Implementation Flaws in TLS Stacks: Lessons Learned and Study of TLS 1.3 Benefits

Olivier Levillain
olivier.levillain@telecom-sudparis.eu

Télécom SudParis, Institut Polytechnique de Paris

Abstract. In the years leading to the definition of TLS 1.3, many vulnerabilities have been published on the TLS protocol, including numerous implementation flaws affecting a wide range of independent stacks. The infamous Heartbleed bug, was estimated to affect more than 20 % of the most popular HTTPS servers. We propose a structured review of these implementation flaws. By considering their consequences but also their root causes, we present some lessons learned or yet to be learned. We also assess the impact of TLS 1.3,the latest version of the protocol, on the security of SSL/TLS implementations.

SSL (Secure Sockets Layer) is a cryptographic protocol designed by Netscape in 1995 to protect the confidentiality and integrity of HTTP connections, mainly to secure online commercial or financial operations. Since 1999, the protocol has been maintained by the IETF (Internet Engineering Task Force) and has been renamed TLS (Transport Layer Security). In this article, when referring to the protocol in general, we will use the SSL/TLS denomination.

SSL/TLS has now become an essential part of Internet security. The most recent version of the protocol is TLS 1.3 [24], which was published after more than 5 years of discussion within the IETF TLS Working Group.

In this paper, we do not address the question of the theoretical security of the protocol and of its underlying cryptographic mechanisms, but the actual security of the implementations. Indeed, because of bugs or more subtle quirks, there may be unexpected vulnerabilities whose consequences may go beyond the security of the considered transaction or service. We thus studied in depth many known implementation flaws in SSL/TLS stacks to understand their root causes and to propose ways to improve the situation. Indeed some mistakes keep being repeated and affect independent software stacks; this led us to question whether blaming the developers for ignoring the "state of the art" was the right answer.

First, we look at simple, classical programming errors such as buffer overflows or logic errors in Section 1. The second category, in Section 2, is about parsing bugs, which are sometimes the result of the complexity of the structures to interpret. Next, we look at cryptography-related vulnerabilities in Section 3, which are sometimes presented as mistakes from the developers, whereas we strongly believe that obsolete cryptographic primitives are the underlying cause in many cases, leaving the developers who must choose between code modularity and security in a difficult position. The last category, described in Section 4, is about issues in SSL/TLS state machines. It is indeed sometimes possible to get an implementation into an invalid state, where it would accept unsolicited or meaningless messages, which can lead to catastrophic consequences for the security of the communications. Each section ends with a decription of the lessons we can learn from the presented problems, as well as a discussion about what TLS 1.3 has brought to improve the situation on the subject.

1 Common programming errors

1.1 CVE-2014-1266: Apple's `goto fail`

In February 2014, Apple released a security advisory, indicating that an attacker could bypass the server authentication mechanism on the client side in its operating systems. In the vulnerable function (see listing in appendix A), a `goto fail` instruction was wrongly duplicated, which means that the actual verification of the server signature (the `sslRawVerify` call) was skipped. So, when the client took that code path (as soon as a Diffie-Hellman ciphersuite was negotiated), the server signature over the parameters was *not* checked and the server was automatically authenticated.

An attacker could thus simply impersonate a TLS server by forcing the use of a vulnerable suite in the `ServerHello` message, then present the legitimate certificates, and finally send an arbitrary `ServerKeyExchange` message, since its authenticity would not be checked: from the client's point of view, the server certificate was however correctly validated, leading to an authentication bypass.

The problem was quickly and easily fixed, but it showed that the corresponding code had not been sufficiently tested or checked using static analysis, since such a trivial case of dead code should have been detected. Such dead code can indeed be detected with modern compilers such as `clang` with `-Wunreachable-code`[1].

After this bug, people started to advocate for better compilers, analysis tools, or even for the use of safer alternative languages instead of the C language. Several commentators explained that this bug would have been avoided, had the developers used curly brackets for their `if` statements. Yet, without tools enforcing such practice, we would still have no guarantee; and if we were using tools, it would be safer to have them identify the root cause (dead code).

1.2 CVE-2014-0092: GnuTLS' `goto fail`

A few days after Apple's `goto fail`, GnuTLS released another advisory concerning a vulnerability in the code checking signatures. The issue was (a little) subtler, about the `check_if_ca` function checking whether a certificate was from a certification authority (CA), and the `_gnutls_verify_certificate2` function checking the signature. Even though the documentation stated these functions returned a boolean value (that is 0 or 1), they could actually also return a negative value in some parsing error cases.

So, when critical functions, such as `gnutls_x509_crt_check_issuer`, called those functions treating the result as a boolean, they would reject certificates with invalid signature, accept valid certificates, but also *accept ill-formed certificates*. Listings in appendix B contain the relevant code excerpts.

[1] However, `gcc` has been silently ignoring this option since version 4.5.

GnuTLS developers fixed the corresponding logic in a defensive way, on the one hand by insuring the called functions indeed returned only `0` or `1`, and on the other hand by having the calling functions check the result in a stricter way.

It is worth noting that a similar bug had already been found six years earlier in OpenSSL (CVE-2008-5077) (except that the bug was triggered by the code parsing the signature block, not the certificate). There again, the question of the used tools and languages was raised.

1.3 CVE-2002-0862 and CVE-2011-0228: `BasicConstraints` checks.

To distinguish a CA certificate from a server certificate, the X.509 standard requires relying parties to check the `BasicConstraints` extension. This X.509 extension contains a boolean, `cA`, which should only be true when the certificate belongs to a certification authority. Valid X.509 stacks must in particular check this information when checking a certificate chain. The impact of this boolean is critical, since a certificate with a true `cA` boolean allow, in the common cases, its owner to issue arbitrary certificates for arbitrary domain names.

In 2002, Marlinspike showed that Internet Explorer did not actually check this boolean [19], allowing an attacker to reuse any certificate she possesses (even a simple SSL server one) to sign new arbitrary certificates. This was a trivial yet critical bug. What is more interesting is that the same bug reappeared in 2011 in Apple's TLS stack [21], a different, independent, TLS stack. In both cases, one can at least question the test process.

1.4 CVE-2014-0160: Heartbleed

In April 2014, a devastating vulnerability in OpenSSL was presented – a "common" buffer overrun in the Heartbeat implementation.

Heartbeat is a TLS extension. When it has been negotiated, the client or the server can, at any time, send a Heartbeat record containing data, and the recipient has to echo those data back. Such a mechanism can theoretically be used for two purposes: Path MTU (i.e. maximal packet size) Discovery and a secure Keep Alive mechanism. In practice, both goals are mostly relevant to DTLS, the datagram version of TLS, which can be used over UDP. Yet the Heartbeat extension was integrated into OpenSSL for both DTLS *and* TLS on December 31st 2011[2], and activated by default.

When a vulnerable version of OpenSSL received a Heartbeat request advertising a content longer than the sent payload, it filled the response with the received content and whatever was present afterwards in memory. All kinds of heap-allocated data could be endangered: contents of communications between *other clients* and the server, authentication cookies, user passwords, and even the server's private key. In practice, Durumeric et al. estimated that 24 to 55 % of the most popular HTTPS sites were affected by Heartbleed [13].

[2] The first official OpenSSL version containing the Heartbeat extension was 1.0.1, published on March 14th 2012.

Fixing the code was trivial, but not handling all the possible consequences. It is hard to say whether or not this vulnerability was known and exploited, but the precautionary principle advocated e.g. for revocation of millions of vulnerable server certificates.

1.5 CVE-2014-6321: WinShock

In November 2014, Microsoft published a security advisory, MS14-066, with multiple vulnerabilities. One of them, dubbed WinShock, was about a buffer overflow in SChannel, Microsoft's TLS stack.

