The TLS protocol implementation recommendations are listed in RFC 5256, specifically “Appendix D: Implementation Notes”. Below is a summary table:

|  |
| --- |
| **TLS 1.2, Appendix D: Implementation Notes** |
| **REFERENCE** | **REQUIREMENT from RFC 5256: TLS 1.2** | **EXAMPLE** |
| D.1. Random Number Generation and Seeding | TLS requires a cryptographically secure pseudorandom number generator (PRNG). **Care must be taken in designing and seeding PRNGs**. PRNGs based on secure hash operations, most notably SHA-1, are acceptable, but cannot provide more security than the size of the random number generator state**.** | Not using large seed pool for **r***a*nd0m generator. |
| D.2. Certificates and Authentication | Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). **The selection and addition of trusted CAs should be done very carefully.** Users should be able to view information about the certificate and root CA. | Configure Tableau Server with a valid, **trusted certificate (not a self-signed certificate)** so that Tableau Desktop, mobile devices, and web clients can connect top the server over a secured connection (Tableau, 2018). |
| D.3. Cipher Suites | TLS supports a range of key sizes and security levels, **including some that provide no or minimal security**. **A proper implementation will probably not support many cipher suites.**  | For instance, anonymous Diffie-Hellman is strongly discouraged because it cannot prevent man-in-the-middle attacks. Applications should also enforce minimum and maximum key sizes. For example, certificate chains containing 512-bit RSA keys or signatures are not appropriate for high-security applications (EITF, 2008). |
| D.4. Implementation Pitfalls “TLS PROTOCOLS” | Do you correctly handle handshake messages that are fragmented to multiple TLS records (see Section 6.2.1)? Including corner cases like a ClientHello that is split to several small fragments? Do you fragment handshake messages that exceed the maximum fragment size? In particular, the certificate and certificate request handshake messages can be large enough to require fragmentation. | Length: The length (in bytes) of the following TLSPlaintext.fragment. **The length MUST NOT exceed 2^14.**Fragment: Implementations **MUST NOT send zero-length fragments of Handshake, Alert, or ChangeCipherSpec content types**. Zero-length fragments of Application data MAY be sent as they are potentially useful as a traffic analysis countermeasure. |
| Do you ignore the TLS record layer version number in all TLS records before ServerHello (see Appendix E.1)? | If **the version chosen by the server is not supported by the client (or not acceptable**), **the client MUST send a "protocol\_version" alert** **message and close the connection.**If a TLS server receives a ClientHello containing a version number greater than the highest version supported by the server, **it MUST reply according to the highest version supported by the server.**Whenever a client already knows the highest protocol version known to a server (for example, when resuming a session), **it SHOULD initiate** **the connection in that native protocol.**Earlier versions of the TLS specification were not fully clear on what the record layer version number (TLSPlaintext.version) should contain when sending ClientHello (i.e., before it is known which version of the protocol will be employed). Thus, **TLS server compliant with this specification MUST accept any value {03,XX} as the record layer version number for ClientHello.** |
| Do you handle TLS extensions in ClientHello correctly, including omitting the extensions field completely? | **An extension type MUST NOT appear in the ServerHello** unless the same extension type appeared in the corresponding ClientHello.When multiple extensions of different types are present in the ClientHello or ServerHello messages, the extensions MAY appear in any order. **There MUST NOT be more than one extension of the same type.**Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, **but extreme care is needed** when the extension changes the meaning of messages sent in the handshake phase. **Designers and implementors should be aware of the fact that until the handshake has been authenticated**, active attackers can modify messages and insert, remove, or replace extensions.**It would be technically possible to use extensions to change major aspects of the design of TLS; for example the design of cipher suite negotiation. This is not recommended;** it would be more appropriate to define a new version of TLS -- particularly since the TLS handshake algorithms have specific protection against version rollback attacks based on the version number, and the possibility of version rollback should be a significant consideration in any major design change, |
