

Improving the security and usability of TLS in MariaDB



Daniel Lenski

Amazon RDS for MySQL/MariaDB core engine team

Wednesday, 4 October 2023

About me

- In the Amazon **RDS** open-source core team at Amazon Web Services for 2 years.
- Lots of contributions to free/open-source software over 20+ years (see dlenksi.github.io)
- Core developer of the **OpenConnect** VPN client



Monty visited us in May 2022 :-)



Outline

- Background:
 - What is TLS?
 - Why do we need TLS?
- A critical look at TLS in MariaDB
 - Protocol and code problems
 - Resulting vulnerabilities
 - User experience problems
- Proposed solutions
- Questions?

Background

What is TLS?

Why do we need it?

How have these needs changed over time?

Introduction to TLS

- Transport Layer Security is an Internet standard protocol
- Designed to provide privacy and security to network protocols
- Known as TLS since 1999
- Earlier iterations were called SSL (Secure Sockets Layer)
- The name SSL is still widely used to refer to TLS, including in MySQL and MariaDB



Protocol ↕	Published ↕	Status ↕
SSL 1.0	Unpublished	Unpublished
SSL 2.0	1995	Deprecated in 2011 (RFC 6176)
SSL 3.0	1996	Deprecated in 2015 (RFC 7568)
TLS 1.0	1999	Deprecated in 2021 (RFC 8996) ^{[20][21][22]}
TLS 1.1	2006	Deprecated in 2021 (RFC 8996) ^{[20][21][22]}
TLS 1.2	2008	In use since 2008 ^{[23][24]}
TLS 1.3	2018	In use since 2018 ^{[24][25]}

Improving security with each revision of the standard!

- Not *just* fixing known vulnerabilities
- Thinking about newer and more sophisticated **threat models**

What does TLS do?

- TLS promises that, **if it is used correctly**, it will provide:

- **End-to-end encryption and authentication**


Applications can create a confidential channel over an untrusted network, which *only* the endpoints can read or write to.

- **Peer authentication**

Client can cryptographically verify that they've established a channel *with the intended server*.

- For application developers: TLS aims to be a drop-in replacement for unencrypted network connections (like TCP)

What *were* our privacy and security concerns?

- Many applications started adding support for TLS around 2000-2005 (including [MySQL](#))
- What was the *threat model* that users were worried about in **2005**?
-  Everything was plaintext. If you could see our packets, you could read them.

Opportunistic Eavesdropper

She's running [Firesheep](#) to steal Facebook and webmail logins.



Simple Automated Censor

The school's routers look for:

- BitTorrent packets
- HTTP requests that appear to download .EXE files

... and inject [TCP resets](#) to kill them.

Changing threats and requirements over time

- Since then...
- Ever-increasing computing power...
- More and more of *everything* is online...
- More study of Internet protocols...
- Better-organized and better-funded attackers...
- More sophisticated threats to privacy and security on the Internet.
- Approaches that were “good enough” back in 2005 no longer are.
- ***Applications and protocols need to keep up.***

How have our our privacy and security concerns *changed*?

- Consolidation of network control
- Major [revelations about Internet surveillance](#) by governments (2013)
- We should be thinking about ***pervasive attackers***, including:
 - Intelligence agencies
 - Censorship agencies
 - Internet service providers
 - Datacenters
 - Collaborations among the above

What can pervasive attackers do?

- Track *any* network connection statefully
- Inspect, log, inject **at every layer below TLS**
- Fingerprint for vulnerable software versions
- Research and discover vulnerabilities and exploit them without public disclosure
- Inject **attacks targeted against individuals or groups**

Defending against modern threat model

- *Lots* to worry about, but no need to despair.
 - Consistent expert consensus: TLS, implemented and used correctly, is a very strong defense.
- Pervasive attackers exploit vulnerabilities
 - Use well-funded, well-tested, standards-compliant, up-to-date TLS libraries.
- Pervasive attackers can do machine-in-the-middle attacks (MITM)
 - Clients must verify servers' identities

<http://www.wired.com/2013/09/black-budget-what-exactly-are-the-nsas-cryptanalytic-capabilities>



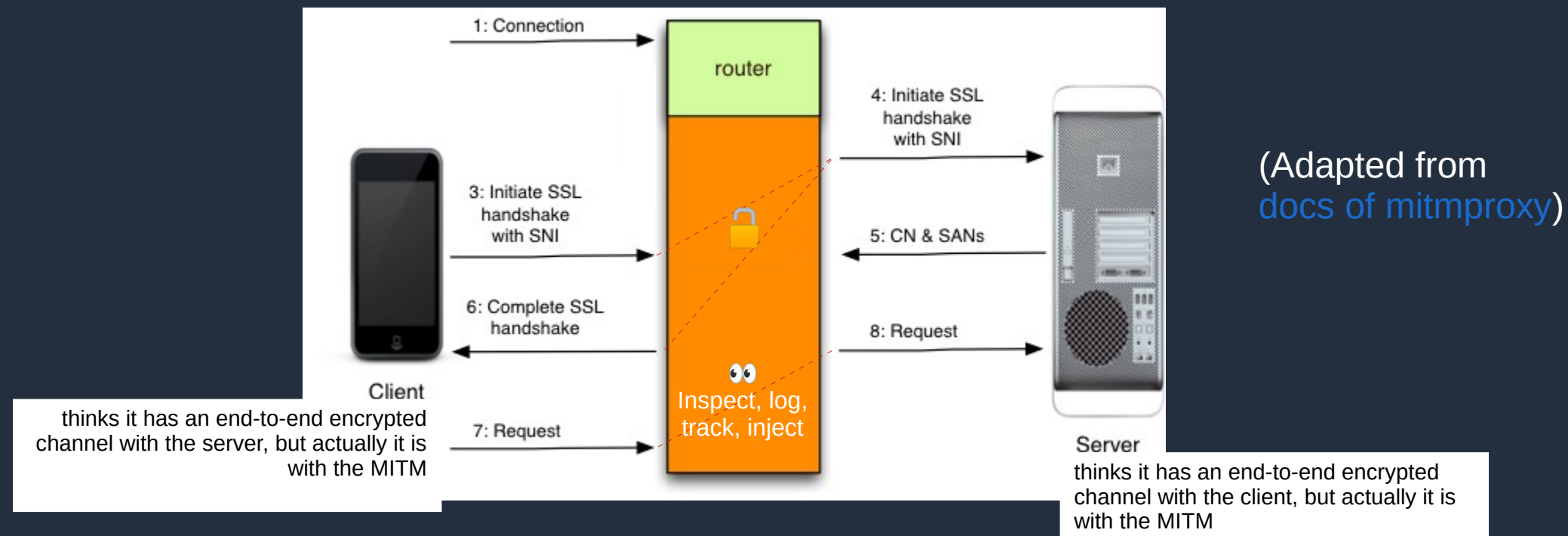
BRUCE SCHNEIER OPINION SEP 4, 2013 9:29 AM

What Exactly Are the NSA's 'Groundbreaking Cryptanalytic Capabilities'?