The flaw was located in the `DecodeSigAndReverse` function, which parses ECDSA signatures in the context of client certificate authentication. When handling a client certificate using elliptic-curve cryptography, the certificate, present in the `Certificate` message, is first parsed. The server extracts the public key the elliptic curve used for the signature, either using well known identifiers (several curves can be identified by ASN.1 Object Identifiers) or in an explicit manner (the description then contains the underlying finite field and the curve equation). In both cases, the public key sets the coordinate size. In a second phase, the signature is read from the `CertificateVerify` message, which contains two scalars whose size is given by the curve.

In SChannel, the vulnerable code allocated memory regions for the signature, based on the curve description from the certificate, then read the data from the signature, using this time the encoded ASN.1 length of the signature, *without checking the consistency between both lengths*. It was thus possible to trigger a buffer overflow using long coordinates within a crafted signature in the `CertificateVerify` message. Smashing data in the heap using this vulnerability was proven exploitable in IIS servers: proofs of concept could even lead to remote code execution on vulnerable systems.

In a typical configuration, client certificates are rarely used. However, even unsolicited `Certificate` and `CertificateVerify` were parsed by SChannel, which means an attacker could trigger the vulnerability in any SChannel deployment. So this is a second bug affecting the state machine (more on similar bugs in Section 4).

1.6 Lessons learned

When one look at these vulnerabilities, one might be tempted to blame the developers for making so many mistakes, and for repeating them over and over.

At the same time, it seems the languages or tools used for the development could and should have been more helpful, e.g. by offering a *real* boolean type[3] or by warning for trivial errors such as having obvious dead code in a function.

One way for hardening the code is to configure the tools to be as strict as possible. For example, in C, one can require additional checks using `-Wall -Wextra -Werror` and other similar options at all times.

Some programming languages are better than others to avoid whole classes of bugs, but there is no silver bullets, and it is essential to always understand exactly what you are trading in exchange for what. For example, languages with garbage collector eradicate

[3] The C99 standard introduced such a type, but it seems almost no one makes use of it.

memory management error such as use-after-free and double-free. Yet having no fine-grain control on memory management means that you cannot easily ensure that secrets (private keys, passwords) are erased as soon as possible from the memory, and actually, they could even be copied at multiple memory locations several times by the garbage collector during their lifetime.

Once developers understand languages constructions and their behavior, they can adopt more robust structures, e.g. always err on the safe side as this was done by GnuTLS developers to fix the `goto fail` bug, by only considering `1` as the valid case, instead of *everything that is not* `0`. Another example is the use of bound-checking arrays, where each read or write access is checked at runtime to stay within the array boundaries (as in Java, OCaml or Rust). Even if this kind of mechanisms induces a small overhead, it is a good way to avoid buffer overflows. Yet, developers should be aware that some constructions may evade these safety mechanisms and stay away from unsafe features. Ensuring that third-party libraries behave correctly is also recommended.

All in all, mastering programming languages, choosing adequate compilers and tools and correctly using them can help improve code quality and avoid several classes of software bugs.

A word about tests Another best practice is *negative* testing: when security is involved, it is not sufficient to check that what should work works, it is crucial to also check that what should not work does actually not. Checking that a Handshake fails as expected when a signature is invalid would have prevented Apple's `goto fail` vulnerability.

Moreover, since programmers seem to make the same mistakes time and again in different code bases, it might be useful to build a collection of *negative* and non-regression tests to share between implementations. In this domain, the situation has slightly improved, with tools such as tlsfuzzer[4] and TLS-Attacker[5].

TLS 1.3 benefits With regards to languages and tools, TLS 1.3 does not bring anything, since the RFC is a specification. Moreover, the IETF always insisted on letting developers be free to make implementation choices.

2 Parsing bugs

In the previous section, we studied classical simple errors such as memory management issues. Such bugs can also arise in the parser code. Beyond these somewhat common bugs, parsers trigger another class of vulnerabilities, when the parsed content does not correspond to its intended value. Such bugs can result from a confusion in the specification or a lack of precision in the parsing code.

[4] `https://github.com/tomato42/tlsfuzzer`

[5] `https://github.com/RUB-NDS/TLS-Attacker`

2.1 CVE-2009-2408: Null characters in distinguished names

In 2009, Marlinspike presented several bugs in TLS stacks leading to authentication bypass [20]. In particular, he presented a difference of behavior between several X.509 implementations in the presence of null characters.

ASN.1 specifications are clear on the subject: the length of a string is explicitly set by a separate field, and most ASN.1 string types do not allow for null characters. Yet, several browsers, e.g. Firefox, actually accepted and interpreted null characters as the end of the string, leading to an alternate interpretation.

Let's consider an attacker requesting a certificate for the `www.mybank.com\0.evil.com` domain, where `evil.com` is controlled by the attacker and `\0` is the null character. Moreover, we assume the contacted CA simply extracts the top-level domain `evil.com` and sends a validation email to `postmaster@evil.com`. Under these assumptions, the attacker can get their certificate. The provided certificate could then be used against vulnerable browsers to impersonate `www.mybank.com`.

Beyond the obvious misinterpretation from browsers, which should not rely on null characters to end ASN.1 DER strings, there is another bug: the CA should not have accepted ill-formed data as part of a fully-qualified domain name in the first place. This example shows that, as soon as two implementations do not agree on the interpretation of a given element, there is a gap that an attacker can (and will) exploit.

2.2 CVE-2014-3511: OpenSSL downgrade attack

TLS allows records from the same type to be split and merged in a very liberal way. What is allowed and forbidden is not always clear in the specification. Yet, splitting records is required in some cases, since Handshake messages can be 16 MB long whereas TLS records are limited to 16 KB.

In July 2014, Benjamin and Langley showed that OpenSSL exhibited a strange behavior when it receives a `ClientHello` message split in very small records. When parsing the first `ClientHello` fragment, an implementation needs at least 6 bytes in the record payload to identify the proposed protocol version. In the absence of this information, OpenSSL was not able to extract the proper version and systematically used TLS 1.0 instead of waiting for the rest of the `ClientHello` message. Moreover, since only the *aggregated content* of the records are integrity-protected, the exact way Handshake messages are split can easily be changed by an attacker without detection.

To fix this bug, OpenSSL developers chose to reject tiny `ClientHello` fragments. This is an incorrect behavior with respect to the specification, but the alternative was deemed too complex to implement. We find that the decision is actually relevant, and that the specification should probably contain some constraints to allow for reasonable expectations from the developers.

This attack shows that the complexity of TLS, combined with the need to support several protocol versions, can lead to subtle implementation difficulties. A similar example is given by Bhargavan et al. [7], with the Alert attack, where an attacker can misalign the boundaries of alert messages (which are 2 bytes long) with the records encapsulating them. It is then possible to send one byte in an unprotected alert record that may be interpreted later as an authenticated piece of alert.

2.3 CVE-2014-1568: NSS/CyaSSL/PolarSSL Signature Forgery

In September 2014, another vulnerability allowing to bypass server authentication on several TLS clients was published. The vulnerability affected NSS, the Firefox cryptographic library, as well as CyaSSL and PolarSSL. It takes its root in the code parsing DER-encoded RSA signature. DER is a concrete representation of ASN.1 enforcing normal forms: there should be one and only one correct representation for each abstract value.

It is actually a variant of an attack presented in 2006 by Bleichenbacher. The original vulnerability relied on broken RSA implementations that did not check the absence of data beyond the `DigestInfo` block [9]. In the case of a small public exponent (such as 3), it is easy to forge a signature for such a message, that would be accepted by fuzzy implementations.