| Do you support renegotiation, both client and server initiated? While renegotiation is an optional feature, supporting it is highly recommended. | no\_renegotiation: Sent by the client in response to a hello request or by the server in response to a client hello after initial handshaking. Either of these would normally lead to renegotiation; when that is not appropriate, the recipient should respond with this alert.One case where this would be appropriate is where a server has spawned a process to satisfy a request; **the process might receive security parameters (key length, authentication, etc**.) |
| When the server has requested a client certificate, but no suitable certificate is available, do you correctly send an empty Certificate message, instead of omitting the whole message (see Section 7.4.6)? | **1.2. Major Differences from TLS 1.1**After a certificate request, if no certificates are available, **clients now MUST send an empty certificate list.** |
| D.4. Implementation Pitfalls “CRYPTOGRAPHIC DETAILS” | In the RSA-encrypted Premaster Secret, do you correctly send and verify the version number? When an error is encountered, do you continue the handshake to avoid the Bleichenbacher attack (see Section 7.4.7.1)? | **7.3. Handshake Protocol Overview**Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret… Generate a master secret from the premaster secret and exchanged random values.**7.4.7.1. RSA-Encrypted Premaster Secret Message**If RSA is being used for key agreement and authentication, the client generates a 48-byte premaster secret, encrypts it using the public key from the server's certificate, and sends the result in an encrypted premaster secret message. This structure is a variant of the ClientKeyExchange message and is not a message in itself.**client\_version:****The latest (newest) version supported by the client. This is used to detect version rollback attacks.**Structure of this message: *struct {* *ProtocolVersion client\_version;* *opaque random[46];* *} PreMasterSecret;* Client implementations **MUST always send the correct version number in** **PreMasterSecret.** If ClientHello.client\_version is TLS 1.1 or higher, server implementations **MUST check the version number as described in** **the note below.**  If the version number is TLS 1.0 or earlier, server implementations SHOULD check the version number, but MAY have a configuration option to disable the check. Note that if the check fails, the PreMasterSecret SHOULD be randomized as described below. Note: Attacks discovered by Bleichenbacher [BLEI] and Klima et al. [KPR03] can be used to attack a TLS server that reveals whether a particular message, when decrypted, is properly PKCS#1 formatted, contains a valid PreMasterSecret structure, or has the correct version number. As described by Klima [KPR03], these vulnerabilities can be avoided **by treating incorrectly formatted message blocks and/or mismatched** **version numbers in a manner indistinguishable from correctly** **formatted RSA blocks**. In other words: 1. Generate a string R of 46 random bytes 2. Decrypt the message to recover the plaintext M 3. If the PKCS#1 padding is not correct, or the length of message M is not exactly 48 bytes: pre\_master\_secret = ClientHello.client\_version || R else If ClientHello.client\_version <= TLS 1.0, and version number check is explicitly disabled: pre\_master\_secret = M else: pre\_master\_secret = ClientHello.client\_version || M[2..47 |
| What countermeasures do you use to prevent timing attacks against RSA decryption and signing operations (see Section 7.4.7.1)? | **7.4.7.1. RSA-Encrypted Premaster Secret Message**Implementation note: It is now known that remote timing-based attacks on TLS are possible, at least when the client and server are on the same LAN. Accordingly, implementations that use static RSA keys **MUST** **use RSA blinding or some other anti-timing technique**, as described in [TIMING]. **From [TIMING]:** “Two other possible defenses are suggested often, but are a second choice to blinding. The ﬁrst is to try and **make all RSA decryptions not dependent upon the input ciphertext**... If an extra reduction is not needed, we carry out a “dummy” extra reduction and do not use the result…. Another alternative is to require all **RSA computations to be quantized**, i.e. always take a multiple of some predeﬁned time quantum…Currently, the preferred method for protecting against timing attacks is to use RSA blinding. The immediate drawbacks to this solution is that a good source of randomness is needed to prevent attacks on the blinding factor, as well as the small performance degradation” (Brumley & Boneh, 2003). |
