Typical problems and how to debug them

Originalartikel

Backup

<html> <hr/> Please note that republishing this article in full or in part is only allowed under the conditions described here. This guide tries to help with debugging of SSL/TLS problems and shows the most common problems in interaction between client and server. It is not intended to help with writing applications and thus does not care about specific API's etc. But it should help with problems outside of a specific API, like different or broken SSL stacks or misconfigurations. The guide is based on the knowledge gained as the maintainer of the IO::Socket::SSL Perl module or by debugging SSL problems at work or for fun. Unfortunatly SSL/TLS is a hard to debug protocol because: Error messages are missing, are not very specific or even hide the real problem. There are lots of broken configurations and SSL stacks in the wild. And while browsers try to work around it as much as possible the stacks in applications or scripts are mostly not that tolerant. There are lots of bad tips out there which often only work around the underlying problem by seriously degrading the security of the protocol. Deeper knowledge of the protocol and standards is necessary to understand and fix most problems instead of applying some insecure workaround found somewhere on the internet. <hr/> Basic information <li class=„c1“> Useful/required knowledge Common misunderstandings about SSL/TLS Security relevant errors which don't cause obvious problems Start with debugging <li class=„c1“> Useful tools for debugging The usual steps in debugging How to check for common problems Commonly seen and more unusual problems <li class=„c1“> Common problems caused by SSL stacks at server, client or middlebox Common problems caused by misconfiguration Problems due to bad certificates Problems caused by inconsistent handling of root certificates More unusal but existing problems Finding and fixing the problem <li class=„c1“> Problem solving by error message or symptom When it worked before, works with other applications, servers … It still does not

work <hr/> <h2>Useful/required knowledge</h2> While SSL/TLS is a complex protocol there a some basics one should understand in order to debug and fix most problems: SSL/TLS provides encryption and identification. Encryption without proper identification (or a pre-shared secret) is insecure, because Man-in-the-middle attacks (MITM) are possible. Identification is mostly done with certificates: Builtin trust anchors (Root-CA) in the application (e.g. browser, mobile app, ...). The server provides its own certificate and the intermediate certificates (trust chain) leading to the trust anchor. A similar mechanism can be used to authenticate the client too (client certificates). The servers certificate must match the expected identity, i.e. usually the hostname. For HTTPS see RFC 2818 and CA Browser Forum Baseline Requirements for details, for other protocols see RFC 6125.   Certificate/public key pinning can be used as an alternative to local trust anchors In this case the application knows up-front the fingerprint of the certificate or embedded public key. This fingerprint is hard-coded into the application. A lesser secure alternative saves the fingerprint on the first connect to the peer. Of course this can not detect if an MITM attack is already done on the first connect and then trust the attacker for future connections. There are different versions of the protocol (SSL 3.0, TLS 1.0...TLS 1.2), each fixing design flaws in the previous version or adding features. TLS 1.0 is in reality SSL 3.1, but the name of protocol has been changed. TLS extensions like Server Name Indication (SNI) can only be done with TLS1.x. SSL 3.0 is considered broken (POODLE) and should no longer be used. Cipher suites decide about methods for authentication, encryption ... Cipher suites are mostly independend of the protocol version. The version only specifies when this cipher was introduced: There are no TLS1.0 or TLS1.1 cipher suites, but TLS1.2 added some. SSL3.0 ciphers are still used in TLS1.x Ciphers vary in their strength and there are weak ciphers which should no longer be used. There are lots of resources about the optimal ciphers, one of them is Mozilla. Before the encryption starts the peers agree to the protocol version and cipher used within the connection, exchange certificates used for authentication and exchange the keys for encryption. Almost all of the problems occure within this initial handshake. <h2>Common misunderstandings about SSL/TLS</h2> I only want encryption and don't like all this certificate stuff: Any encryption without identifcation (or a shared secret) is open to MITM attacks. A self-signed certificate is secure enough: True, but only if the certificate is trusted up-front in the application, like with certificate/public key pinning. I want TLS, but not SSL: TLS1.0 is SSL3.1, that is they changed the name of the protocol. In the context of SMTP, IMAP or FTP, „SSL“ is often used to describe SSL/TLS from start, while „TLS“ is used to describe upgrade to SSL/TLS after some kind of STARTTLS command. It is better to use „implicit“ and „explicit“ SSL/TLS here. Disabling SSL3.0 (because of POODLE) can be done by disabling all SSL3.0 ciphers: Not really, because these ciphers are needed for TLS1.x too. You should disable the SSL3.0 protocol instead. <h2>Security relevant errors which don't cause obvious problems</h2> These kind of problems are not obvious, because everything seems to work fine. But they open ways for attacks and thus need to be fixed. Unfortunatly, often these kind of problems