Whatever the NSA has up its top-secret sleeves, the mathematics of cryptography will still be the most secure part of any encryption system.

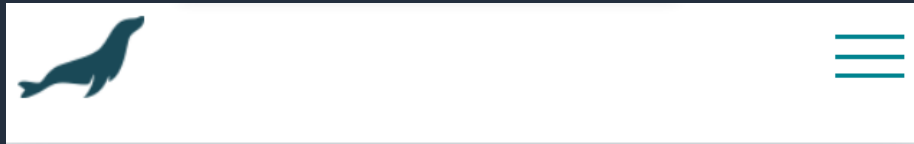
What is a MITM attack on TLS?

- Machine in the middle sees a client connecting to a server with TLS
- Attacker completes 2 TLS handshakes: 1 with client, 1 with server
- Attacker can read, relay, and modify traffic as if unencrypted
- Client must verify server's identity; TLS gives tools for this.



What do users expect from TLS?

TLS/SSL is a *brand*



| → |
| → |

End-to-end encryption

The connections between clients, proxies and databases can be encrypted with Transport Layer Security (**TLS**) to protect data in motion while tables and binary logs can be encrypted with Advanced Encryption Standard (AES) algorithms to protect data at rest – and there are plugins for the AWS Key Management Service (KMS) and the *eperi* Gateway.

Does Skype use encryption?

All Skype-to-Skype voice, video, file transfers and instant messages are encrypted. This protects you from potential eavesdropping by malicious users.

If you make a call from Skype to mobile and landline phones, the part of your call that takes place over the PSTN (the ordinary phone network) is not encrypted.

For example, in the case of group calls involving two users on Skype-to-Skype and one user on PSTN, then the PSTN part is not encrypted, but the Skype-to-Skype portion is.

For instant messages, we use **TLS** (transport-level security) to encrypt your messages between your Skype client and the chat service in our cloud, or AES

Pages from MariaDB Corp, Skype, Amazon shopping, F5, Canadian credit card consortium... advertising their products' security.

How Secure Is Information About Me?

We design our systems with your security and privacy in mind.

- We work to protect the security of your personal information during transmission by using encryption protocols and software.
- We follow the Payment Card Industry Data Security Standard (PCI DSS) when handling credit card data.

Surf on **Secure** Websites

Be extra careful when entering your credit card online. Only enter your credit card information on **secure** websites. What's a **secure** website? The website address should begin with HTTPS, rather than HTTP. HTTPS provides you with an added level of **security**, by authenticating and encrypting sensitive data. Internet **security** is so important that Google is even giving [websites with HTTPS a higher search engine ranking](#).

Why Are VPNs Important?

VPNs have become a crucial part of many organizations' security strategies regardless of business size, industry vertical, or geographic location. VPNs provide a way for authorized remote users to gain access to files, databases, and other network applications in a secure manner.

Due to the inherent security risks of the Internet, companies providing remote access and telecommuting choices must protect their private data through a VPN.

An even more secure version of the VPN is the [Secure Sockets Layer Virtual Private Network \(SSL VPN\)](#). An **SSL** VPN uses the Secure Sockets Layer (**SSL**) protocol to create a secure and encrypted connection over the Internet. The **SSL** VPN was created to ensure enhanced security and privacy.

What do users expect from this brand?

- If an application says that TLS/SSL is in use...
- Users **expect** data to be transmitted without possibility of eavesdropping, forgery, or interference.
- **If these expectations are violated, then**
 - **“TLS is supported and enabled” is**
 - **Incomplete at best**
 - **Misleading and dangerous at worst.**
- **Users expect to be protected against modern threat models**

TLS needs to be easy to use and hard to misuse

- Many users know they want TLS, but not how to use it well
- Make it as easy as possible **to use TLS securely by default** with:
 - Good design.
 - *Secure-by-default* configuration, *fail-secure* behavior.
 - Good documentation
 - Including communication about *changes that affect security*.
- *Remove options or features* that make it too easy to use software *insecurely*.

Critical look at TLS in MariaDB

Overarching technical problem
Specific vulnerabilities

User experience problems
Specific examples

Switching to TLS securely needs careful design

- MariaDB starts with a plain TCP socket and then *switches to using TLS* on the same TCP socket.

Opportunistic TLS (Transport Layer Security) refers to extensions in plain text communication protocols, which offer a way to upgrade a plain text connection to an encrypted (TLS or SSL) connection instead of using a separate port for encrypted communication. Several protocols use a command named "**STARTTLS**" for this purpose. It is a form of [opportunistic encryption](#) and is primarily intended as a countermeasure to [passive monitoring](#).