| When verifying RSA signatures, do you accept both NULL and missing parameters (see Section 4.7)? Do you verify that the RSA padding doesn't have additional data after the hash value? [FI06] | **1.2. Major Differences from TLS 1.1**Substantial cleanup to the client's and server's ability to specify which hash and signature algorithms they will accept. Note that this also relaxes some of the constraints on signature and hash algorithms from previous versions of TLS.**4.7. Cryptographic Attributes**For hash algorithms without parameters (which includes SHA-1), the DigestInfo.AlgorithmIdentifier.parameters field MUST be NULL, but **implementations MUST accept both without parameters and with NULL parameters.** Note that earlier versions of TLS used a different RSA signature scheme that did not include a DigestInfo encoding.**From [FI06]:**The signature verifier first applies the RSA public exponent to revealthis PKCS-1 padded data, checks and removes the PKCS-1 padding, thencompares the hash with its own hash value computed over the signed data.…The error that Bleichenbacher exploits is if the implementation doesnot check that the hash+ASN.1 data is right-justified within the PKCS-1padding. Some implementations apparently remove the PKCS-1 padding bylooking for the high bytes of 0 and 1, then the 0xFF bytes, thenthe zero byte; and then they start parsing the ASN.1 data and hash.The ASN.1 data encodes the length of the hash within it, so this tellsthem how big the hash value is**.** These broken implementations go aheadand use the hash, without verifying that there is no more data after it.Failing to add this extra check makes implementations vulnerable to asignature forgery.…**Daniel also recommends that people stop using RSA keys with exponents****of 3.** (Finney, 2006). |
| When using Diffie-Hellman key exchange, do you correctly strip leading zero bytes from the negotiated key (see Section 8.1.2)? | **8.1.2. Diffie-Hellman**Leading bytes of Z that contain **all zero bits are stripped before it is used as the pre\_master\_secret.****From Microsoft:** *“This results in the bad mac error when client sends the finished message. I have confirmed that the Finished message decrypts properly if I leave the leading zero in the Premaster secret (pad the value to the modulus size of the DH group)”* (Doug, 2016). |
| Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable (see Section F.1.1.3)? | **F.1.1.3. Diffie-Hellman Key Exchange with Authentication**When Diffie-Hellman key exchange is used, **the server can either supply a certificate containing fixed Diffie-Hellman parameters or use the server key exchange message to send a set of temporary Diffie-Hellman parameters signed with a DSA or RSA certificate.** Temporary parameters are hashed with the hello.random values before signing to ensure that attackers do not replay old parameters. In either case, the client can verify the certificate or signature to ensure that the parameters belong to the server.If the client has a certificate containing fixed Diffie-Hellman parameters, its certificate contains the information required to complete the key exchange. Note that in this case the client and server will generate the same Diffie-Hellman result (i.e., pre\_master\_secret) every time they communicate. To prevent the pre\_master\_secret from staying in memory any longer than necessary, it should be converted into the master\_secret as soon as possible. **Client Diffie-Hellman parameters must be compatible with those supplied by the server for the key exchange to work.**If the same DH keypair is to be used for multiple handshakes, either because the client or server has a certificate containing a fixed DH keypair or because the server is reusing DH keys, **care must be taken to prevent small subgroup attacks. Implementations SHOULD follow the guidelines found in [SUBGROUP**]. Small subgroup attacks are most **easily avoided by using one of the** **DHE cipher suites and generating a fresh DH private key (X) for each** **handshake**. If a suitable base (such as 2) is chosen, g^X mod p can be computed very quickly; therefore, the performance cost is minimized. Additionally, using a fresh key for each handshake provides Perfect Forward Secrecy. Implementations SHOULD generate a new X for each handshake when using DHE cipher suites.Because TLS allows the server to provide arbitrary DH groups, the client should verify that the DH group is of suitable size as defined by local policy. The client SHOULD also verify that the DH public exponent appears to be of adequate size. [KEYSIZ] provides a useful guide to the strength of various group sizes. The server MAY choose to assist the client by providing a known group, such as those defined in [IKEALG] or [MODP]. These can be verified by simple comparison.**From [SUBGROUP] Protection Considerations:** * Public Key Validation