The vulnerability presented in September 2014 is another universal, relying on three elements to be exploitable: the attacker needs to find an RSA key with a public exponent equal to 3 (this exponent can be anywhere in the certificate chain she is trying to spoof); the ASN.1 DER parser must be too liberal, i.e. accept non-canonically encoded values; DER length computation can silently overflow. Details can be found in appendix C.

The obvious fix here is to use a strict DER parser. However, it is even possible to avoid the parsing step altogether by re reversing the comparison process while checking a signature: instead of computing $m = s^e$ from the signature s, then *parse* m and finally compare the encompassed hash value inside m, a robust implementation should produce the message $m^\star$ containing the expected `DigestInfo`, then compute $m = s^e$ and compare m to $m^\star$.

By comparing concrete representations instead of abstract ones, we skip the parsing step and the only operations manipulating attacker-controlled data are the s^e computation and the trivial binary comparison. Moreover, since DER is a canonical representation of the abstract value, $m^\star$ is unambiguously defined[6].

2.4 Lessons learned

Despite the important number of implementations affected by the parsing issues described in this section, it would not be fair to conclude that all these bugs were only the result of poor programming practices. Developers obviously bear their share of responsibility, but several errors were also the result of complex or ill-specified protocols and formats.

In particular, parsing attacker-controlled data is an error-prone process that should never be overlooked. As soon as parsing is not straightforward and can lead to ambiguities, security vulnerabilities may arise, either because of different actors interpreting the same messages differently, or because it allows an attacker to tamper with the expected execution path. We must insist that the so-called robustness principle (*Be liberal in what you accept, conservative in what you send*) is a terribly wrong advice regarding

[6] This is actually an approximation, since some implementations still produce ill-formed `DigestInfo` where the algorithm parameters is omitted, instead of being a DER NULL element. To accommodate such pervasive deviations, a robust implementation should thus produce two versions of $m^\star$.

security: it should be replaced by another, simpler, statement: be conservative, always (and report bugs in confusing specifications).

Recipes to improve security would include writing strict parsers, avoid exposing them when possible (e.g. by comparing concrete representations instead of abstract, parsed ones), stress-test the parsers in corner cases. Yet, the real long-term advice is to simplify the specification and to express them using a more formal language, to reduce the possibilities of bugs and ambiguities in the resulting code.

TLS 1.3 benefits From the message parsing point of view, RFC 8446 is similar to the previous TLS specifications, but some problematic cases have been described, to disambiguate corner cases such as the Alert attack discussed earlier (section 5.1 of the RFC 8446 is crystal clear on the encapsulation of alerts within records). However, one might still feel uneasy with Handshake messages or extensions whose exact content depends on the context, which adds unnecessary complexity in the parsers.

3 The real impact of obsolete cryptography on security

SSL/TLS is a rather old protocol, dating back 1995. The cryptography community has since learned a lot about algorithms, schemes and protocols. This knowledge has not always been taken into account in recent versions of the protocol, mostly for compatibility reasons:TLS 1.2 still (partly) relies on PKCS#1 v1.5 encryption, the CBC mode, and the MAC-then-Encrypt paradigm. In this section, we present the implications on implementations of using obsolete cryptography.

3.1 CVE-2013-0169: the dangers of MAC-then-Encrypt

Since its inception, SSL/TLS has been supporting the MAC-then-CBC paradigm to protect its records. This led to Lucky13, an attack using a timing information leak during TLS record decryption as a padding oracle [3]. Even if one may think this flaw is *only* an implementation issue (writing constant-time code to decrypt and check the integrity of a record), we believe the problem runs deeper.

Indeed, when one looks at the complex corresponding patch in OpenSSL [18], one is forced to note that it is a vast amount of complex and intricate calls to hash compression functions and decryption primitives. We have traded a simple and intuitive decrypt/unpad/MAC-check sequence with low-level instructions. Moreover, the portability of the OpenSSL fix is debatable, since Langley had to trick the compiler to avoid low-level optimization related to modular reductions on small integers[7].

This is the reason why researchers (and the TLS 1.3 standard) promote higher-level and secure-by-design constructions, such as AEAD ciphers, to obtain strong guarantees on both the confidentiality and the integrity of the protected data.

A simpler path was even presented in 2001 by Krawczyk [15]: Encrypt-then-MAC, which can be proven to be safer. So, despite the Record Protocol protection was known

[7] In a nutshell, the DIV instruction takes a variable amount of time depending on its argument on Intel CPUs, which could be observable.

to be flawed in 2001, it was only partially fixed in 2008 with TLS 1.2 and the introduction of AEAD constructions.Only TLS 1.3 completely deprecates the flawed CBC mode (and the biased RC4 algorithm), by forcing the use of AEAD algorithms.

The impact on TLS stacks is a difficult choice between straightforward and modular, but flawed, code on the one side, and a complex, hard-to-follow and error-prone, but theoretically sound implementation on the other side.

Considering the difficulty to fix this issue, it is worth looking the story of `s2n`, a TLS implementation released by Amazon [17]. Despite including countermeasures against Lucky13, Albrecht and Paterson presented evidence that the library was nevertheless vulnerable to a weaker, yet still exploitable, form of padding oracle [2]. To avoid writing too low-level code, `s2n` decryption code execution time was indeed not exactly constant.

3.2 CVE-2016-0270: Issues with GCM Nonce Generation

Another way symmetric cryptography can fail is when you do not fulfill the expected assumption. In general, blockcipher modes of operation require the use of parameters, such as IVs (Initial Values) or Nonces, which are required to be unpredictable or unique, depending on the schemes.

In 2016, Böck et al. showed that several HTTPS servers at large reused nonce values, or generated them in a non-optimal way [11]. Indeed, GCM requires the 64-bit nonce used in TLS to be unique. Reusing a value twice fully breaks the authenticity of connections. It is interesting to notice that drawing random values leads to collisions (hence nonce reuse) faster than a simple counter. The correct fix here is to use such a counter

In other schemes, what is important is not uniqueness, but unpredictability, as is the case with the CBC mode, where leaking the next IV to use can lead to real-world attacks such as BEAST [12].

One way to solve the problem is to force the developer to make the right decision. This is why TLS 1.3 mandates how to generate the nonce in a deterministic way: the value is derived by each participant using authenticated information sent on the wire and a shared secret.

3.3 CVE-2014-0411 and others: PKCS#1 v1.5 and Bleichenbacher

A valid PKCS#1 v1.5 message is produced by formatting the plaintext and then encrypting it using the raw RSA operation. The expected format for an encrypted message is the following: a null byte, followed by a block type byte (here, 2), then at least 8 random padding bytes, a null character and finally the message to encrypt (see appendix D for more details).

It thus means that every correctly padded plaintext starts with `00 02`, which corresponds to a big integer between $2 \times 2^{n-16}$ and $3 \times 2^{n-16}$ (with an n-bit modulus). If an attacker wishes to recover the plaintext P associated to a given ciphertext C, she can multiply C by X^e and submit the new ciphertext to a decryption oracle: the padding will be correct as soon as $P \times X$ is between the expected bounds. By iterating such attempts, it is possible to aggregate information about the original plaintext P and recover it, as was shown by Bleichenbacher in 1998 [8] in his so-called Million Message Attack.

The attack is applicable to RSA encryption key exchange in TLS. As described in RFC 3218,there are three classical countermeasures:

- group all possible errors so they lead to a unique signal, where the padding errors are indistinguishable from other errors;
- where possible, ignore all errors silently and replace the decrypted message by a random string (this is what is recommended for RSA encryption key exchange in TLS since version 1.0);
- use PKCS#1 v2.1 encryption (OAEP).