are caused by an attempt to fix another problem and by not understanding the security implications of the applied workaround. Use of insecure protocols or features: SSL2.0, SSL3.0 are broken and should not be used. Other attacks are possible by using insecure renegotiation, compression … . For details see Wikipedia Use of insecure implementations: In 2014 all major TLS stacks where affected by serious implementation problems: OpenSSL Heartbleed, Apple Secure Transport goto fail, Microsoft SChannel Winshock, and certificate verification problems with GnuTLS Insecure certificate checks: Due to insecure defaults in lots of programming languages (Python, Ruby, PHP, Perl…) or libraries, certificates are either not verfified at all or only the trust chain is verified but not the hostname against the certificate. This gets only slowly fixed because the developers fear to break existing code. Because proper certificate checking is often in the way of testing, lots of iOS- and Android developers explicitly disable these checks and fail to enable checks in production version. Lots of applications don't have proper hostname checks, i.e. accept wildcards anywhere or multiple wildcards or even check the subject against a regex. Sometimes these checks are too broad, but in some cases they are too narrow (missing check of subject alterative names) so users disable checks completely. Use of insecure ciphers Some application just accept 'ALL', which includes very weak cphers (EXPORT, LOW) and also anonymous cipher suites (with no authentication) which make MITM easy. Others allow or even require broken ciphers like DES-CBC-SHA or RC4-SHA.   <h2>Useful tools for debugging</h2> Often an error message alone is not sufficient to solve the problem. In this case the following tools can be of help: SSLLabs can be used to check problems with public accessible HTTPS servers. It shows problems about certificate verification and also about potential problems with specific TLS clients. In case it is not https or the server is not public accessible analyze.pl from my SSL tools can help. It can be used to debug TLS problems with plain TLS or explicit TLS on SMTP, IMAP, POP3 and FTPS and with HTTP proxies. openssl helps with debugging too, especially with the s_client, s_server and x509 commands. And wireshark can be used to analyse packet captures done by tcpdump or wireshark. It is able to show lots of details about the TLS handshake. <h2>The usual steps in debugging</h2> The steps shown here are useful to solve the problem. Even if one can not solve the problem by oneself by using these steps it is recommended to do as much of them as possible and provide the collected information to anybody willing to help. Chances are much higher that they will then look into the problem. Collect error messages and compare them with the solutions/descriptions below. Do any of the proposed tools show information which might explain the problem? Narrow down the problem to the client or the server or something in between, i.e. Try to access the same server from different clients (browsers, apps, …). Try to access the same server from different networks. If possible access server from the servers machine or at least from the servers local network. Try to access different servers from the same client. Check for known problems with the SSL stack used by the affected application. If the affected part is one's own application: try to strip it down as much as possible and remove any customization which might cause the problems. If other peers work: look at their traffic and try to restrict protocol version,

