HTTPS Posture Assessment

HTTPS has been around since 1994. Historically, HTTP over Secure Sockets Layer (SSL)/Transport Layer Security (TLS) was treated as a dark and capricious form of magic best left undisturbed. For most of its existence, the general consensus was that HTTPS was for securing access to websites such as banks, or to websites from which one could make online purchases. In the early days, it was not uncommon for only the authentication to those sites to be encrypted and, once authenticated, everything reverted back to plain old, unencrypted HTTP. Though the demand for greater privacy and security has only increased, and HTTPS is becoming more common, there is still an alarmingly high number of major websites that do not employ HTTPS at all, implement it incorrectly, or are configured to use old, outdated methods. Given its importance, inadequate, incorrect or nonexistent HTTPS is a cause for concern.

Then there is the matter of internal websites. For decades, conventional wisdom held that if the traffic was on an internal network, it was sufficiently secure and adding HTTPS to the mix was an unnecessary complication. Often paired with this argument was the notion that HTTPS caused such enormous performance degradation that implementation was simply out of the question. Both of these points, while possibly valid at some point in the distant past, could not be more wrong today.

Consensus is changing, however. Slowly, the world is inching in the direction of ubiquitous HTTPS. Unfortunately, pervasive use of secure, correctly configured HTTPS is often surprisingly low. What follows is a discussion of the nature of HTTPS, how it should be configured, and how to remotely assess that configuration for oneself, rather than relying on verbal or written attestation from server or application administrators.

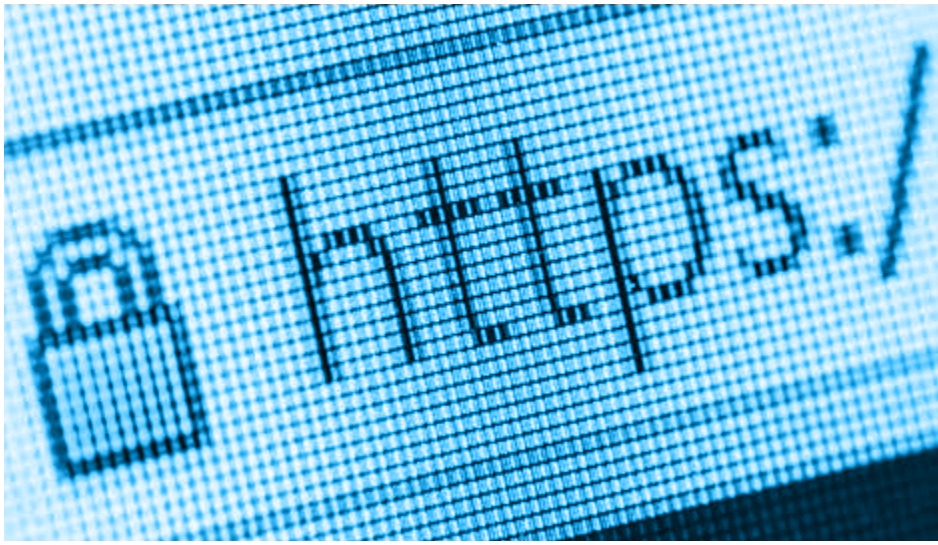
Background

HTTPS is implemented using protocols that operate at the application layer, layer 7 of the OSI model. Version 1 of the SSL protocol had enough serious flaws that it was discarded before being released. The earliest released protocol version was SSLv2, which was released in 1994.[1] SSLv3 was released in 1995, attempting to address vulnerabilities in SSLv2. With the next version of the protocol, the name was changed to TLS and was released in early 1999 as TLSv1. TLSv1.1 was released in 2006, followed by TLSv1.2 in 2008. TLSv1.3 is currently in draft as of the date of this writing.

It is important to note that each successive protocol version was designed to address shortcomings in the previous version. Older protocol versions continue to be used, however. For example, it is not completely unheard of to encounter a site that still supports SSLv2, even though it is known to have been severely flawed for more than 20 years.

