS, the certificate is sent in the clear as part of the protocol negotiation. This meant that anyone in a position of snooping on the communication could glean the name of the server (among other details). Some middle boxes (proxies, load balancers, next-generation firewalls, etc.) leveraged this visibility to block traffic without actively decrypting the connection. Decrypting and re-encrypting the TLS stream is resource intensive, and many of these appliances minimize CPU cycles via these types of passive enforcement mechanisms. The TLS 1.3 standard encrypts the server certificate. This defeats network devices that rely on snooping the certificate in order to make policy decisions.

As you may have guessed, the WSA is not one of these devices. Obtaining the certificate is still very much part of the policy evaluation in the WSA; the **Common Name** and **Subject Alternative Name** fields can be used as one way to identify the server, and traffic can be blocked depending on whether the certificate is valid (e.g., it is not expired, it is properly formed, etc.). The key difference is that in cases where the WSA evaluates the certificate, it does not do this passively by relying on the plaintext being visible on its way to the client. It instead actively connects to the server and completes a full TLS handshake. This independent connection allows it to obtain every part of the negotiation, including the plaintext certificate, even in cases when it is encrypted. For this reason, the encrypted certificate itself presents no real challenge to the current implementation of HTTPS proxy in the WSA.

The end of renegotiation

Previous versions of TLS allowed for the client or server to renegotiate the parameters of a connection at any time during communication. This allowed for a complete change of the encryption parameters used to protect the underlying communication. Patches were issued in 2011 to prevent a **Man in the Middle** (MITM) attack, which leveraged renegotiation to force downgrades to less secure encryption parameters, but until now, renegotiation was still possible. Some middle boxes utilized this in order to reduce the resource usage involved in proxying TLS. After decrypting the first few packets of a connection, they would force a renegotiation and allow the session to resume between the client and server directly, thus “bowing out” of the connection and freeing the CPU cycles that would have been spent decrypting the session. TLS 1.3 has removed renegotiation entirely from the specification. The devices that use this technique will now need to either fully decrypt the TLS session, fully pass it through, or close it and allow another “from scratch” setup between the client and server.

The WSA is a full HTTPS proxy and does not use renegotiation. Per the decryption policies, the appliance will either fully decrypt the stream or pass through the connection to allow a complete tunnel between the client and server. In this way, the removal of renegotiation does not affect the WSA’s ability to proxy TLS.

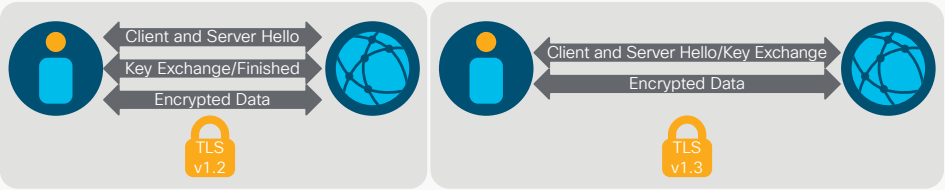
Increased performance

Prior to TLS 1.3, a TLS handshake required two round trips to complete. This means that one packet must be transmitted from client to server and back, and a second from server to client and back. For HTTPS, you can add the TCP handshake (one round-trip time [RTT]) and the HTTP request (one round trip). In places where latency is a real concern (satellite links, 2/3G, etc.), a total RTT of four is impactful to the user experience, especially on the modern Internet in which a single site may cause dozens of connections to be established. It was possible to "resume" a session to a previously connected server in TLS 1.2, which potentially brought the RTT to one. TLS 1.3 promises to bring TLS RTT to one, even for brand-new (nonresumed) sessions. This may not sound like much of a difference, but when the sheer volume of TLS on the Internet is taken into account, it should definitely produce a noticeable impact in the user experience.