ciphers to emulate their traffic. If still not resolved: provide anybody willing to help with the collected information and also with debug information and a packet capture in a form usable by wireshark. Also provide information about the used SSL stacks (i.e. browser or application version, programming language version, OS version). WARNING: while you might disable verification or downgrade ciphers or protocol to insecure versions to track down the problem do not simply leave it this way once you've „fixed“ the problem this way. Instead track down the cause of the problem and fix it, especially: Fix certificates if verification failed due to bad or self-signed certificate. If this is not possible use certificate/public key pinning to accept only this bad certificate. Don't restrict yourself to bad protocol versions or ciphers, even if these solve the problem at the moment. There will be a time when the peer will be upgraded and then you will have problems again. This happened a lot when SSL 3.0 got disabled (POODLE attack) and lots of clients suddenly failed to connect, because they had hard-coded use of SSL 3.0 in their application. <h2>How to check for common problems</h2> How to check if server requires SNI Use 'openssl s_client' with and without '-servername' option. If the returned certificates differ then SNI is required. Some servers even fail completly when accessed without SNI. SSLLabs will also tell you if the site requires SNI („This site works only in browsers with SNI support“). Use analyze.pl, it will tell you if different certificates are returned with and without SNI.   How to check for missing chain certificates SSLLabs will tell you if the chain is incomplete („Chain Issues“) and will try to show the missing intermediate certificates. analyze.pl –show-chain will show the chain too, but not the missing certificates.   How to check for trusted Root-CA SSLLabs will check if one of the common CA is used as the trust anchor. analyze.pl will check against system CA (or Mozilla's CA on Windows and Mac OS X), but can also check against a certificate store specified by the user. 'openssl s_client' can check against a given CA. But it will in this case also check against OpenSSL default CA's too, so the result can be misleading.   How to check using a client certificate analyze.pl can be given a client certificate. 'openssl s_client' can also use client certificate.   How to check which ciphers and protocols are supported by the server. SSLLabs will show the available ciphers and protocols and also emulate the behavior of specific clients to see if a connection should be successful or why not. Please check that their tests use the same IP address as you do, notably SSLLabs currently does not support IPv6 addresses. analyze.pl –all-ciphers shows which ciphers of the locally installed OpenSSL are supported by the peer. It will also show if the server chooses the cipher based on clients or servers preferences. It also shows protocol support.   How do I perform the checks if explicit TLS is used (STARTTLS etc) analyze.pl supports SMTP, IMAP, FTP, POP3, HTTP proxy and PostgreSQL with the '–starttls' option. 'openssl s_client' supports SMTP, IMAP, FTP and POP3 with the '-starttls' option.   <h2>Common problems caused by SSL stacks at server, client or middlebox</h2> No SNI support for SSL 3.0, Android (depending on application), MSIE on XP, Java 6 and various other programming languages. This will cause problems , when the server has multiple certificates on the same IP address (like Cloudflare Free SSL). It will usually result in certification errors because the