Ideal Configuration

Before getting into the details of how to check an organization's HTTPS security posture, it is



Kurt Kincaid, CISA, Lean Six Sigma Green Belt
Is an information security professional with a Fortune 50 company. He focuses on encryption and its implementation, certificate/key management, access management, and open-source security solutions. He can be reached at kurt@kurtkincaid.com.

helpful to have some concept of the ideal secure configuration. While this article is not intended to be a configuration manual, it will help to put things in context, allowing easy identification of strong configuration, configuration that is slightly off the mark, or configuration that is so incorrect or outdated as to be completely unacceptable.

Ideally, TLSv1.2 should be used to the exclusion of all other versions. Payment Card Industry Data Security Standard (PCI DSS) v3.2[2] requires that all payment card data be encrypted using TLSv1.1 or TLSv1.2 by 30 June 2018. Certainly, not all data are PCI-related, but with PCI DSS v3.2 having set the bar, it is reasonable to expect other standards and regulations to follow suit.

As mentioned previously, SSLv2 is so badly broken that it is not suitable for any purpose. SSLv3, which had been on its way out for quite some time, was rendered essentially useless by the Padding Oracle on Downgraded Legacy Encryption (POODLE) attack and, likewise, should not be used. To explain how this affects TLSv1, TLSv1.1 and TLSv1.2, a brief digression into the POODLE attack itself is required, as well as some of the underlying components of TLS. Encryption falls into two categories: symmetric and asymmetric. With symmetric encryption, the same key is used to encrypt and decrypt. With asymmetric encryption, there is a public key and a private key. What is encrypted with the public key can be decrypted only by the private key. It is important to note that asymmetric keys are used for authentication only during the SSL/TLS handshake. Once the handshake completes successfully, all encryption switches over to using symmetric encryption for speed and efficiency.

Symmetric encryption also falls into two categories: stream ciphers and block ciphers. Stream ciphers operate on one byte at a time, whereas block ciphers operate on blocks of bytes of a fixed length. Depending upon the length of the message to be encrypted, it may need to be padded at the end to make the length a multiple of the block size. The POODLE attack exploits how this padding is done. Therefore, any TLS block cipher that uses padding is potentially vulnerable to some variant of the POODLE attack. This encompasses the vast majority of cipher suites. The only remaining option is to look toward stream ciphers, which do not use padding.

The only stream cipher options are RC4 (very badly broken and should never be used) or Advanced Encryption Standard (AES)—a block cipher—when it is used in a mode that makes it behave like a stream cipher. One such mode, Galois/Counter Mode (GCM), was introduced in TLSv1.2. Prior protocol versions do not support AES-GCM cipher suites. An interesting side benefit of the AES-GCM cipher suites is that they tend to be significantly faster than the other AES cipher suites. In extensive performance testing performed on a wide variety of hardware, AES-GCM cipher suites are 40 to 80 percent faster.

As a result, the only configuration one should be using is TLSv1.2 with the only AES-GCM cipher suites enabled. This leaves open the question of backward compatibility and, depending upon the circumstances, one may need to leave lower-protocol versions and weaker ciphers enabled. These should be enabled only if there is a very specific reason.

“Encryption falls into two categories: symmetric and asymmetric.”

What follows is a checklist of configuration items for an ideal deployment. Some of the points mentioned fall outside the scope of this article. For detailed explanations, readers are strongly encouraged to refer to a reliable source such as the Qualys SSL Labs website[3] or *Bulletproof SSL and TLS*,[4] both of which are discussed herein.

Configuration checklist:

- Enable TLSv1.2 only. Explicitly disable SSLv2, SSLv3, TLSv1 and TLSv1.1.