- This needs *careful* design:
 - Backwards-compatibility with peers who can't do TLS (if necessary)
 - TLS-capable peers are protected against downgrade attacks
 - Leak as little information as possible in pre-TLS exchanges

Pre- and post-TLS application state

- *Any* data or configuration exchanged before switching to TLS should be **thrown out** after the handshake.
- If the state of the application *after* the TLS handshake is influenced by the state *before* the handshake...
 - Information leakage
 - Downgrading or forgery

How to switch to TLS correctly

- Design the protocol *before* writing code, not the other way around. Will it meet the requirements?
- Use **language features** and good **program design** to
 - Ensure pre-TLS and post-TLS state are isolated
 - No global vars, no preexisting “god object”
 - Make code as compact and self-contained as possible

Server code:

```
static CLIENT_CONN *greet_client_and_setup_TLS(
    int tcp_socket, const SERVER_TLS_CONTEXT *t, bool allow_insecure)
{
    send_server_greeting_packet(tcp_socket);
    CLIENT_GREETING *g = recv_pkt(tcp_socket);

    if (g->wants_TLS) {
        int tls_sess = TLSLib_do_handshake_as_server(tcp_socket, t);

        /* Don't trust the plaintext/pre-TLS greeting packets: redo! */
        free(g);
        send_server_greeting_packet(tls_session);
        g = recv_pkt(tls_session);

        /* Create our CLIENT_CONN (app state object) from scratch */
        CLIENT_CONN *c = malloc(sizeof(*c));
        *c = (CLIENT_CONN){.greeting=g, .fd=tls_session, .transport="TLS"}; // C99
        return c;
    }

    if (allow_insecure) {
        CLIENT_CONN *c = malloc(sizeof(*c));
        *c = (CLIENT_CONN){.greeting=g, .fd=tcp_socket, .transport="TCP"}; // C99
        return c;
    }

    error("Refusing request for insecure connection!");
    return NULL;
}

int accept_client_connection(
    int tcp_socket, SERVER_TLS_CONTEXT *t, bool allow_insecure)
{
    /* Greet client and secure the transport layer */
    CLIENT_CONN *cc = greet_client_and_setup_TLS(
        tcp_socket, t, allow_insecure);

    /* Do the application-layer authentication */
    application_authentication(cc);

    /* ... */
}
```

This should **default to False** in *both* client and server implementations.

Problems with how MariaDB switches to TLS

0) Flawed design for switching to TLS.

1) Global state and spaghetti code. Hard to read, test, or simplify.

2) No separation of concerns between code for TLS setup and application setup.

3) No separation of pre-TLS and post-TLS state.

4) Large amount of code. `sql_acl.cc` is *15,000 lines long*, and includes code for two different authentication methods in addition to TLS setup and general application setup.

Vulnerabilities

- Pervasive attackers can watch for MariaDB connections and...
 - 1) Undetectably downgrade to plaintext ([MDEV-28634](#) and [CONC-656](#))
 - 2) Or undetectably MITM the TLS ([CONC-656](#))
 - 3) Mislead and DOS clients by sending forged server errors, with no TLS awareness ([CONC-648](#))
 - 4) Fingerprint clients for location-specific character sets ([CONC-654](#))
 - 5) Fingerprint clients for specific software versions ([CONC-654](#)) and launch other as-yet-unknown attacks
- First two can be defended by using `--ssl-verify-server-cert`
 - ... but non-default and hard to configure, *so a lot of users don't*

A resulting vulnerability

- **CONC-648**: Client improperly trusts errors sent before TLS handshake (reported 6 June 2023)
- Clients using TLS *should not* trust messages allegedly sent by the server before the TLS handshake.

```
1556 MYSQL *mthd_my_real_connect(MYSQL *mysql, const char *host, const char *user,
1557                             const char *passwd, const char *db,
1558                             uint port, const char *unix_socket, unsigned long client_flag)
1559 {
1560     char      buff[NAME_LEN+USERNAME_LENGTH+100];
1561     char      *end, *end_pkt, *host_info;
1562     MA_PVIO_CINFO cinfo= {NULL, NULL, 0, -1, NULL};
1563     MARIADB_PVIO *pvio= NULL;
1564     char      *scramble_data;
1565     const char *scramble_plugin;
1566     uint pkt_length, scramble_len, pkt_scramble_len= 0;
1567     NET *net= &mysql->net;
1568     my_bool is_multi= 0;
1569     char *host_copy= NULL;
1570     struct st_host *host_list= NULL;
1571     int connect_attempts= 0;
1572
1573     if (!mysql->methods)
1574         mysql->methods= &MARIADB_DEFAULT_METHODS;
1575
1576     if (net->pvio) /* check if we are already connected */
1577     {
1578         SET_CLIENT_ERROR(mysql, CR_ALREADY_CONNECTED, SQLSTATE_UNKNOWN, 0);
1579         return(NULL);
1580     }
```

200 lines later... have we switched to TLS yet?

```
1788     if ((pkt_length=ma_net_safe_read(mysql)) == packet_error)
1789     {
1790         if (mysql->net.last_errno == CR_SERVER_LOST)
1791             my_set_error(mysql, CR_SERVER_LOST, SQLSTATE_UNKNOWN,
1792                         ER(CR_SERVER_LOST_EXTENDED),
1793                         "handshake: reading initial communication packet",
1794                         errno);
1795
1796         goto error;
1797     }
```

CONC-648: Client improperly trusts errors sent before TLS handshake

- A client connects, with TLS:
 - `mariadb --ssl-verify-server-cert mariadb.server.com`
- MITM injects fake error packet:

```
15:24:46.185305 IP 127.0.0.1.3306 > 127.0.0.1.40234: Flags [P.], seq 1:189,
0x0000: 4508 00f0 622c 4000 8006 99d1 7f00 0001 E...b,@.....
0x0010: 7f00 0001 0cea 9d2a c10b a180 5c31 bae4 .....*\1..
0x0020: 8018 0200 fee4 0000 0101 080a 06c3 d23b .....;
0x0030: 06c3 d238 b800 0000 ff17 0749 6e74 6572 ...8.....Inter
0x0040: 6e61 6c20 6572 726f 723a 2043 6c69 656e nal.error:.Clie
0x0050: 7420 7769 6c6c 2061 6363 6570 7420 7468 t.will.accept.th
0x0060: 6973 2065 7272 6f72 2061 7320 6765 6e75 is.error.as.genu
0x0070: 696e 6520 6576 656e 2069 6620 7275 6e6e ine.even.if.runn
0x0080: 696e 6720 7769 7468 202d 2d73 736c 202d ing.with.--ssl.-
0x0090: 2d73 736c 2d76 6572 6966 792d 7365 7276 -ssl-verify-serv
0x00a0: 6572 2d63 6572 742c 2061 6e64 2065 7665 er-cert,.and.eve
0x00b0: 6e20 7468 6f75 6768 2074 6869 7320 6572 n.though.this.er
0x00c0: 726f 7220 6973 2073 656e 7420 696e 2070 ror.is.sent.in.p
0x00d0: 6c61 696e 7465 7874 2050 5249 4f52 2054 laintext.PRIOR.T
0x00e0: 4f20 544c 5320 4841 4e44 5348 414b 452e O.TLS.HANDSHAKE.
```

