ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA TRIENNALE IN INGEGNERIA E SCIENZE INFORMATICHE

Valutazione della sicurezza delle comunicazioni con i principali istituti di credito online

---

Relatore:
Gabriele D'Angelo

Presentata da:
Carlo Alberto Scola

Tesi di Laurea in Programmazione di Reti

Sessione I
Anno Accademico 2016-2017

«Companies spend millions of dollars on firewalls, encryption and secure access devices, and it's money wasted, because none of these measures address the weakest link in the security chain.»

Kevin Mitnick

Introduzione

Dato l'inglese come lingua principale globalmente riconosciuta nell'informatica, nel corso di questa trattazione mi limiterò, per non alterarne il significato, a non tradurre in italiano alcuni termini i quali verranno lasciati in corsivo.
Attualmente i *web server* e le *web application* costituiscono una parte fondamentale di internet, ormai quasi la totalità dei servizi, enti ed aziende si basa sul loro utilizzo facendo transitare quindi qualsiasi tipo di informazione sulla rete. Informazioni che possono essere classificate in base al loro livello di importanza/segretezza, quindi di valore, il più delle volte, non indifferente.
Ma come vengono protette queste informazioni?
Non essendo la mente umana perfetta, qualsiasi algoritmo e metodologia sviluppata può a sua volta essere (da una mente più astuta di chi l'ha progettata) piegata ed utilizzata per scopi fraudolenti o comunque non previsti al momento della sua concezione.
Motivo per cui millenari e tuttora irrisolti dibattiti fra queste due fazioni si svolgono senza riuscire a trovare una soluzione definitiva.
La crittografia è una tecnica che ha apportato benefici indispensabili, una materia che applicata alle comunicazioni di oggi, ieri e domani è in grado di garantire, in base agli algoritmi utilizzati, un livello più o meno avanzato di protezione delle informazioni.

Algoritmi e funzioni crittografiche vengono quindi sempre più studiate e perfezionate per cercare di salvaguardare la confidenzialità, l'integrità e l'autenticazione di una comunicazione tra due host, che possono essere smartphone, server, siti di e-commerce, istituti di credito, siti governativi e così via.
In questa tesi parleremo del protocollo di comunicazione HTTP ma soprattutto approfondiremo la sua versione "sicura", detta HTTPS. Lo scopo è di sviluppare un tool per l'analisi delle comunicazioni SSL/TLS e utilizzarlo verso alcuni istituti di credito italiano per individuare ed identificare errate configurazioni e/o possibili miglioramenti.

Indice

Introduzione **v**

1 Introduzione HTTP/S **1**

1.1 HTTP . . . 1

1.1.1 Web Server . . . 2

1.1.2 Browser Web . . . 3

1.1.3 Web Application (In)Security . . . 4

1.1.4 Cookie HTTP . . . 6

1.1.5 HTTP Connection . . . 7

1.1.6 Problematiche di HTTP . . . 8

1.2 HTTPS . . . 9

1.2.1 Funzionamento generale e cenni di crittografia . . . 10

1.3 Importanza delle comunicazioni oggi . . . 12

2 Public Key Infrastructure **15**

2.1 Introduzione . . . 15

2.1.1 PKI Vista d'insieme . . . 17

2.2 Certificati . . . 18

2.2.1 Struttura . . . 19

2.2.2 Catene di Certificati . . . 20

2.2.3 Ciclo di vita di un certificato . . . 21

2.2.4 Revoca dei Certificati . . . 21

2.3 Importanza della root CA . . . 22

2.3.1 Debolezze della PKI . . . 23

2.4 PKI possibili miglioramenti . . . 25

2.5 Attacchi alla/alle Certification Authority . . . 27

3 Cenni di crittografia **29**

3.1 Crittografia Simmetrica . . . 29

3.1.1 Stream Cipher . . . 30

3.1.2 Block Cipher . . . 31

3.1.3 Block Cipher Modes . . . 31

3.2 Hash Functions 33
3.3 Message Authentication Codes (MAC) 35
3.4 Random Number Generation 36
3.5 Firma digitale 36
3.6 Conclusione 37

4 SSL e TLS **39**
4.1 Record Protocol 39
4.2 Handshake Protocol 41
4.2.1 Full Handshake 41
4.2.2 Client authentication 47
4.2.3 Session Resumption 47
4.3 Key Exchange 48
4.3.1 RSA 48
4.3.2 Diffie-Hellman Key Exchange 49
4.3.3 Elliptic Curve DH Key Exchange 50
4.3.4 Elliptic Curve Digital Signature Algorithm 50
4.4 Application Data Protocol 51
4.5 Alert Protocol 51
4.6 Cifratura 52
4.7 Rinegoziazione 55
4.8 Operazioni crittografiche 56
4.9 Cipher suites 58
4.10 TLS Extensions 59
4.11 Differenze tra varie versioni SSL/TLS 62
4.11.1 TLSv1.3 64