Even if the Million Message Attack has been known since 1998, it is still a problem in recent TLS implementations.The Bleichenbacher attack resurfaced in the JSSE (Java Secure Socket Extension) SSL/TLS implementation [23]: by reusing standard cryptographic libraries, the JSSE implementation has to rely on them to handle padding errors, which generated a timing difference due to the use of exception. This example shows again a dilemma between code reuse and security: it is impossible to safely reuse standard PKCS#1 v1.5 libraries that throw exceptions. Actually, the attack keeps on resurfacing, with two recent publications exploiting Bleichenbacher oracles and targeting TLS: ROBOT (Return Of Bleichenbacher's Oracle Threats [10]), relying on new signals from vulnerable state machines, and CAT (Cache-like ATtacks [25]).

It is thus clear that PKCS#1 v1.5 is inherently flawed, and, as with the MAC-then-CBC scheme described earlier, developers will get it wrong, time and again, until this obsolete mechanism is removed from the specification. In the mean time, it is crucial to avoid reusing the same RSA key in different contexts (decryption and signature, PKCS#1 v1.5 and v2.1), since a vulnerability in one context may indirectly be used to attack the other (e.g. the DROWN attack [4]).

3.4 Lessons learned

We can expect three properties from applications involving cryptographic mechanisms: security with regards to known attacks, compatibility with the existing ecosystem, and code modularity (i.e. the ability to reuse and combine existing high-level primitives). In practice, until old versions of TLS have disappeared, it seems difficult or even impossible to have the three properties at once. A developer must pick at most two out of three:

- modularity and compatibility, which corresponds to using standard primitives without specific countermeasures, leading to attacks such as Lucky 13;
- security and compatibility, which consists in rewriting large chunks of low-level cryptographic code to add complex countermeasures. The resulting code is error-prone and hard to maintain;
- security and modularity can be obtained by using only up-to-date robust cryptographic constructions (e.g. AEAD modes), at the expense of a compatibility loss.

As history showed with Bleichenbacher attacks and CBC padding oracles, attacks only get better over time: attacks originally considered as impractical later become exploitable. As the very purpose of cryptographic protocols is security, it seems to us that the sensible approach is the third one, to only use sound algorithms and schemes to help developers do their job without having to jump through improbable hoops: a good cryptographic design should be easy to implement, in a modular and portable way, while not allowing for dangerous combinations.

Hopefully, we are now in a situation where HTTP software can rely on modern endpoints supporting at least TLS 1.2. Let us hope this situation expands to other TLS ecosystems (we can cite the use of TLS in SMTP as an area where a huge progression is still needed).

Protocol specification committees should thus listen to cryptographer's advices, and ban flawed algorithms or constructions as soon as possible. The problem with most cryptographic flaws is not *whether* they are exploitable but *when* they will be.

TLS 1.3 benefits TLS 1.3 was designed with the best intentions, and no broken or obsolete primitive has survived in the new version of the protocol.

On the symmetric front, the CBC mode and RC4 have disappeared, and only the more modern AEAD constructions have been kept (AES-GCM and Chacha20-Poly1035). Moreover, the nonce derivation is completely deterministic, which removes the possibility for error in this area. Finally, after years of using ad-hoc key derivation functions, TLS now uses HKDF, a clean and well-studied scheme proposed by Krawczyk in 2010 [16].

Regarding asymmetric primitives, RSA encryption is no longer used (which removes the possibility of RSA-EXPORT-related attacks such as FREAK) and only signed ephemeral Diffie-Hellman key exchange is possible with TLS 1.3. Moreover, the new version of the protocol uses named groups with acceptable sizes (removing other small key attacks such as LogJam [1]). Also, RSA signatures in TLS 1.3 use the Probabilistic Signature Scheme, from the most recent version of the RSA standard (PKCS#1 v2.1).

TLS 1.3 only proposes up-to-date and robust cryptographic algorithms, which should remove some worries from the developers' mind. Strictly speaking, there is still one area where legacy cryptography can be found in TLS 1.3: X.509 certificate management (ECDSA certificates are still rare, while the vast majority of RSA certificates still use the PKCS#1 v1.5 signature scheme). We can only hope that progress is made on this front, which is not directly specified by TLS.

4 The consequences of complex state machines

Since 2014, several attacks concerning flaws in TLS state machine implementations were published. Their impact can be catastrophic, either by skipping essential steps of the protocol or by exposing rarely used parts of code. Such attacks demonstrate how specification complexity can lead to security issues in implementations.

4.1 CVE-2014-0224: *EarlyCCS*

In June 2014, Kikushi showed that the OpenSSL state machine is vulnerable to a subtle attack: a man-in-the-middle between an OpenSSL client and an OpenSSL server, *both* vulnerable, could forge early `ChangeCipherSpec` messages and force the parties to use weak keys, relying only on public data [14]. The precise cinematics are described in appendix E.

The main idea behind this attack is to exploit the OpenSSL state machine that, both as a client and a server, accepts an early `ChangeCipherSpec` message, instead of discarding it and/or ending the negotiation. The real `ChangeCipherSpec`, which is still required, will be ignored in practice. At reception time, since no shared secret is defined yet, session keys are derived from a null secret and public random values. Next, the attacker has to keep both connections in a consistent state, encrypting messages with the weak keys and keeping track of record numbers to compute correct MAC values.

In the end, for the handshake to terminate successfully, the attacker has to send correct `Finished` messages to the client and to the server. Since this message must contain a hash value covering, among other things, the shared secret that was eventually agreed upon, the attacker needs both the client and the server to be vulnerable to complete the handshake.

Actually, as stated in the author's blog post, the corresponding code had already been fixed several times to handle wrongly-ordered `ChangeCipherSpec` messages: CVE-2004-0079 fixed a null-pointer assignment arising when the message was received before the ciphersuite was specified, CVE-2009-1386 fixed a similar problem in DTLS. Yet, only the direct consequence (a segmentation fault) was investigated in both cases, leaving aside the bigger picture. The genuine flaw was ignored, as well as its security consequences.

It is worth noting that `ChangeCipherSpec` is *not* a Handshake message, and as such it is *not* hashed in the transcript covered by the `Finished` message. Thus, adding or removing a `ChangeCipherSpec` cannot be detected by cryptographic means. Yet, after being removed from the standard, the `ChangeCipherSpec` were reintroduced as dummy messages in the late drafts of TLS 1.3, to accommodate so-called middleboxes. Even though these messages are not supposed to have any meaning at all, this kind of unnecessary redundancy might again lead to new issues in the years to come.

4.2 SMACK: State Machine AttaCKs

In January 2015, several vulnerabilities were published about various TLS implementations. Using FlexTLS, a flexible TLS stack, researchers tested the state machines of many different TLS stacks [6]. The results were especially worrying since they affected in practice all the known TLS stacks, to various degrees.

CVE-2014-6593: Early Finished (server impersonation). In the first described attack, the attacker answers a vulnerable client with the following messages: `ServerHello`, `Certificate` (with the identity of the server to impersonate) and `Finished`, and *skips* the rest of the negotiation (including the `ChangeCipherSpec` message. Faced with such a shortened handshake, JSSE (Java) and CyaSSL TLS implementations consider the server authenticated and start sending cleartext `ApplicationData` messages!

***Skip Verify* (client impersonation)** In the case of a mutually authenticated connection, the server requests the client to present a certificate (using a `Certificate` message) and to sign the current Handshake transcript with his private key (`CertificateVerify`). Both these messages are required to properly authenticate the client. However, several implementations accept the `Certificate` message alone, where the client announces its identity, without the corresponding proof of identity: the Mono implementation indeed considers the second message as optional, but nevertheless authenticates the client; with CyaSSL, the attacker also needs to skip the client `ChangeCipherSpec` message; finally, with OpenSSL, the flaw is more subtle, since the attack only works when the client presents a certificate containing a static Diffie-Hellman public key.