- Connector/C library reports this as a real error from the real server, with no indication that it was sent pre-TLS:

```
ERROR 1815 (HY000): Internal error: Client will accept this error as genuine even if running with --ssl --ssl-verify-server-cert, and even though this error is sent in plaintext PRIOR TO TLS HANDSHAKE.
```

- Clients do need to report certain errors before TLS handshake is complete:
 - *Those* error conditions are determined by the *client*
 - They should not involve trusting information sent by the server

CONC-648: Client improperly trusts errors sent before TLS handshake

- *Immediate* risks?
- 😊 Can't be directly used to extract application-level data
- 😬 Trivial to use for DOS attacks...
 - Inject ER_ACCESS_DENIED_ERROR (“wrong password”) to convince clients to **stop** *retrying*.
 - Inject ER_GET_TEMPORARY_ERRMSG (“temporary failure”) errors to convince clients to **keep** *retrying*
 - Inject ER_CON_COUNT_ERROR / ER_OUT_OF_RESOURCES to get clients to connect to another server.

Daniel Lenski added a comment - 2023-06-06 18:29

Do you think it's a problem?

Due to this vulnerability in MariaDB, an on-path attacker can signal an arbitrary error to a client, which might (for example) cause the client to alter its retry behavior or to misdiagnose a connection problem.

CONC-648: Client improperly trusts errors sent before TLS handshake

- *Future risks?*
- As long as this bug exists...
- The MariaDB protocol **cannot evolve** in a way where clients would automatically take consequential actions based on error messages sent by the server.

Daniel Lenski added a comment - 2023-06-06 18:29

Do you think it's a problem?

I discovered this vulnerability in MariaDB because I'm working on an approach to MDEV-15935 ("server redirection mechanism") in which the server tells the client to switch to another server using an error packet. If error packets are indeed chosen as the mechanism for server redirection, this means that **attackers would be able to redirect clients to arbitrary attacker-controlled servers even when the clients think they are using TLS.**

CONC-648: Client improperly trusts errors sent before TLS handshake

- I created a tiny fix for this issue, [Connector/C PR#223](#)
- It has been up since 12 June
- Revised based on feedback from Sergei Golubchik and Andrew Hutchings.

```
libmariadb/mariadb_lib.c
@@ -1787,7 +1787,12 @@ MYSQL *mthd_my_real_connect(MYSQL *mysql, const char
1787 1787  */
1788 1788  if ((pkt_length=ma_net_safe_read(mysql)) == packet_error)
1789 1789  {
1790 -   if (mysql->net.last_errno == CR_SERVER_LOST)
1790 +   if (mysql->options.use_ssl)
1791 +       my_set_error(mysql, CR_CONNECTION_ERROR, SQLSTATE_UNKNOWN,
1792 +           "Received error packet before completion of TLS handshake
1793 +           "The authenticity of the following error cannot be verifi
1794 +           mysql->net.last_errno, mysql->net.last_error);
1795 +   else if (mysql->net.last_errno == CR_SERVER_LOST)
1791 1796       my_set_error(mysql, CR_SERVER_LOST, SQLSTATE_UNKNOWN,
1792 1797           ER(CR_SERVER_LOST_EXTENDED),
1793 1798           "handshake: reading initial communication packet",
.....
```

dlenski marked this conversation as resolved.

Vulnerabilities due to badly-designed redundant switch to TLS

- Reported 30 June, a protocol-level problem:
 - [CONC-654](#): Clients send too much info before TLS handshake
 - [MDEV-31585](#): ... and the servers *requires* it to be sent
- Results from poor design in the protocol *for switching to* TLS:
 - Servers expect clients to send a near-identical greeting packet once in plaintext (before TLS handshake), and once over TLS

CONC-654 & MDEV-31585: too much required info in pre-TLS exchanges

- Clients reveal their “capabilities”
- Clients reveal their preferred character set in plaintext

```
29 193.493655208 127.0.0.1 127.0.0.1 MySQL 158 Server Greeting proto=10 version=11.2.0-MariaDB
30 193.493695724 127.0.0.1 127.0.0.1 TCP 66 47472 → 3307 [ACK] Seq=1 Ack=93 Win=65536 Len=0
31 193.493803266 127.0.0.1 127.0.0.1 MySQL 102 Login Request user=
32 193.493825208 127.0.0.1 127.0.0.1 TCP 66 3307 → 47472 [ACK] Seq=93 Ack=27 Win=65536 Len=0
```

▼ MySQL Protocol
Packet Length: 32
Packet Number: 1
▼ Login Request
▼ Client Capabilities: 0xaa84
.....0 = Long Password: Not set
.....0 = Found Rows: Not set
.....1 = Long Column Flags: Set
.....0 = Connect With Database: Not set
.....0 = Don't Allow database.table.column: Not set
.....0 = Can use compression protocol: Not set
.....0 = ODBC Client: Not set
.....1 = Can Use LOAD DATA LOCAL: Set
.....0 = Ignore Spaces before '(': Not set
.....1 = Speaks 4.1 protocol (new flag): Set
.....0 = Interactive Client: Not set
.....1 = Switch to SSL after handshake: Set
.....0 = Ignore sigpipes: Not set
.....1 = Knows about transactions: Set
.....0 = Speaks 4.1 protocol (old flag): Not set
.....1 = Can do 4.1 authentication: Set
▼ Extended Client Capabilities: 0x00bf
.....1 = Multiple statements: Set
.....1 = Multiple results: Set
.....1 = PS Multiple results: Set
.....1 = Plugin Auth: Set
.....1 = Connect attrs: Set
.....1 = Plugin Auth LENENC Client Data: Set
.....0 = Client can handle expired passwords: Not set
.....1 = Session variable tracking: Set
.....0 = Deprecate EOF: Not set
0000 000 = Unused: 0x00
MAX Packet: 16777216
Charset: utf8 COLLATE utf8_general_ci (33)
Unused: 00
▼ MariaDB Extended Client Capabilities: 0x0000001d
.....1 = Progress indication: Set
.....0 = Multi commands: Not set
.....1 = Bulk Operations: Set
.....1 = Extended metadata: Set
Username:

CONC-654 & MDEV-31585: too much required info in pre-TLS exchanges

- Immediate risks?
- **Many opportunities** for fingerprinting specific client versions
 - Easy to iterate through every release, or even every commit, of Connector/C library.
 - Build it.
 - See how its default capability bits change.
 - Pervasive attackers may know of undisclosed vulnerabilities in specific client versions, and target them based on this fingerprint.
- Geographic fingerprinting based on character sets.