There is another feature of TLS that is even more dramatic when it comes to latency. There is support in the specification for **"0 RTT,"** meaning that the client can send encrypted application data in the very first packet without any negotiation. This relies on a previous connection to the server in which the client received a preshared key that maps to a session identifier on the server. There are real security concerns with this, however. The drafters of the RFC warn that a 0-RTT packet is "replayable." This means that a passive listener on the network could capture the packet and send it out again later, masquerading as the original client. Imagine an HTTP POST that transferred money from one bank account to another. If that request were to be replayed, it could cause actions with real consequences to occur on the server side. For this reason, there has been guidance to only allow 0 RTT for GET requests—that is, HTTP methods that don't make any change on the server side. However, this does not address other possible privacy concerns of allowing GET requests to be replayed.

For the reasons above, it is not clear that 0 RTT will be widely adopted with TLS 1.3. At least initially, this will not be a supported feature in the WSA TLS 1.3 implementation. Once the standard has been put into practice and a clearer picture of its use emerges, there could be some more exploration of this, but for now, it will not be available.

Faster handshake



Encrypted SNI

The **Server Name Indication** (SNI) extension to TLS was introduced in the mid 2000s in order to solve the problem that name-based virtual hosting created. A single server could host many differently named web servers, and it is not clear from the **Client Hello** message which certificate should be sent. The SNI extension is a way for the client to provide the name of the server to which it intended to connect so that the server can send the appropriate certificate. This can also be snooped passively, as it is sent in the clear. The TLS 1.3 specifications do not include encrypted SNI, but there is a separate draft RFC (**draft-ietf-tls-esni-03** as of the publishing of this document) that describes it. The current draft explains that a public key would be provided as an extension to the DNS record of a host name. When the client resolved the server name to an IP address, it would also get a key that it could use to encrypt the SNI when reaching out to the server.

The loss of visibility into this field poses some challenges, but it is important to point out that there are nontrivial obstacles to the practical implementation of encrypted SNI. The first of these obstacles is the burden it will place on the operators of Content Delivery Networks (CDNs). There are large, distributed content providers like Akamai who terminate TLS on behalf of third parties.

CDNs introduce load balancing, caching, and other performance enhancements by simply bringing content closer to users. Until recently, many mainstream CDNs prioritized the host header field in the encapsulated HTTP request over the SNI. This gave rise to a technique known as domain fronting, where the SNI of a legitimate resource could be used when connecting to a CDN, but the actual HTTP request inside the TLS tunnel requested a malicious resource that was reachable behind the CDN. While this technique was used by many legitimate applications to evade censorship and tracking by oppressive governments, it was also being abused by malware authors to hide malicious command-and-control traffic, and thus Google and Amazon have begun blocking it in their CDN networks. The blocking of domain fronting requires that CDNs rely more heavily on the SNI extension, and if they plan to do this when encrypted SNI is in use, there will be an additional burden on them to be able to decrypt the field.

The second obstacle to the successful implementation of encrypted SNI is the current state of DNS. If a passive eavesdropper can snoop the DNS traffic on a network, it is trivial to discover the intended destination of a TLS connection from a given user. If a TLS connection is seen destined for a given IP address, and the same client resolved a given host name to that IP address immediately before initiating the TLS connection, it is reasonable to assume that the host in the DNS query is the destination server. For this reason, SNI is suggested to be implemented alongside DNS over HTTPS (DOH) or DNS over TLS (DOT). This requires that a DNS server known to support DNS over HTTPS be configured as the resolver for the client. In fact, this is the implementation in the latest builds of Mozilla Firefox and Google Chrome. This also presents problems for businesses that rely on internal DNS to maintain their Active Directory domains and intranet sites because Firefox and Chrome cannot currently distinguish internal from external DNS resources.

The above obstacles are not impossible to overcome, but they are reasons to believe that encrypted SNI will not be widely adopted, at least as much as TLS 1.3.

Conclusion

There are many other interesting aspects of the new TLS 1.3 specification, and as with any standard, only time will tell how and to what extent many of them take hold in the real world. The features mentioned here are only a few of those that directly affect the Cisco Web Security Appliance. Cisco is dedicated to following the evolution of the modern Internet and empowering administrators and engineers to attain the highest levels of security while maintaining control of their networks. Much more will be written on this and other subjects as Internet standards continue to rapidly evolve.

C11-742216-01 05/19