CVE-2015-0204: FREAK (Factoring RSA Export Keys) The last attack described in the article is FREAK, which got some media coverage. As for the previous attacks, FREAK relies on an active network attacker able to modify the messages on the fly.

The attack consists in forcing a client to use the RSA-EXPORT key exchange method, which was designed to comply with cryptographic restrictions. In a nutshell, with RSA-EXPORT, the server is authenticated using a strong RSA key, but the actual key exchange is done using RSA encryption with a shorter RSA key (at most 512-bit long), to respect the rules limiting the size of encryption keys.

Initially flagged as not critical for OpenSSL (which is rarely used as TLS client stack on desktop computers), FREAK was discovered in practice to affect many different TLS clients beyond OpenSSL: BoringSSL, LibreSSL, Apple SecureTransport, Microsoft SChannel, the Mono TLS stack and Oracle JSSE.

4.3 Black-box fuzzing to evaluate TLS state machines

In 2015, de Ruiter et al. described another approach to evaluate state machines in TLS stacks [26]. They use state machine learning techniques to analyze different implementations as black boxes. To this aim, they choose an alphabet of abstract TLS messages (typical Handshake messages, application data and Heartbeat messages). Thanks to a software layer translating this abstract alphabet into concrete messages (the so-called test harness), they could build the observable state automata of different implementations.

The expected automata should be a straightforward “happy flow”, showing the different steps of a successful TLS session, which should typically consist in 5 states, and one more state to handle all the error cases. This is the observed behavior for the RSA BSAFE Java library. The other studied libraries show more complex state machines. Examples of inferred automata are reproduced from the article in the appendix G.

It is worth noting that, by studying the deviations of the implementations with regards to the expected simple automata, the researchers have been able to find vulnerabilities, including the Early Finished flaw described earlier. They also uncovered another security bug in GnuTLS 3.3.8, where sending a Heartbeat message would reset the buffer containing the handshake messages; this flaw could allow an attacker to mangle a handshake between a vulnerable client and a vulnerable server.

4.4 Lessons learned

As shown with these examples, all major TLS implementations did not correctly keep track of the current *state* a session is in, since they all accepted illegal messages in at least one configuration. Ideally, the TLS state machine should be driven by its current state only, not by the incoming messages: at each step, a client or a server should exactly know which messages are valid, and every other messages should trigger an `UnexpectedMessage` fatal alert. The best way to achieve this is to write simple and crystal clear specifications in a formal language (instead of a natural one).

TLS 1.3 benefits When we study the state machines for TLS 1.3, the first remark we have to make is that they are not formally defined in the RFC. Indeed, as stated earlier, the IETF insists in letting the developers make their implementation choices, even if this leads to them making avoidable mistakes. We strongly believe that sometimes, there is a good way to implement a protocol, and more formal state machines could and should have been provided in the specification.

That being said, if we consider vanilla TLS (that is TLS without 0 RTT nor Post-Handshake Client Authentication), TLS 1.3 state machines are somewhat simpler that the previous ones. This is obviously true for the automaton handling post handshake traffic, since the only messages to handle are `NewSessionTicket` and `KeyUpdate` messages (which are very simple Handshake messages), and `ApplicationData` records.

For the first (and only) negotiation, a lot of the complexity has disappeared with the removal of several features (renegotiation, the original mechanism to resume sessions). However, during the last months of the review process, several fields and messages were brought back to the specification, to accommodate so-called middleboxes. Indeed, some network devices were shown to be intolerant to TLS 1.3, so the TLS working group proposed to make TLS 1.3 look more like TLS 1.2, by adding useless fields in the `ServerHello` message (compression methods, session identifiers) and by bringing back the cursed `ChangeCipherSpec` message. We would obviously advocate to remove this useless, unauthenticated and dangerous message, which already led to several flaws in real stacks.

Another source of concern are the 0 RTT mode, which allow for an even more efficient protocol, at the expense of weaker security properties (e.g. regarding anti-replay protection), and Post-Handshake Client Authentication, a feature allowing the server to ask for client certificate authentication after the initial handshake has been completed. Both mechanisms introduce an added complexity to the specification.

Overall, it is hard to tell what the exact track record of TLS 1.3 is with regards to specification simplicity and clarity. If we restrict the protocol to what we called vanilla TLS without `ChangeCipherSpec`, the net profit is rather clear to us. Yet, a lot of actors will be tempted to use 0 RTT or compatibility messages to accommodate middleboxes, making the profit less obvious.

5 Related work

Meyer et al. have proposed a comprehensive presentation of SSL/TLS flaws in 2013 [22], which describes many security vulnerabilities affecting TLS, not only implementation ones. At that time, the work on TLS 1.3 had not yet begun.

It is also worth mentionning the work of Bernstein et al. on developing a new cryptographic library with a safe API [5]. We indeed believe complex specifications should include implementation constraints to avoid known (and dangerous) traps.

Regarding test suites and tools, the situation has improved over the recent years, with the publication of tools such as tlsfuzzer[8] and TLS-Attacker[9].

6 Conclusion

Development, network protocols, and cryptography are complex subjects. Implementing a TLS stack combines all those, and as illustrated in the previous examples one may encounter numerous flaws with critical security consequences.

In this article, we analyzed different implementation bugs in TLS stacks that led to security flaws. We also offered ideas as to what could help the software community produce more reliable tools.

There are huge classes of security flaws that rely on recurring trivial bugs such as memory management errors or integer overflows. To overcome them, there already exists type-safe programming languages or static analysis tools to avoid introducing several kinds of bugs in the first place. The attacks have also shown that the TLS ecosystem lacked an extensive, shared set of security tests, since multiple flaws were discovered, several years apart, in independent implementations of the protocol. Finally, several vulnerabilities result from the complexity and the ambiguities of the TLS specifications.

Overall, the situation has improved with TLS 1.3. At least from the cryptographic point of view, TLS 1.3 represents significant advances, since it removes many cryptographic algorithms (such as RC4, MD5 and to a lesser extent SHA-1), modes (CBC, PKCS#1 v1.5) and parameters (arbitrary finite field group are replaced by properly sized named groups for the Diffie-Hellman key exchange). Regarding the protocol specification, the negotiation has been simplified, is more efficient, and has been proven secure... unless we consider complex features such as 0 RTT.

Acknowledgments This work was supported in part by the French ANR GASP project (ANR-19-CE39-0001).