wrong certificate is received. But in some cases the server will also just close the connection or issue an alert or similar, depending on servers configuration and TLS stack. The fix is to upgrade to a version which supports SNI. Workaround at the server is to have a separate IP address for the affected certificates. SNI is not supported by Internet Explorer 8 and older versions. If the system can not be upgraded an alternative browser like Firefox, which is not using SChannel, can be used. The Apache HTTPClient library as used in Android does not support SNI. For workarounds see here. No SNI in Java 6 and lower, Python 2 (until 2.7.8) and older versions of other programming languages or packages. No workarounds for the client is known, that is an upgrade is required.   F5 Big IP: TLS handshake times out, because of no response to ClientHello. Older versions of F5 Big IP simply absorb ClientHello with a size between 256 and 511 bytes. Because TLS 1.2 offers more ciphers this mostly happens with TLS 1.2 handshakes, but was also seen with TLS 1.1. Workaround is to reduce the number of ciphers offered by the client. Fix is to patch the device. Newer versions of OpenSSL contain a workaround SSL_OP_TLSEXT_PADDING which can break IronPort instead. wget <1.12: checks hostname only against commonName, not against Subject Alternative Names. Fix is to upgrade wget. phantomjs currently defaults to SSL 3.0, which gets more and more disabled by the servers because it is insecure. Use `–ssl-protocol=any` to use more recent versions of TLS. Some servers are broken and don't support the most common SSLv23 handshake. But cURL (at least version 7.41 with OpenSSL backend) will try an SSLv23 handshake in all cases, except when use of SSL 3.0 is explicitly requested. Other clients instead can instead do a TLS1.0 only handshake. If `openssl s_client` is used with the `-CAfile` option it will not only check against the certificates given in this file but additionally against the system defaults. Thus the result might be different between various systems (http://stackoverflow.com/a/29115499/3081018][especially UNIX and Windows) because the defaults differ. <h2>Common problems caused by misconfiguration</h2> Server allows only allow bad ciphers, like RC4-SHA. Some clients like curl 7.35.0 have disabled these ciphers by default (see workaround) and there are recommendations for others like Microsoft Windows. Administrators tried to make systems safe against POODLE by disabling all SSL 3.0 ciphers instead of the protocol version. Because these ciphers are needed for TLS1.0 and TLS1.1 clients, at most TLS1.2 clients could connect. <h2>Problems due to bad certificates</h2> Bad certificates are a very common error. The most common problems are: Self-signed certificates. In this case the trust can not be checked against a local trust anchor and thus the certificate can not be trusted. Browsers allow the user to explicitly trust the certificate. Certificate contents does not match hostname. There are clear rules how the checks should be done, but some applications are less strict and others implement the checks wrong: IP addresses should be stored as type IP in Subject Alternative Names (SAN) section. Most browsers currently accept IP in commonName too, but Safari does not. But for MSIE IP addresses have to be specified as DNS type inside the SAN section. Wildcards are only allowed in Subject Alternative Names section. Most browsers currently accept wildcard in commonName too, but not Safari. If a SAN section contains entries of type DNS than commonName should not be checked. Most browsers currently check commonName too, but Safari does not. Other applications do not check commonName, even if SAN section only contains entries for

IP addresses.   In these cases either the certificate need to be fixed or the application must import the certificate as trusted or use certificate/public key pinning. Less common errors are: Certificate expired or not yet valid. Insecure certificates with a too small RSA key length or MD5 signatures. Most software does not accept these certificates anymore. Some (or all?) browser require the extKeyUsage of serverAuth inside the certificate, while most script languages ignore any usage restrictions. <h2>Problems caused by inconsistent handling of root certificates</h2> Each SSL stack has its own way to handle the trust anchors (the root certificates). Even different applications using the same stack often do not share the same root certificates: Mozilla Firefox (NSS library) comes with its own root certificates and can manage trust for each profile separatly. Chrome uses the NSS library too (except on Android), but integrates with the systems CA store on Windows and Mac OS X. On Linux platforms it uses its own trust store which is shared between different Chrome accounts („People“). Internet Explorer on Windows and Safari on Mac OS X use the systems CA store. Java comes with its own CA store. Python, Ruby, PHP, Perl… can behave in different ways, depending on language version. Even packages inside these languages might have their own rules: They might integrate with the OpenSSL CA store. This works on UNIX, but on Windows this will mostly result in verification errors, because there is no OpenSSL CA store. To get usable Root-CAs check here. They might come with their own CA store. They might even try to integrate with the systems CA store on Windows. <h2>More unusal but existing problems</h2> Mac OS X hacks into OpenSSL to verify against systems key store if nothing is found in OpenSSL key store. Thus verification might succeed if failure was expected. Some commonly used AntiSpamProxy just closes connection when it receives a MD5-signed client certificate within a TLS1.2 connection. Using TLS1.1 or SHA-1 instead is no problem. At least some versions of HP ILO2 cause a handshake failure with „bad record mac“ when used with TLS1.x. Workaround is to use only SSL3.0. Some SSL stacks claim to support more ciphers or elliptic curves than they actually have implemented. This might be due to misconfiguration, incomplete disabling of specific features at compile time or bugs. See this and that where you get „elliptic curve routines: EC_GROUP_new_by_curve_name: unknown group“ in the client. And in this case the server just closes the connection. Workaround is to disable the affected ciphers on the client side. The Perl package LWP::UserAgent changed with version 6.0 (03/2011) the TLS backend from Crypt::SSLeay to IO::Socket::SSL but the https proxy support was broken until version 6.06 (04/2014). Before that fix you usually got „Bad request“ or similar back from the proxy. Python 3 might send a zero-length server name extension (SNI), causing tlsv1 alert decode error. Cross-Signing of CA certificates can result in multiple possible trust chains, depending on which chain certificates the server is sending. Different SSL stacks behave differently when verifying these chains, which can result in verification errors on Windows or <a href=„http://kriscience.blogspot.de/2013/03/supporting-trusted-but-untrusted.html“