- Enable cipher suites that support AES in GCM mode only, for example, ECDHE-ECDSA-AES256-GCM-SHA384.
- Enable only cipher suites that support forward secrecy,[5] denoted by having "DHE" in the name, as in the previous example.
- Enable cipher suites that support SHA256 or SHA384 only.
- Use only digital certificates that have an RSA key of at least 2,048 bits or an elliptic curve key of at least 256 bits.
- Include the certificate's full trust chain, not including the root certification authority (CA) certificate.
- Do not include unnecessary certificates in the trust chain.
- Ensure that compression is disabled.
- Ensure the server supports TLS Fallback Signaling Cipher Suite Value (SCSV)[6] to prevent protocol downgrade attacks.

Checking the Posture

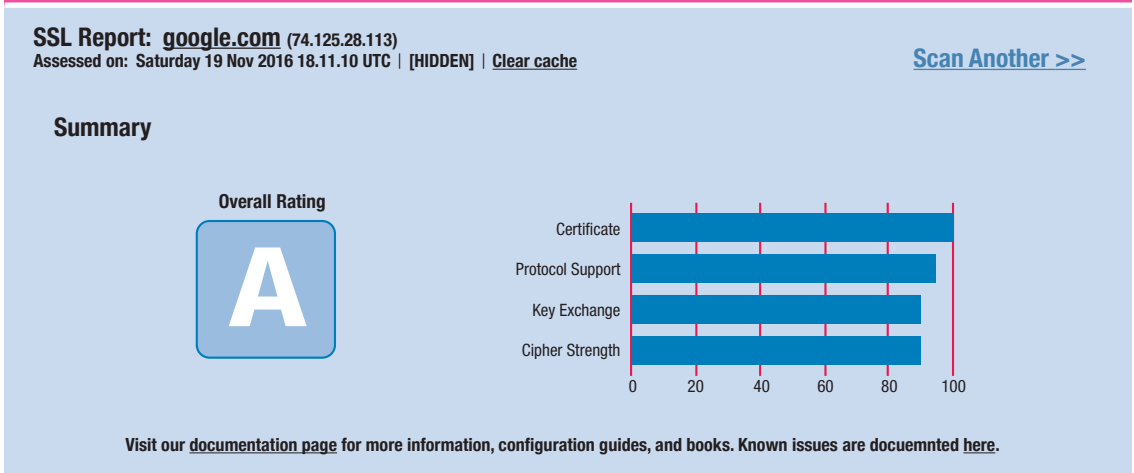
While there are many subtleties to an HTTPS connection and the configuration can become quite complex, assessing the details of the configuration is actually relatively simple, given the right tools. What follows is a brief discussion of the tools that can be used for this task.

SSL Labs

To assess the HTTPS configuration of an Internet-facing server, the easiest and, arguably, the most thorough method is to use the Qualys SSL Labs[7] website. At the site, one can enter a server's address and, after a few minutes, the SSL Labs site will return the full details of how HTTPS is configured on the server, including supported protocol versions, supported cipher suites and numerous other details. Even better, the site does much of the interpretation of the results, assigning the organization's site a grade between A and F, along with an explanation for the grade (**figure 1**).

In addition to the details, the site also provides documentation and recommendations on how to address whatever shortcomings it detects in the server's configuration. The author of the site knows the details and inner workings of TLS as well as anyone, and his documentation and methodology are among the best this author has ever encountered. His book, *Bulletproof SSL and TLS*,[8] is necessary reading for anyone who wants a full understanding of TLS, how it works and how to configure it correctly.

Figure 1—Qualys SSL Labs Sample Output



Source: Qualys SSL Labs. Reprinted with permission.

The SSL Labs site works very well for sites that are Internet facing. For internal sites, there are multiple tools, but the two discussed herein are OpenSSL[9] and sslyze.[10]

> **While there are many subtleties to an HTTPS connection and the configuration can become quite complex, assessing the details of the configuration is actually relatively simple.**

OpenSSL

For spot testing, OpenSSL is the most direct approach. On Linux and related operating systems, OpenSSL is likely already installed. If it is not, it is readily available as an additional package from the OS's package repository. For Windows, one has the option of compiling it for oneself or downloading a compiled version from a third party. Unless there is a specific need to compile OpenSSL, it is strongly encouraged to go the third-party route. To this end, *https://indy.fulgan.com/SSL/* is recommended. On a daily basis, the site owner compiles all versions of OpenSSL, dating back to v0.9.8r. After downloading the version of choice and unzipping the file, one is ready to start testing using OpenSSL's *s_client* command.