CONC-654 & MDEV-31585: too much required info in pre-TLS exchanges

- This is a *protocol* flaw involving redundant exchanges, and different interpretations of them.
- It **cannot** be fixed in a fully-backwards-compatible way in either the client or the server alone.
- [Connector/C PR#227](#) and [server PR#2684](#) (submitted 3 July, revised based on feedback)
 - Client and server can negotiate a new “v2 TLS handshake”
 - Other compatible client libraries are interested in supporting this as well (see [mysql.net #1342](#) from 9 July)

v2 handshake

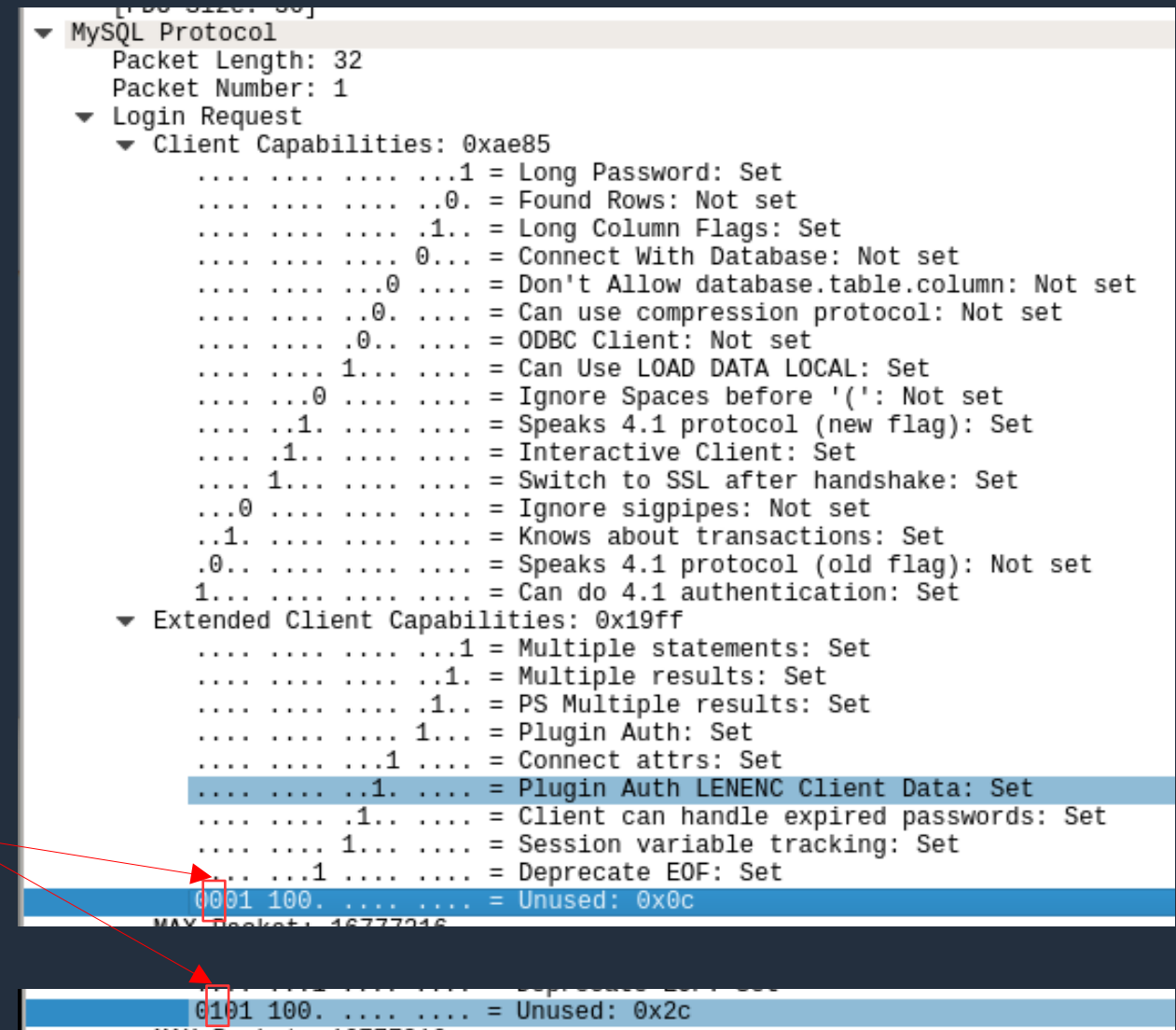
- The *server* greeting packet indicates to the client that the server knows how to handle the v2 handshake.
- The client reveals nothing in its plaintext greeting packet other than the fact that it wants to use TLS.

```
214 3796.5992149... 127.0.0.1 127.0.0.1 MySQL 72 Login Request
215 2706.5002207 127.0.0.1 127.0.0.1 TCP 66 3307 45500
▶ Frame 214: 72 bytes on wire (576 bits), 72 bytes captured (576 bits) on interface lo, id 0
▶ Ethernet II, Src: 00:00:00:00:00:00, Dst: 00:00:00:00:00:00
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 45598, Dst Port: 3307, Seq: 1, Ack: 93, Len: 6
▼ MySQL Protocol
  Packet Length: 2
  Packet Number: 1
  ▼ Login Request
    ▼ Client Capabilities: 0x0800
      .... 0 = Long Password: Not set
      .... 0 = Found Rows: Not set
      .... 0.. = Long Column Flags: Not set
      .... 0... = Connect With Database: Not set
      .... 0.... = Don't Allow database.table.column: Not set
      .... 0..... = Can use compression protocol: Not set
      .... 0..... = ODBC Client: Not set
      .... 0..... = Can Use LOAD DATA LOCAL: Not set
      .... 0..... = Ignore Spaces before '(': Not set
      .... 0..... = Speaks 4.1 protocol (new flag): Not set
      .... 0..... = Interactive Client: Not set
      .... 1... = Switch to SSL after handshake: Set
      .... 0..... = Ignore sigpipes: Not set
      .... 0..... = Knows about transactions: Not set
      .... 0..... = Speaks 4.1 protocol (old flag): Not set
      .... 0..... = Can do 4.1 authentication: Not set
```