* CA Performs Public Key Validation
* Choice of Prime p
* Compatible Cofactor Exponentiation
* Non-compatible Cofactor Exponentiation
 |
|  | How do you generate unpredictable IVs for CBC mode ciphers (see Section 6.2.3.2)? | **The Initialization Vector (IV) SHOULD be chosen at random, and MUST be unpredictable.** Note that in versions of TLS prior to 1.1, there was no IV field, and the last ciphertext block of theprevious record (the "CBC residue") was used as the IV. This was changed to prevent the attacks described in [CBCATT]. For block ciphers, the IV length is of length **SecurityParameters.record\_iv\_length, which is equal to the SecurityParameters.block\_size..****[D.1](https://tools.ietf.org/html/rfc5246%22%20%5Cl%20%22appendix-D.1). Random Number Generation and Seeding**[RANDOM] provides guidance on the generation of random values: BCP 106, RFC 4086, |
|  | Do you accept long CBC mode padding (up to 255 bytes; see Section 6.2.3.2 | **6.2.3.2. CBC Block Cipher**paddingPadding that is added to force the length of the plaintext to be an integral multiple of the block cipher's block length. The padding **MAY be any length up to 255 bytes, as long as it results in the TLSCiphertext.length being an integral multiple of the block length. Lengths longer than necessary might be desirable to frustrate attacks on a protocol that are based on analysis of the lengths of exchanged messages.**  Each uint8 in the padding data vector MUST be filled with the padding length value. The receiver MUST check this padding and MUST use the bad\_record\_mac alert to indicate padding errorspadding\_lengthThe padding length MUST be such that the total size of the GenericBlockCipher structure is a multiple of the cipher's block length. Legal values range from zero to 255, inclusive. This length specifies the length of the padding field exclusive of the padding\_length field itself |
|  | How do you address CBC mode timing attacks (Section 6.2.3.2)? | **6.2.3.2. CBC Block Cipher**Implementation note: Canvel et al. [CBCTIME] have demonstrated timing attack on CBC padding based on the time required to compute the MAC. In order to defend against this attack, **implementations MUST ensure that record processing time is essentially the same whether or not the padding is correct.** In general, the best way to do this is to compute the MAC even if the padding is incorrect, and only then reject the packet. For instance, if the pad appears to be incorrect, the implementation might assume a zero-length pad and then compute the MAC. This leaves a small timing channel, since MAC performance depends to some extent on the size of the data fragment, but it is not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal. |
|  | Do you use a strong and, most importantly, properly seeded random number generator (see Appendix D.1) for generating the premaster secret (for RSA key exchange), Diffie-Hellman private values, the DSA "k" parameter, and other security-critical values? | **D.1. Random Number Generation and Seeding**[RANDOM] provides guidance on the generation of random values.**See RFC 4086**1. Entropy sources
2. De-skewing
3. Mixing
4. Pseuedo-random number generators
 |

**References**

Brumley, D. Boneh, D. (2003). *Remote Timing Attacks are Practical.* *In proceedings of the 12th Usenix Security Symposium, 2003*. Retrieved from https://crypto.stanford.edu/~dabo/papers/ssl-timing.pdf

Doug, S. (2016, October 10). *TLS DH handshake fails when Premaster secret contains leading zeros.* https://developer.microsoft.com/en-us/microsoft-edge/platform/issues/9285068/

Finney, H. (2006, August 27). *Bleichenbacher's RSA signature forgery based on implementation error.* Retrieved from https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html

IETF. (2005, June). BCP 106: *Randomness Requirements for Security*. Retrieved from https://tools.ietf.org/html/bcp106

IETF. (2008, August). RFC 5246: *The Transport Layer Security (TLS) Protocol Version 1.2.* Retrieved from https://tools.ietf.org/html/rfc5246

OWASP. (2018, June 14). *Transport Layer Protection Cheat Sheet.* Retrieved from https://www.owasp.org/index.php/Transport\_Layer\_Protection\_Cheat\_Sheet#Server\_Certificate

Tableau. (2018, July 11). *Security Hardening Checklist.* Retrieved from https://onlinehelp.tableau.com/current/server/en-us/security\_harden.htm