[8] `https://github.com/tomato42/tlsfuzzer`

[9] `https://github.com/RUB-NDS/TLS-Attacker`

References

1. Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015. pp. 5–17 (Oct 2015)
2. Albrecht, M.R., Paterson, K.G.: Lucky Microseconds: A Timing Attack on Amazon's s2n Implementation of TLS. IACR Cryptology ePrint Archive (2015), http://eprint.iacr.org/2015/1129
3. AlFardan, N.J., Paterson, K.G.: Lucky thirteen: Breaking the TLS and DTLS record protocols. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA. pp. 526–540 (May 2013)
4. Aviram, N., Schinzel, S., Somorovsky, J., Heninger, N., Dankel, M., Steube, J., Valenta, L., Adrian, D., Halderman, J.A., Dukhovni, V., Käsper, E., Cohney, S., Engels, S., Paar, C., Shavitt, Y.: DROWN: Breaking TLS with SSLv2. In: 25th USENIX Security Symposium, Austin, Texas, USA (Aug 2016)
5. Bernstein, D.J., Lange, T., Schwabe, P.: The Security Impact of a New Cryptographic Library. In: Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile. pp. 159–176 (Oct 2012)
6. Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P., Zinzindohoue, J.K.: A messy state of the union: Taming the composite state machines of TLS. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA. pp. 535–552 (May 2015)
7. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.: Implementing TLS with verified cryptographic security. In: 2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA. pp. 445–459 (May 2013)
8. Bleichenbacher, D.: Chosen ciphertext attacks against protocols based on the rsa encryption standard pkcs #1. In: 18th Annual International Cryptology Conference on Advances in Cryptology, CRYPTO '98, Santa Barbara, CA, USA. pp. 1–12. Springer-Verlag (Aug 1998)
9. Bleichenbacher, D.: Rump session at CRYPTO '06: Forging some RSA signatures with pencil and paper. Transposed by Hal Finney on the IETF Web mailing list: https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html (Aug 2006)
10. Böck, H., Somorovsky, J., Young, C.: Return of bleichenbacher's oracle threat (ROBOT). In: 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 817–849 (Aug 2018)
11. Böck, H., Zauner, A., Devlin, S., Somorovsky, J., Jovanovic, P.: Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. In: 10th USENIX Workshop on Offensive Technologies, WOOT'16, Austin, USA (Aug 2016)
12. Duong, T., Rizzo, J.: Here come the XOR ninjas. Ekoparty Security Conference (Sep 2011)
13. Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The matter of heartbleed. In: Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014. pp. 475–488 (Nov 2014)
14. Kikuchi, M.: How I discovered CCS Injection Vulnerability (CVE-2014-0224). http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html (Jun 2014), http://ccsinjection.lepidum.co.jp/blog/2014-06-05/CCS-Injection-en/index.html

15. Krawczyk, H.: The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?). In: Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA. pp. 310–331 (Aug 2001)
16. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Advances in Cryptology - CRYPTO 2010, 30th Annual Cryptology Conference, Santa Barbara, CA, USA. pp. 631–648 (Aug 2010)
17. Labs, A.W.S.: s2n: an implementation of the TLS/SSL protocols. https://github.com/awslabs/s2n (2015), https://github.com/awslabs/s2n
18. Langley, A.: Lucky Thirteen attack on TLS CBC. https://www.imperialviolet.org/2013/02/04/luckythirteen.html (Feb 2013), https://www.imperialviolet.org/2013/02/04/luckythirteen.html
19. Marlinspike, M.: Internet Explorer SSL Vulnerability. http://www.thoughtcrime.org/ie-ssl-chain.txt (2002), http://www.thoughtcrime.org/ie-ssl-chain.txt
20. Marlinspike, M.: More Tricks For Defeating SSL In Practice (Jul 2009), http://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-SLIDES.pdf
21. Marlinspike, M.: BasicConstraints Back Then. http://www.thoughtcrime.org/blog/sslsniff-anniversary-edition/ (Jul 2011), http://www.thoughtcrime.org/blog/sslsniff-anniversary-edition/
22. Meyer, C., Schwenk, J.: Sok: Lessons learned from SSL/TLS attacks. In: Kim, Y., Lee, H., Perrig, A. (eds.) Information Security Applications - 14th International Workshop, WISA 2013, Jeju Island, Korea, August 19-21, 2013, Revised Selected Papers. Lecture Notes in Computer Science, vol. 8267, pp. 189–209. Springer (2013)
23. Meyer, C., Somorovsky, J., Weiss, E., Schwenk, J., Schinzel, S., Tews, E.: Revisiting SSL/TLS Implementations: New Bleichenbacher Side Channels and Attacks. In: Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA. pp. 733–748 (Aug 2014)
24. Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard) (Aug 2018). https://doi.org/10.17487/RFC8446, https://www.rfc-editor.org/rfc/rfc8446.txt
25. Ronen, E., Gillham, R., Genkin, D., Shamir, A., Wong, D., Yarom, Y.: The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In: 40th IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA (May 2019)
26. de Ruiter, J., Poll, E.: Protocol State Fuzzing of TLS Implementations. In: 24th USENIX Security Symposium, Washington, D.C., USA. pp. 193–206 (Aug 2015)

A Apple's `goto fail` vulnerable code

The following excerpt shows the vulnerable code, with the duplicated `goto` statement.

```
SSLVerifySignedServerKeyExchange( ... ) {
    OSStatus        err;

    ...

    if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
    if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
        goto fail;
        goto fail;
    if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
        goto fail;
    err = sslRawVerify(ctx, ctx->peerPubKey, dataToSign, signature);

    ...

fail:
    SSLFreeBuffer(&signedHashes);
    SSLFreeBuffer(&hashCtx);
    return err;
}
```

B GnuTLS' `goto fail` vulnerable code

The following listing is an extract from the vulnerable function in GnuTLS which returns `-1` in case `_gnutls_x509_get_signed_data` fails while parsing the data. It is interesting to notice that this contradicts to the comments on top of the function.

```
/*
* Returns only 0 or 1. If 1 it means that the certificate
* was successfully verified. [...]
*/
static int _gnutls_verify_certificate2( ... ) {

    ...

    result = _gnutls_x509_get_signed_data( ... );
    if (result < 0) {
        gnutls_assert();
        goto cleanup;
    }

    ...

cleanup:
    if (result >= 0 && func)
        func(cert, issuer, NULL, out);
    _gnutls_free_datum(&cert_signed_data);
    _gnutls_free_datum(&cert_signature);

    return result;
}
```

And here is typical site call of the vulnerable function, where only the `ret == 0` condition (corresponding to an invalid signature) would lead to reject the certificate, letting negative results be interpreted as good certificates.

```
    ret = _gnutls_verify_certificate2( ... );
    if (ret == 0) {
        /* if the last certificate in the certificate list is
         * invalid, then the certificate is not trusted.  */
        gnutls_assert();
        status |= output;
        status |= GNUTLS_CERT_INVALID;
        return status;
    }
```

C Details on CVE-2014-1568: NSS/CyaSSL/PolarSSL Signature Forgery

Within TLS, RSA signatures use the PKCS#1 v1.5 standard (it is also the standard commonly used in X.509 certificates, hence the ability to exploit the vulnerability anywhere in the chain). To sign a message m with a private key (N, d) using the hash function H, PKCS#1 v1.5 requires the following steps:

- Hash: compute $h = H(m)$;
- Format: prepare an ASN.1 DER block encoding a sequence containing the identifier of the used hash function and the hash value h. We denote by d this block, also called `DigestInfo`;
- Pad: with n the length of the RSA modulus N, create an n-byte long octet string starting with `00 01 ff ... ff 00`, followed by d, where the number of `ff` bytes is adjusted accordingly. Let x be the integer represented by this value (see figure 1);
- Sign: the final result is $x^d[N]$.

In 2006, Bleichenbacher proposed an attack to exploit broken RSA implementations that did not check the absence of data beyond the `DigestInfo` block [9]. Figure 2 presents an *incorrect* message that could be accepted by such implementations. In the case of a small public exponent (such as 3), it is easy to forge a signature for such a message. Assuming an attacker wishes to forge a signature for a given `DigestInfo` block, she can prepare a message m similar to the one presented in the figure, with a small value for p[10] and the garbage part filled with zeroes. Let s be the smallest integral value greater than the *real* cubic root of the big integer represented by m. As soon as n is big enough, s^3 will only differ from m in the garbage part of the message, leaving the `DigestInfo` part intact. With such a fuzzy implementation, it is thus possible to forge an arbitrary signature for an RSA key using 3 as its public exponent.