For the most basic usage, the following command should be issued:

openssl s_client -connect duckduckgo.com:443

A port must always be specified, even if it is the default HTTPS port of 443.

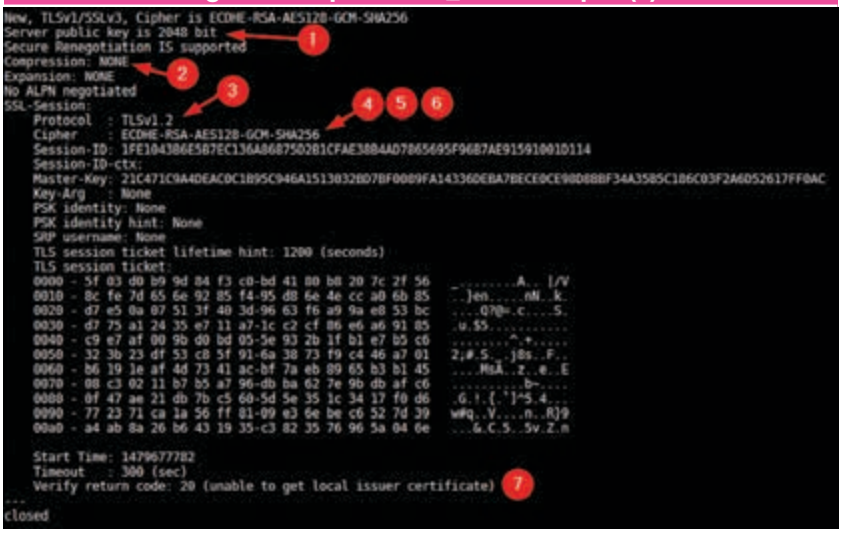
By default, OpenSSL tries its highest level of encryption options first. As a result, this is generally a quick test of the maximum encryption level supported by the server. The previous command produces a modest amount of output, with the most useful part at the end (**figure 2**).

The output shown in **figure 2** indicates that https://duckduckgo.com:

1. Uses a 2,048-bit key
2. Has compression disabled
3. Supports TLSv1.2
4. Uses cipher suites that support Forward Secrecy (as indicated by the DHE)
5. Uses AES cipher suites in GCM mode, thus addressing block padding issues
6. Uses cipher suites that support SHA256
7. Does not include the root CA in the trust chain. If the root CA had been present in the trust chain, the Verify line at the end would have read: `Verify return code: 19 (self signed certificate in certificate chain).`

   Also, the "unable to get local issuer" message is expected, given the way the command is executed. OpenSSL was not provided with a list

Figure 2—OpenSSL *s_client* Output (1)



Source: K. Kincaid. Reprinted with permission.

of trusted certificate issuers against which to compare the server certificate.

Thus, in a single command, seven of the nine configuration points from the configuration checklist discussed in the previous section have been confirmed. What the command does not indicate, however, is what else is permitted by the server. True, in this case the secure features are enabled, but that does not preclude the possibility that something very insecure may be configured as well. Fortunately, the *s_client* command allows one to be much more tactical about connections to the server.