CONC-656: Clients reveal if they can be undetectably MITM'ed

- Connector/C clients *reveal in plaintext* whether or not they are verifying the server's certificate.
- Protocol-level problem, *subset* of CONC-654
- **CONC-656**: Clients reveal if they can be undetectably MITM'ed

Wireshark showed this bit as “unused”
It *should* actually be labeled **Client verifies server certificate**
I fixed this in [wireshark MR!11498](#)



```
MySQL Protocol
Packet Length: 32
Packet Number: 1
Login Request
  Client Capabilities: 0xae85
    ...1 = Long Password: Set
    ...0 = Found Rows: Not set
    ...1 = Long Column Flags: Set
    ...0 = Connect With Database: Not set
    ...0 = Don't Allow database.table.column: Not set
    ...0 = Can use compression protocol: Not set
    ...0 = ODBC Client: Not set
    ...1 = Can Use LOAD DATA LOCAL: Set
    ...0 = Ignore Spaces before '(': Not set
    ...1 = Speaks 4.1 protocol (new flag): Set
    ...1 = Interactive Client: Set
    ...1 = Switch to SSL after handshake: Set
    ...0 = Ignore sigpipes: Not set
    ...1 = Knows about transactions: Set
    ...0 = Speaks 4.1 protocol (old flag): Not set
    ...1 = Can do 4.1 authentication: Set
  Extended Client Capabilities: 0x19ff
    ...1 = Multiple statements: Set
    ...1 = Multiple results: Set
    ...1 = PS Multiple results: Set
    ...1 = Plugin Auth: Set
    ...1 = Connect attrs: Set
    ...1 = Plugin Auth LENENC Client Data: Set
    ...1 = Client can handle expired passwords: Set
    ...1 = Session variable tracking: Set
    ...1 = Deprecate EOF: Set
    0001 100. .... = Unused: 0x0c
  MAY Packet: 16777916

0101 100. .... = Unused: 0x2c
  MAY Packet: 16777916
```


CONC-656: Clients reveal if they can be undetectably MITM'ed

- Many deployed MariaDB clients are *not actually verifying servers' TLS certificates* even if they are using TLS for encryption.
- Clients literally reveal whether or not they can detect the basic TLS MITM attack.
- Risks? ***Potentially massive.***
- If pervasive attackers already know about this vulnerability...
- They're already opportunistically decrypting tons of connections from MariaDB clients that aren't verifying certs...
 - ... without anyone noticing it.

CONC-656: Clients reveal if they can be undetectably MITM'ed

- Although this is a subset of [CONC-654](#)...
- This *can* be fixed *purely with a client-side change*.
- Could be done in one line, but I added 10 lines of explanatory comments
- [Connector/C PR#228](#) (submitted Jul 12)

```
plugins/auth/my_auth.c
@@ -297,10 +297,23 @@ static int send_client_reply_packet(MCPVIO_EXT *mpvio,
297 297
298 298     if (mysql->client_flag & CLIENT_PROTOCOL_41)
299 299     {
300 +     /* Whether or not the client can/will verify the server's certificate is
301 +     * not the server's concern. The client should never reveal this; varying
302 +     * it effectively tells a MITM attacker whether or not the client will be
303 +     * able to *detect* a MITM attack based on 2-sided TLS.
304 +     *
305 +     * We set this bit to OFF, not to ON, in order to make our output
306 +     * indistinguishable from older clients which vary this bit, so that those
307 +     * older clients become more difficult for MITM attackers to reliably target
308 +     * once this change is deployed.
309 +     */
310 +     ulong sanitized_client_flag= mysql->client_flag & ~CLIENT_SSL_VERIFY_SERVER_CERT;
311 +
312     /* 4.1 server and 4.1 client has a 32 byte option flag */
313     if (!(mysql->server_capabilities & CLIENT_MYSQL))
314         mysql->client_flag&= ~CLIENT_MYSQL;
315 -     int4store(buff,mysql->client_flag);
316 +     int4store(buff, sanitized_client_flag);
317     int4store(buff+4, net->max_packet_size);
```

MDEV-28634: Silently downgrade from TLS to no TLS

- Reported in **2020** (not by me)
- Connector/C clients using TLS will *silently* switch to a plaintext connection if the server doesn't support TLS
- ***Trivial downgrade attack***
- Exists in this form since at least 2015 ([this Con/C commit](#) or [this one](#))

Few-line fix for this in [Connector/C PR#224](#)
(submitted by me, June 15)

```
libmariadb/mariadb_lib.c
@@ -1918,6 +1918,16 @@ MySQL *mthd_my_real_connect(MYSQL *mysql, const char *host, con
1918 1918     }
1919 1919     }
1920 1920
1921 + /* We now know the server's capabilities. If the client wants TLS/SSL,
1922 +  * but the server doesn't support it, we should immediately abort.
1923 +  */
1924 + if (mysql->options.use_ssl && !(mysql->server_capabilities & CLIENT_SSL))
1925 + {
1926 +     SET_CLIENT_ERROR(mysql, CR_SSL_CONNECTION_ERROR, SQLSTATE_UNKNOWN,
1927 +         "Client requires TLS/SSL, but the server does not support it");
1928 +     goto error;
1929 + }
1930 +
1931 1931     /* Set character set */
1932 1932     if (mysql->options.charset_name)
1933 1933 +     mysql->charset= mysql_find_charset_name(mysql->options.charset_name);
```

MDEV-28634: Silently downgrade from TLS to no TLS

- Is this a technical problem, or a user experience problem?
- Neither `mariadb --help` nor the online docs mention that this option *might result* in a plaintext connection:

```
$ mariadb --help
--ssl          Enable SSL for connection (automatically enabled with
              other flags).
--ssl-ca=name  CA file in PEM format (check OpenSSL docs, implies
              --ssl).
--ssl-capath=name CA directory (check OpenSSL docs, implies --ssl).
--ssl-cert=name X509 cert in PEM format (implies --ssl).
--ssl-cipher=name SSL cipher to use (implies --ssl).
--ssl-key=name  X509 key in PEM format (implies --ssl).
--ssl-crl=name  Certificate revocation list (implies --ssl).
--ssl-crlpath=name Certificate revocation list path (implies --ssl).
--tls-version=name TLS protocol version for secure connection.
--ssl-verify-server-cert
              Verify server's "Common Name" in its cert against
              hostname used when connecting. This option is disabled by
              default.
```

```
--ssl
```

Enables [TLS](#). TLS is also enabled even without setting this option when certain other TLS options are set. The `--ssl` option does not enable [verifying the server certificate](#) by default. In order to verify the server certificate, the user must specify the `--ssl-verify-server-cert` option. Set by default from [MariaDB 10.10](#).