The vulnerability exposed in September 2014 is subtler, since it relies on a non-canonical DER representation. For example, instead of representing the SHA-1 hash length by a simple byte (`0x14`), the attacker uses an alternative representation, which should be forbidden in DER, `XX 00 .. 00 14`, with `XX` equal to `0x80` + l, the number of bytes used to encode the length (here, the number of null bytes plus one).

Even if this lack of precision allows the attacker to mount an attack, it is still hard to exploit in practice. There was actually another flaw in the DER parsing code: when reading a very long length field (as the one described above), an integer overflow allowed the attacker to use arbitrary values for all but the four bytes, which would lead

[10] As a matter of fact, even if p is specified to be at least 8, some implementations allow the padding to be empty ($p = 0$), which fives the attacker more wiggle room.

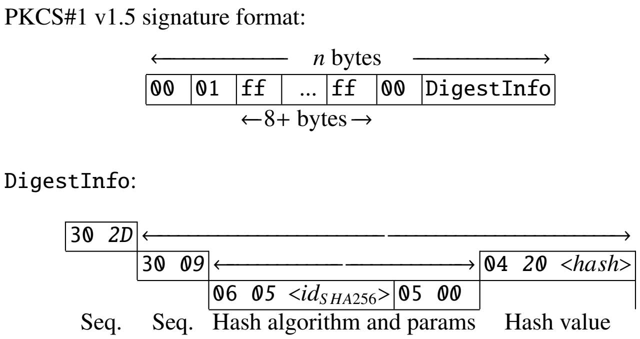

Fig. 1. PKCS#1 v1.5 signature format. *Seq.* are ASN.1 sequences, length are represented in italic, and <hash> is the hash value of the message to sign (here using SHA-256).

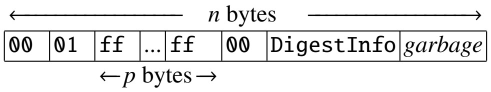

Fig. 2. Example of a malformed PKCS#1 v1.5 message. With a liberal parser, messages of this form will be accepted, which leads to a Universal signature forgery attack.

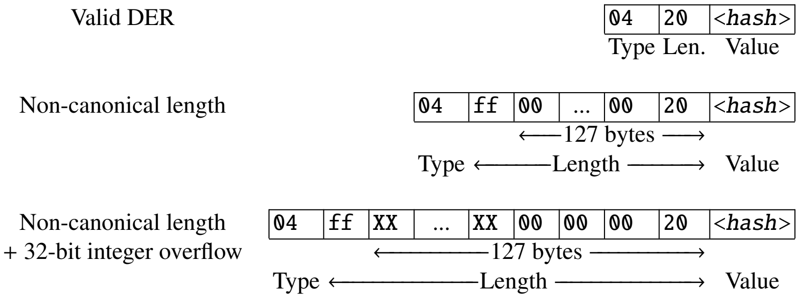

Fig. 3. Example of a subtler malformed PKCS#1 v1.5 message, which was accepted by many implementations in 2014. In the ASN.1 DER part of the message describing the hash value, the length field can be mangled and still be accepted. In the third case, because of the integer overflow, XX can be chosen arbitrarily by the attacker. Of course, the rest of the DigestInfo must be built accordingly.

to a length field of the form XX YY .. YY 00 00 00 14, where YY is a sequence of $l - 4$ bytes controlled by the attacker.

Figure 3 shows an example of message targeted by this attack with a 1024-bit modulus (since the XX only allows for a 7 bit-long length field, the attack is more complex and requires splitting the target in more pieces with 2048-bit moduli). As in the original Bleichenbacher attack against PKCS#1 v1.5 signature forgery attack, the public exponent must be equal to 3 for the attack to succeed in practice.

D Details on Bleichenbacher's attack against PKCS#1 v1.5 encryption

Figure 4 describes the expected format for the structure to be encrypted with PKCS#1 v1.5. A correctly padded plaintext thus starts with 00 02, which corresponds to a big integer between $2 \times 2^{n-16}$ and $3 \times 2^{n-16}$ (with an n-bit modulus).

PKCS#1 v1.5 encryption format:

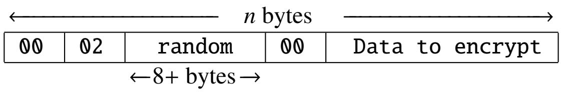

Fig. 4. A valid PKCS#1 v1.5 encrypted message: the padding must contain at least 8 non-null random bytes.

E EarlyCCS attack cinematics

Figure 5 describes how the EarlyCCS attack can be mounted between a client and a server that are both vulnerable.

Indeed, even if the secret used to protect records after the ChangeCipherSpec messages is known to the attacker, the legitimate parties still properly derive the contents of the Finished messages, which is still secure and out of reach of the attacker.

F FREAK attack cinematics

The FREAK scenario, as presented in Figure 6, is the following:

- First the attacker A finds a server S accepting RSA-EXPORT ciphersuites, and using the same short-term RSA-512 key across sessions. Many TLS servers still accept to negotiate RSA-EXPORT ciphersuites, and even reuse the same short-term RSA-EXPORT key until they reboot[11].

[11] Even if the short-term RSA should ideally be ephemeral, generating even a small RSA key on the fly is still considered a costly operation. RSA-EXPORT implementations thus usually cache the short-term RSA key for a certain amount of time, typically between one hour and forever.

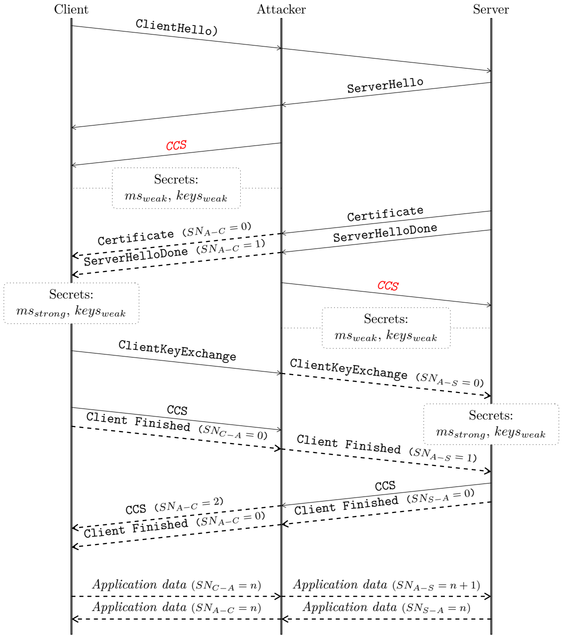
Client
Attacker
Server
ClientHello)
ServerHello
CCS
Secrets:
ms_{weak}, $keys_{weak}$
Certificate
ServerHelloDone
Certificate ($SN_{A-C} = 0$)
ServerHelloDone ($SN_{A-C} = 1$)
CCS
Secrets:
ms_{strong}, $keys_{weak}$
Secrets:
ms_{weak}, $keys_{weak}$
ClientKeyExchange
ClientKeyExchange ($SN_{A-S} = 0$)
Secrets:
ms_{strong}, $keys_{weak}$
CCS
Client Finished ($SN_{C-A} = 0$)
Client Finished ($SN_{A-S} = 1$)
CCS
Client Finished ($SN_{S-A} = 0$)
CCS ($SN_{A-C} = 2$)
Client Finished ($SN_{A-C} = 0$)
Application data ($SN_{C-A} = n$)
Application data ($SN_{A-S} = n + 1$)
Application data ($SN_{A-C} = n$)
Application data ($SN_{S-A} = n$)

Fig. 5. EarlyCCS attack cinematics. *ms* stands for *master secret* and SN_{XX} corresponds to the number of sent *record*. The attacker needs to keep track of four such numbers: between the client and the attacker and between the attacker and the server (with a counter for each direction).