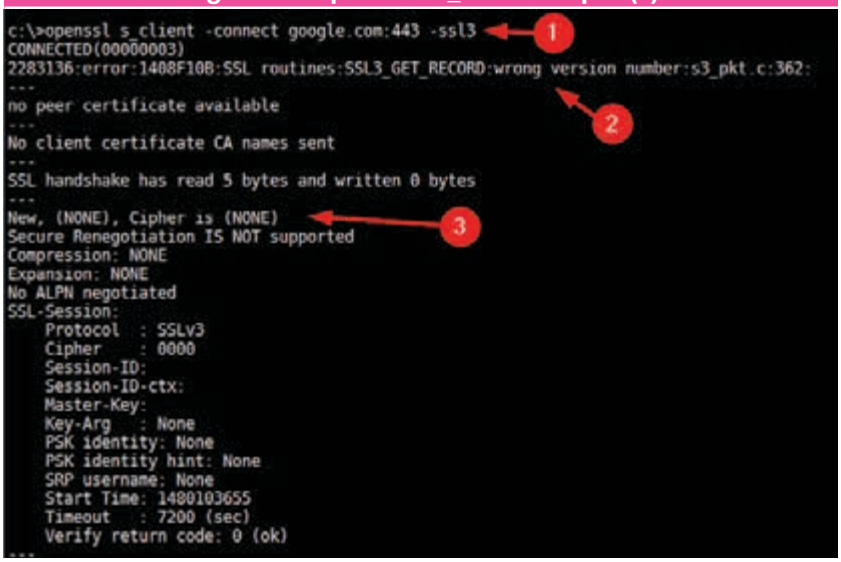
> **"While encrypting everything may not actually be feasible, the concept has merit."**

Among the various *s_client* options, it is possible to specify the protocol version to use. The option is exclusive; for example, if TLSv1.2 is specified, OpenSSL will connect with only TLSv1.2 and will not negotiate with other versions. The syntax of the command is nearly identical to what was shown previously, with the addition of the protocol version.

The output shown in **figure 3** demonstrates:

1. The protocol version to use is specified. The options are:
   - -ssl2
   - -ssl3
   - -tls1
   - -tls1_1
   - -tls1_2
2. The "wrong version number" error is indicative of a protocol version mismatch between client and server. This implies that the client has requested a protocol version that the server does not support.
3. There is further evidence of the failed connection by the presence of "NONE" for the connection protocol and for the cipher suite.

Figure 3—OpenSSL *s_client* Output (2)



Source: K. Kincaid. Reprinted with permission.

The command shows that Google.com does not support SSLv3. By repeating the command with the various protocol versions, one is able to quickly determine which versions are supported by the server.

SSLYZE

For more detail on how the server is configured, sslyze is the perfect tool for the job. It connects to the server and walks through all of the connection options and, from the command line, provides much of the same details provided by the SSL Labs site.

The tool is written in Python, but binary executable versions are also available for Windows. There are many connection options, but for a fairly thorough assessment of the target server, the command syntax may look like the following (this should be on a single line):

sslyze --resum --reneg --http_
headers --compression --heartbleed --certinfo_full
--sslv2 --sslv3 --tlsv1 --tlsv1_1 --tlsv1_2 --hide_
rejected_ciphers --ca_file=C:\myPrivateTrustChain.
pem --json_out mySslyzeOutput.txt myServer:443

In addition to the specified checks, the private trust chain has also been provided. This is particularly useful if the server is using a certificate from an internal public key infrastructure (PKI). Condensed output will be delivered to the screen, while exhaustive output (including all of the cipher suites that were rejected for each protocol, etc.) will be written to a JavaScript Object Notation (JSON) file for later analysis. Even though the screen output is condensed compared to the JSON output, it is still rather lengthy. **Figure 4** shows a small sample of the output.

One caveat with using both SSL Labs and sslyze is that both tools will easily make more than 100 connections to the target server. While the actual burden on the server is minimal, if the servers are being monitored closely for connection errors—for example, with a digital commerce site—both tools will definitely show up on the radar. Use with caution and only after getting approval from management to proceed. OpenSSL, on the other hand, while not as comprehensive, can confirm support for all of the protocol versions with as few as five connections.