target=„_blank“>with OpenSSL. <h2>Problem solving by error message or symptom</h2> TCP connection failed or timed out: This is no TLS problem at all. In this case no TCP connection is possible to the peer, because the peer might be down, a firewall in between or similar. Make sure that it is a really at the TCP level by using telnet or similar tools. certificate verify fail Client with known verification problems? Is SNI required by server, like with Cloudflare free SSL? Does client support SNI? Check for missing chain certificates. Desktop browsers might work with missing chain certificates since they cache these from previous sessions to other sites and also sometimes load them by URL given in related certificates. Firefox does not do this, but Chrome and MSIE might do it. Other applications usually don't do this. Is the certificate valid at all? Invalid local time might cause reports about expired or not yet valid certificates. SSL interception inside a company will cause to be signed by a proxy CA. Verification will fail if this CA is not trusted by the application. Verification might even fail in case of SSL interception if the proxy CA is trusted, because the application uses certificate/public key pinning. While most browsers ignore the pinning if the certificate is signed by a CA which was explicitly added by the user, pinning using EMET on Windows might not make this exception. The needed Root-CA might be known on the system, but maybe not in the trust store used by the specific application. no shared ciphers Check support ciphers by client and server. Typical problems are Misconfiguration because all SSL 3.0 ciphers got removed. Server uses old ciphers which are no longer supported by client, or the other way. No certificates are configured at the server, which then falls back to anonymous authentication. These ciphers are not supported by most clients for security reasons (MITM).   unknown protocol This happens if the peer does not speak TLS at all, typically by attempting TLS against port 80 (non-TLS), by trying to access an SMTP server neeeding explicit TLS (STARTTLS) using implicit TLS or by accessing a badly configured server which provides plain http instead of https on port 443. This can also happen if server and client have no protocol versions in common. SSL handshake timed out, „want read“ This can be some bad middlebox like here. Retry from another network, with different TLS versions or less ciphers. Or it might be that the peer does not speak TLS at all and just waits for more data. „connection closed“ or „connection reset by peer“ or „handshake failure“or „error 40“ or „SSL_connect SYSCALL …“ Might be lot of different things, like SChannel (Microsoft) peers often do not send a TLS alert back on errors, but simply close connection. In this case it would be helpful to check at the peer side for error messages. Peer might have crashed and thus connection got closed. The problem has been seen when client uses SNI but server has no configuration for the provided name (misconfiguration server or DNS). The problem has been seen when client does not use SNI but server requires SNI (bad server, should send alert back). It was seen when the client provided an unexpected certificate, or provided no certificate even if server requested one. Or some other broken client.