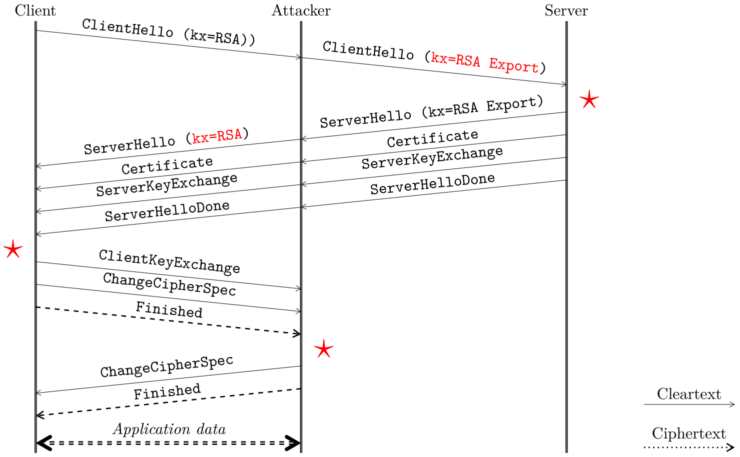

Fig. 6. FREAK attack. Key steps are in red: Hello message mangling, server support for EXPORT ciphersuite, client tolerance to an unsolicited EXPORT-specific message, and the ability of the attacker to decrypt the `ClientKeyExchange` (which depends on a reused weak RSA key).

- *A* factors the RSA-512 modulus (it takes several hours with a reasonable budget) to get the private key;
- *A* leads a man-in-the-middle attack between a vulnerable client *C* and *S*, forces the client to negotiate an RSA key exchange while fooling the server into using an RSA-EXPORT ciphersuite. Several TLS clients indeed accept a `ServerKeyExchange` message containing a short-term weak RSA key, which should only appear with an RSA-EXPORT ciphersuite, whereas a standard RSA encryption key exchange had been negotiated
- *A* then hands over the following messages until the `ServerHelloDone`;
- *A* receives the `ClientKeyExchange` she is able to decrypt with the weak private key;
- finally, *A* answers directly to *C* on behalf of *S* for the rest of the session.

G Examples of automata inferred from TLS implementations

Figure 7 describes the automaton for the BSAFE implementation and Figure 8 the one for GnuTLS.

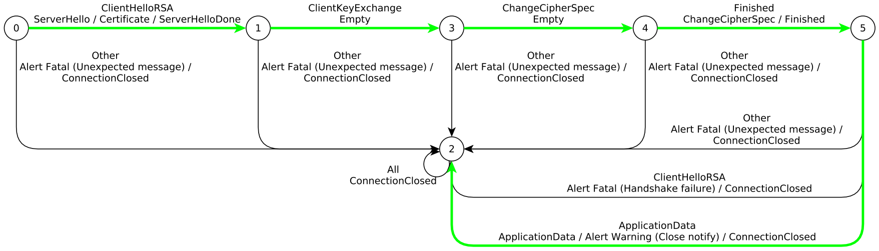
0
1
2
3
4
5
ClientHelloRSA
ServerHello / Certificate / ServerHelloDone
ClientKeyExchange
Empty
ChangeCipherSpec
Empty
Finished
ChangeCipherSpec / Finished
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
All
ConnectionClosed
ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed
ApplicationData
ApplicationData / Alert Warning (Close notify) / ConnectionClosed

Fig. 7. Observable state automata of the RSA BSAFE JAVA stack (version 6.1.1). 5 states clearly form the expected “happy flow”, while the `2` state is the error state, where all invalid sessions eventually end. Source: [26].

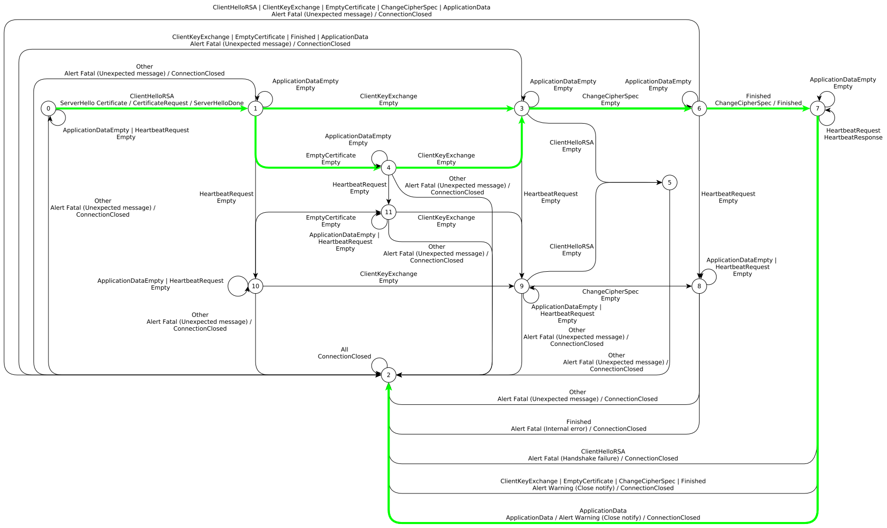
ClientHelloRSA | ClientKeyExchange | EmptyCertificate | ChangeCipherSpec | ApplicationData
Alert Fatal (Unexpected message) / ConnectionClosed
ClientKeyExchange | EmptyCertificate | Finished | ApplicationData
Alert Fatal (Unexpected message) / ConnectionClosed
Other
Alert Fatal (Unexpected message) / ConnectionClosed
ApplicationDataEmpty
Empty
ClientHelloRSA
ServerHello Certificate / CertificateRequest / ServerHelloDone
0
1
ClientKeyExchange
Empty
3
ApplicationDataEmpty
Empty
ChangeCipherSpec
Empty
ApplicationDataEmpty
Empty
6
Finished
ChangeCipherSpec / Finished
7
ApplicationDataEmpty
Empty
HeartbeatRequest
HeartbeatResponse
ApplicationDataEmpty | HeartbeatRequest
Empty
ApplicationDataEmpty
Empty
EmptyCertificate
Empty
4
ClientKeyExchange
Empty
ClientHelloRSA
Empty
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
5
HeartbeatRequest
Empty
HeartbeatRequest
Empty
HeartbeatRequest
Empty
HeartbeatRequest
Empty
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
11
EmptyCertificate
Empty
ClientKeyExchange
Empty
ApplicationDataEmpty |
HeartbeatRequest
Empty
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
ClientHelloRSA
Empty
ApplicationDataEmpty |
HeartbeatRequest
Empty
ApplicationDataEmpty | HeartbeatRequest
Empty
10
ClientKeyExchange
Empty
9
ChangeCipherSpec
Empty
8
ApplicationDataEmpty |
HeartbeatRequest
Empty
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
All
ConnectionClosed
Other
Alert Fatal (Unexpected message) /
ConnectionClosed
2
Other
Alert Fatal (Unexpected message) / ConnectionClosed
Finished
Alert Fatal (Internal error) / ConnectionClosed
ClientHelloRSA
Alert Fatal (Handshake failure) / ConnectionClosed
ClientKeyExchange | EmptyCertificate | ChangeCipherSpec | Finished
Alert Warning (Close notify) / ConnectionClosed
ApplicationData
ApplicationData / Alert Warning (Close notify) / ConnectionClosed

Fig. 8. Observable state automata of GNU TLS 3.3.8. This time, the automata contains 12 states. In particular, states 8 to 10 form a shadow flow, where a Heartbeat message has led to a buffer reset. Source: [26].