Conclusion

At the recent Thales HSM User Conference, one of the phrases used several times over the course of four days was "encrypt everything." While encrypting everything may not actually be feasible, the concept has merit. First, it greatly simplifies things from a policy perspective. No longer are there gray areas requiring a decision about whether encryption is required in this particular case or not. The answer to that question would always be "yes." Simple.

Second, and even more important, it gets everyone thinking not in terms of if, but in terms of how. The difference is subtle, but important. Part of answering the question of how is answering the question of how to do it correctly. After all, if one is not taking the time to do it correctly, it raises the question of how seriously the subject is being taken in the first place. Experience shows there is little difference between the amount of effort required to implement encryption with shortcuts and half-measures versus to implement it correctly.

Once implemented, it is vital to verify and monitor. There are nuances to the configuration of HTTPS that server and application owners cannot be expected to know. Unless they have had reason to spend a significant amount of time learning the inner workings of SSL/TLS configuration, their

Figure 4—Sslyze Condensed Output

```
[...]

SCAN RESULTS FOR MYSERVER:443 - XX.XX.XX.XX:443
 ---------------------------------------------------

  * TLSV1_2 Cipher Suites:
      Preferred:
        TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384        ECDH-256 bits  256 bits

      Accepted:
        TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384        ECDH-256 bits  256 bits
        TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384        ECDH-256 bits  256 bits
        TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA           ECDH-256 bits  256 bits
        TLS_RSA_WITH_AES_256_GCM_SHA384              -              256 bits
        TLS_RSA_WITH_AES_256_CBC_SHA256              -              256 bits

  * TLSV1_1 Cipher Suites:
      Preferred:
        TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA           ECDH-256 bits  128 bits

      Accepted:
        TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA           ECDH-256 bits  256 bits
        TLS_RSA_WITH_AES_256_CBC_SHA                 -              256 bits
        TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA           ECDH-256 bits  128 bits
        TLS_RSA_WITH_AES_128_CBC_SHA                 -              128 bits
        TLS_RSA_WITH_3DES_EDE_CBC_SHA                -              112 bits

  * SSLV3 Cipher Suites:
      Server rejected all cipher suites.

  * Session Renegotiation:
      Client-initiated Renegotiation:    OK - Rejected
      Secure Renegotiation:              OK - Supported

  * Deflate Compression:
                                         OK - Compression disabled
[...]
```

Source: K. Kincaid. Reprinted with permission.

primary concern is delivery, not ensuring that it is done using the ideal secure configuration. As a result, most often, a default configuration is used and it is nearly axiomatic that a default configuration is never an ideal secure configuration. It is important to remember that it is more likely to find flaws in the implementation than in the cryptography itself.[11]

Using the methods described here, it is possible to ensure that an HTTPS posture is secure. Routine reassessment enables an organization to make sure it stays that way and allows prompt detection of those servers that may have slipped out of compliance due to misconfiguration or changes to security requirements. As vulnerabilities are discovered and there are changes to the security landscape itself—such as the release of TLSv1.3—these methods allow stakeholders to adapt to the changing environment and ensure that the HTTPS posture remains strong.

Endnotes

1 Ristic, I.; *Bulletproof SSL and TLS*, Feisty Duck, USA, 2014
2 PCI Security Standards Council, Document Library, *https://www.pcisecuritystandards.org/document_library*
3 Qualys SSL Labs, *https://www.ssllabs.com/*
4 *Op Cit,* Ristic
5 *Ibid.*
6 Moeller, B.; A. Langley; *TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks*, Internet Engineering Task Force, April 2015, *https://tools.ietf.org/html/rfc7507*
7 *Op cit,* Qualys SSL Labs
8 *Op cit,* Ristic
9 Open SSL, *https://openssl.org*
10 Github.com, nabla-c0d3/sslyze, *https://github.com/nabla-c0d3/sslyze*
11 Kohno, T.; N. Ferguson; B. Schneier; *Cryptography Engineering: Design Principles and Practical Applications*, Wiley, USA, 2010