MDEV-28634: Silently downgrade from TLS to no TLS

- It's a massive violation of user expectations for the `--ssl` option to allow a downgrade to plaintext with no user input.
- If users ask for TLS/SSL, they *definitely don't want* a plaintext connection.

- The Jira submitter made this case clearly, 3 years ago:

The current behavior is probably the "expected" behavior according to Engineering. See [MDEV-16409](#) for some details on previous discussion. However, I don't think this behavior is the behavior that would be expected by most users. When implementing security features, the industry standard for design is to fail safe. Security features may be mandatory for compliance reasons, and the fault of a security control may silently increase risk.

- I've been making a fuss about it more recently.

It's too hard to use TLS securely with MariaDB: **Insecure defaults**

- MariaDB *server* accepts non-TLS clients by default (**REQUIRE_SECURE_TRANSPORT=OFF**)
- MariaDB Connector/C accepts non-TLS servers by default
 - ... and by default it won't even try TLS even if the server advertises it
- MariaDB Connector/Python is basically the same; very thin wrapper
- Yes, the `mariadb` CLI started defaulting to `--ssl` in 2022 ([MDEV-27105](#))
 - ... but still no server certificate validation by default
 - ... and [MDEV-28634](#) makes `--ssl` alone even more ineffectual.

It's too hard to use TLS securely with MariaDB: The "SSL" option really isn't

- Launching the client with `mariadb --ssl` ought to mean:
 - Connect to a server using TLS
 - Verify its certificate
 - Abort if the server doesn't support TLS or if you can't verify its cert
- What it actually means...
 - Connect to a server and use TLS if the server offers it
 - Don't verify the server's certificate (and make that clear to attackers, [CONC-656](#))
 - And silently fallback to plaintext if the server doesn't support TLS ([MDEV-28634](#))

It's too hard to use TLS securely with MariaDB: The "SSL" option really isn't

- If you actually want protection against the modern threat model, you need:

- `mariadb --ssl-verify-server-cert`

- Okay, what if you use `mariadb --ssl-ca=trustedCAcert.pem`?
- Does this imply `--ssl`, or `--ssl-verify-server-cert`?

```
$ mariadb --help
--ssl          Enable SSL for connection (automatically enabled with
              other flags).
--ssl-ca=name  CA file in PEM format (check OpenSSL docs, implies
              --ssl).
--ssl-capath=name CA directory (check OpenSSL docs, implies --ssl).
--ssl-cert=name X509 cert in PEM format (implies --ssl).
--ssl-cipher=name SSL cipher to use (implies --ssl).
--ssl-key=name  X509 key in PEM format (implies --ssl).
--ssl-crl=name  Certificate revocation list (implies --ssl).
--ssl-crlpath=name Certificate revocation list path (implies --ssl).
--tls-version=name TLS protocol version for secure connection.
--ssl-verify-server-cert
              Verify server's "Common Name" in its cert against
              hostname used when connecting. This option is disabled by
              default.
```

`--ssl`

Enables TLS. TLS is also enabled even without setting this option when certain other TLS options are set. The `--ssl` option does not enable verifying the server certificate by default. In order to verify the server certificate, the user must specify the `--ssl-verify-server-cert` option. Set by default from MariaDB 10.10.

It's too hard to use TLS securely with MariaDB: **Configuring certificates is too hard**

- Here's what the docs say:

Enabling TLS for MariaDB Server

In order to enable TLS on a MariaDB server that was compiled with TLS support, there are a number of system variables that you need to set, such as:

- You need to set the path to the server's X509 certificate by setting the `ssl_cert` system variable.
- You need to set the path to the server's private key by setting the `ssl_key` system variable.
- You need to set the path to the certificate authority (CA) chain that can verify the server's certificate by setting either the `ssl_ca` or the `ssl_capath` system variables.

- Technically quite accurate
- If you understand TLS *very well*, you might be able to configure MariaDB correctly based on this.
- But if not, it's very hard to succeed by experimenting!

It's too hard to use TLS securely with MariaDB: **Configuring certificates is too hard**

- Let's say we have a typical 3-layer cert chain
- `server-cert.pem` (signed by `ca.pem`, file includes private key)

```
subject=C = CA, L = Vancouver, O = Company, OU = Division, CN = mariadb-server.company.com
issuer=C = US, O = "Certy McCertface", OU = Intermediate Divison, CN = Certy McCertface Intermediate CA
notBefore=Apr 23 23:59:20 2023 GMT
notAfter=Apr 24 00:59:20 2060 GMT
```

- `ca.pem` (signed by `root.pem`)

```
subject=C = US, O = "Certy McCertface", OU = Intermediate Divison, CN = Certy McCertface Intermediate CA
issuer=C = US, O = "CertyCorp", OU = Root Division, CN = CertyCorp Root CA
notBefore=Apr 13 23:53:36 2023 GMT
notAfter=Apr 14 00:53:36 2061 GMT
```

- `root.pem` (*self-signed*)

```
subject=C = US, O = "CertyCorp", OU = Root Division, CN = CertyCorp Root CA
issuer=C = US, O = "CertyCorp", OU = Root Division, CN = CertyCorp Root CA
notBefore=Mar 31 23:52:11 2022 GMT
notAfter=Apr 1 00:52:11 2062 GMT
```

It's too hard to use TLS securely with MariaDB: **Configuring certificates is too hard**

- Things users might try

- Start the server:

- `mariadb --server-cert=server-cert.pem`

- `mariadb --server-cert=server-cert.pem --ssl-ca=ca.pem`

- `mariadb --server-cert=server-cert.pem --ssl-ca=server-cert.pem`

- `mariadb --server-cert=server-cert.pem --ssl-ca=root.pem`

- Start the client:

- `mariadb --ssl-ca=server-cert.pem`

- `mariadb --ssl-ca=ca.pem`

- `mariadb --ssl-ca=root.pem`

It's too hard to use TLS securely with MariaDB: **Configuring certificates is too hard**

- All of those server options will *start* the server without errors or warnings.
- All but one of those combinations of combination will result in client errors, either:
 - `ERROR 2026 (HY000): TLS/SSL error: unable to get issuer certificate`
 - `ERROR 2026 (HY000): TLS/SSL error: unable to get local issuer certificate`
- Will users understand these errors?
- Will they guide users towards *finding the right configuration*?
- Or will they just give up?