Server requires SNI and will even fail with handshake if SNI is not used. See here for how to check for SNI. elliptic curve routines:EC_GROUP_new_by_curve_name:unknown group If ECC is used (like with ECDHE ciphers) the client needs to announce the supported ECC curves. If it does not announce any the server is free to pick any curve, which then might not be available on the client. See here for more information. bad record mac This might be someone tampering with the traffic. But more likely it is some old and broken server like HP ILO2. fails because of OCSP problem Unfortunatly OCSP responders sometimes return bad, expired or no responses, which makes reliance on OCSP a problem. Also, some web servers provide expired OCSP responses with OCSP stapling. This usually happens only if one enforced strict OCSP checking, e.g. by setting „security.OCSP.require“ in Firefox. Workaround: disable strict OCSP checking. But this degrades security. tlsv1 alert decode error Client probably sends improper messages, like a zero-length server name extension. I got a mail about my application, referencing VU#582497 Your application has broken certificate validation. You probably did this because you got certificate problems while testing you saw some post online on how to „fix“ your problem by disabling verification you just used the code you've found there without understanding the implications Now you have to fix your code, by just remove any custom validation you have done if you get errors now use the correct certificates at the peer alternativly use certificate/public key pinning   <h2>When it worked before, works with other applications, servers …</h2> it worked before the browser got upgraded Firefox removed some RSA 1024 Root-CA's recently, other browsers still have them. Some certificates might affected by this and should be replaced. More and more browser disable SSL 3.0 by default. it worked before the local system got upgraded Programming languages like Python, PHP, Ruby, Perl and probably others moved or in the process of moving to proper verification of TLS by default. Proper validation was also added to other tools. This will affect code which implicitly expected no verification. Fix your code to expect proper verification. Disable verification only if these are just test scripts which don't work with sensitive data. Curl disables RC4 ciphers by default with version 7.35.0. Sites which don't support better ciphers will no longer work. Workaround: RC4 support is still there, but has to be explicitly enabled. Perl LWP::UserAgent moved from Crypt::SSLeay to IO::Socket::SSL as SSL back end and thus checks certificates much more rigourous. This might cause problems when no or lazy validation was expected. It might also give problems when proxy is in use. In this case an upgrade to 6.06 for both LWP::UserAgent and LWP::Protocol::https is needed. it worked before the server configuration changed The server might require SNI now, but client might not support SNI. The server might no longer support the

protocol or ciphers used by the client. This affects especially clients on old platforms or clients with hard-coded protocol versions or ciphers. Typical examples are disabling of SSL 3.0 because of POODLE or disabling RC4 ciphers. Also some servers disabled all SSL 3.0 ciphers in a flawed attempt be save against POODLE. The server might have changed the certificate and forgot to send the new chain certificates. it worked yesterday, last week.... Then probably some of the events described above happened. Or the certificate of the server expired (or the local time is wrong and it looks only expired). it works in desktop browsers but not on Android/iOS/script/other application. Then it is probably either an incomplete certificate chain. Or the server requires SNI but your app does not support it. Also, desktop browsers retry the connection with a lower protocol version on most errors, while other application mostly don't automatically downgrade. it works on the same computer at home If the problem shows up as invalid certificate: If you are in a company then SSL interception is probably done for security reason. In this case the certificate is signed by your company or some firewall vendor and not by the original CA. You need to import the relevant CA into your browser as trusted if you want to accept the interception. Sometimes such interception is also done for the initial connection (to a landing page) at some WLAN hotspots. If (TCP) connection fails: there is probably some firewall which blocks the connection. These might be for TCP connections on specific ports so that all traffic on this port fails, but it might also be restricted to only selected target hosts. it works on other similar systems Even if two systems have the same OS and upgrades they might behave differently: Additional trusted Root-CAs might be installed on the system where the connection is successful. The failing system might have disabled protocols like SSL 3.0 or disabled ciphers like RC4, while the other system did not. These settings might by system wide or browser specific.   it works in other browsers The browsers differ in Root-CA's, supported ciphers and protocols. See here for how to solve the problem by the error message or symptom. The browsers might also have different network settings, i.e. which proxy gets used. It works on some systems but not on others, sometimes works on similar systems sometimes not Check if all these systems and application access the same server IP. I've seen problems where the server had several IP's for the same hostname but with different configurations, like different between IPv4 and IPv6. Some setups show erratic behavior, which might be cause be a load balancer with several systems behind, where some of the systems have a different configuration from the rest. <h2>It still does not work</h2> Check for the symptoms and error messages at stackoverflow. If nothing helpful is found ask a new question there. But, don't forget to provide as much information as possible to get a useful response. </html>