It's too hard to use TLS securely with MariaDB: **Configuring certificates is too hard**

- MariaDB is deferring to the configuration semantics and the error messages of the TLS library.
- Server *needs* to advertise a complete cert **chain**. Abort startup unless it is correctly specified.
- If the client can't verify a server's certificate, it should explain why clearly.
- Better application specific error checking and error messages.
- You specified a server certificate and private key, but not a complete certificate chain anchored in a self-signed root. See `--ssl-ca/--ssl-capath` options. Aborting server.
- Could not verify server's certificate using the root(s) of trust specified with `--ssl-ca`. **<Show unverified gaps in the chain.>**

Solutions

Improve the code

Improve the protocol

Backwards-compatibility should be less important than...

Actual security

Manageable complexity

Improve the code

- Vulnerabilities like CONC-648, CONC-654 + MDEV-31585, CONC-656 have been in the code for 10+ years
- I expect there are plenty of others.
- It's easy to find vulnerabilities.
- Most of these have taken me far longer to explain and advocate for, than to discover.
- The code for setting up TLS in MariaDB appears to be too complicated to easily maintain, and never simplified or audited.

Improve the protocol

- The switch-to-TLS protocol used by MySQL/MariaDB is *uniquely* bad.
- I've never seen another one that includes, and requires, sending so much redundant meaningful before and after the switch.
- There is no fix that will solve the security problems *and* preserve client/server compatibility across the ecosystem.
- It's going to have to be replaced.
- Better to do it sooner rather than later.

Improve the code *and* the protocol

- ... by merging my PRs! 🤪
- Connector/C PRs by dlenski, server PRs by dlenski
- In particular, [Connector/C PR#227](#) and [server PR#2684](#) for handshake information leakage.
 - I received a good amount of feedback early on
 - I responded to it and improved the PRs
 - I haven't had any *actionable* feedback on this in 2.5 months.

Backwards-compatibility

- I think backwards-compatibility is generally *very important*.
- But “backwards-compatibility” seems to be the common excuse for
 - Retaining a lot of insecure-by-default behavior for far too long.
 - Ossification of a lot of code.

Still, from a user point of view — they had MariaDB working, ssl enabled, so "everything was secure". Then after an upgrade the application is down, clients cannot connect anymore. Who broke user application (mission critical, of course) — we did.

You can change client, and server, but how much did you test that you did not break the protocol, and that MySQL clients (drivers) and 3rd party clients(drivers) can talk to MariaDB server, and that and that MariaDB clients (other than C) can talk to MariaDB servers or MySQL servers.

Breaking this is worse than alleged catastrophe leaking the extremely meaningful charset information which in 99.9% will be some form of UTF8.

Having software that is allegedly more secure, means nothing if your application does not work at all, because the client can't connect.

Changing a single bit in a protocol requires a huge effort of compatibility testing, that's a little more than "my patched client can talk to my patched server".

Backwards-compatibility isn't a good excuse

- “Everything is ***not*** working fine” for users who are using `mysql --ssl`, and silently downgraded to plaintext.
- They say they want the security of the TLS/SSL brand.
- ***But they are not getting it.***
- Many of them would be far happier for their connections to stop working with a new software version ...
- ... than to find out that their usage of MariaDB databases had been compromised for years due to “backwards-compatibility.”
- *Especially* if the new release includes clear help messages and documentation about what's changed, why, and how to adapt.

Manage complexity

- In MariaDB, there seems to be a strong tendency towards solving problems by making the software *more complex*.
- Existing feature has a problem? Add a new non-default option!
 - `--ssl` has no certificate verification? Add `--ssl-verify-server-cert`.
- Sometimes more complexity is necessary, but it's usually a *necessary evil*.
- The level of complexity in MariaDB in this area is unmanageable.

Simplify things

- *Better defaults* for options
 - Make `require_secure_transport=ON` *the default* in the server.
 - Make `--ssl-verify-server-cert` *the default* in Connector/C.
- Remove bad or obsolete options or features entirely
 - If there's a good replacement, inconveniencing some users is okay
 - **OpenConnect removed an option in 2016:**

```
$ openconnect --no-cert-check
The --no-cert-check option was insecure and has been removed.
Fix your server's certificate or use --servercert to accept a specific fingerprint.
```

Totally insecure against MITM

Use PKI correctly (scales)

Manual verification based on key fingerprints

Thank you

- Thanks for attending this.
- I know that this is a very critical take on MariaDB
- I'm highlighting these issues because I think they're important
- I think MariaDB can be a much better tool and product if they're addressed.

- Thank you to my colleagues for support, inspiration, and feedback on this presentation.

- Questions, discussion?

My question?

- What if you use `mariadb --ssl-ca=trustedCAcert.pem`?
 - Does this imply `--ssl`, or `--ssl-verify-server-cert`?
 - Is that implemented in the connector library, or in the client application?

Book recommendation

- “A Philosophy of Software Design” by John Ousterhout (2018)
- This is an amazing book, and not too long!
- Largely about managing complexity in software

You should avoid configuration parameters as much as possible. Before exporting a configuration parameter, ask yourself: “will users (or higher-level modules) be able to determine a better value than we can determine here?” When you do create configuration parameters, see if you can provide reasonable defaults, so users will only need to provide values under exceptional conditions.

Realize that **working code isn't enough**. It's not acceptable to introduce unnecessary complexities in order to finish your current task faster. The most important thing is the long-term structure of the system.