5 Attacchi conosciuti a SSL/TLS **67**
5.1 Insecure renegotiation 67
5.2 BEAST 69
5.3 Compression attacks 71
5.3.1 CRIME 72
5.3.2 TIME 72
5.4 Lucky 13 73
5.5 POODLE 75
5.6 Heartbleed 76
5.7 Conseguenze della Export-Cryptography 77
5.7.1 FREAK 78
5.7.2 Logjam 78

5.8 Triple handshake attack . . . 79

6 TLSScan **83**

6.1 Funzionalità . . . 83

6.2 Requisiti . . . 84

6.2.1 Analisi dei requisiti . . . 85

6.3 Struttura . . . 85

6.4 Descrizione dello strumento . . . 86

6.5 Analisi del problema . . . 88

6.5.1 Protocolli supportati . . . 88

6.5.2 TLS Extension e compressione . . . 89

6.5.3 HTTP Strict Transport Security . . . 90

6.5.4 Cipher suite . . . 90

6.6 Implementazione . . . 91

6.6.1 tlsscan . . . 91

6.6.2 TLSScanner . . . 92

6.7 Esempi di utilizzo . . . 94

6.7.1 Esempio 1 . . . 94

Considerazioni . . . 96

6.7.2 Esempio 2 . . . 96

Considerazioni . . . 98

6.7.3 Esempio 3 . . . 98

Considerazioni . . . 100

6.7.4 Esempio 4 . . . 100

Considerazioni . . . 103

6.7.5 Esempio 5 . . . 103

Considerazioni . . . 106

6.8 Valutazioni finali . . . 107

Conclusioni **108**

Bibliografia **113**

Capitolo 1

Introduzione HTTP/S

1.1 HTTP

HTTP, acronimo di *Hypertext Transfer Protocol* utilizzato per accedere al *World Wide Web*, è un protocollo applicativo tra i più utilizzati al mondo. Il suo sviluppo fu iniziato da Tim Berners-Lee al CERN nel 1989, proseguì con la prima definizione di HTTP/1.1 nel 1997 (RFC 2068) andando poi verso un successore degno di nota quale è HTTP/2 nel 2015 (basato su SPDY). ([66])

HTTP è sostanzialmente un protocollo semplice, utilizzato inizialmente per trasferire piccole porzioni di dati come testi statici. Successivamente ha esteso le sue funzionalità fino ad essere in grado oggi di offrire un mezzo per trasportare contenuti di ogni genere e dimensione e non solo. Oggi interi applicativi girano lato server e interagiscono con l'utente mediante web browser, ovvero, il tutto sotto HTTP. Ovviamente data la necessità dirompente con la quale le informazioni che transitano su internet debbano essere protette da occhi indiscreti ne esiste una versione "sicura" detta HTTPS, della quale parleremo fra poco.

È un protocollo che segue un approccio *message-based* nel quale, come prima cosa, c'è sempre un *client* che invia una richiesta ad un *server* il quale a sua volta risponde. Questo approccio si chiama *pull*. L'ultima versione, la 2.0, tra le tante permette anche un approccio parzialmente*push* da parte del server, funzionalità rivoluzionaria per HTTP, il quale ha ora le capacità di inviare risorse al client prima che quest'ultimo le richieda. Questo permette, soprattutto con la diffusione online di contenuti sempre più dinamici, di ridurre notevolmente la latenza della connessione e di offrire quindi una velocità di navigazione ed una esperienza di navigazione nettamente superiore.

Inoltre, utilizzando un protocollo di trasmissione affidabile come lo è TCP, HTTP garantisce che ogni singolo pacchetto di informazione non venga danneggiato ne perduto indipendentemente dalla locazione geografica dei due estremi della connessione. L'utente potrà stare tranquillo e non doversi "mai preoccupare dell'integrità dei dati". Tra virgolette poiché l'integrità o meno della comunicazione è dipendente da diversi fattori che vedremo bene più avanti.

1.1.1 Web Server

I contenuti del Web risiedono nei cosiddetti *web server* ovvero applicazioni eseguite su server reali o virtualizzati in grado di gestire un gran numero di richieste di trasferimento file mediante protocollo HTTP-HTTPS. I *web server* possono essere di due tipi:

- *Static web server*: Server invia copie dei file che risiedono nel suo file system, "così come sono".
- *Dynamic web server*: I contenuti oltre a file statici possono essere generati dinamicamente da software che tiene conto ad esempio dell'identità dell'utente, di cosa è stato immesso in input e dell' orario della giornata. Infatti, una classe di web server chiamati *application server* connettono web server a sofisticate *backend application* come ad esempio database, software gestionali, etc.

Ad oggi ci sono molteplici implementazioni diverse di *web server*. Tra i più popolari troviamo:

- *Apache HTTP Server*: Il più famoso ed utilizzato web server, anche se in calo, possiede ancora la fetta maggiore di utilizzo.
- *Apache Tomcat*: Java Servlet Container, un Java HTTP server.
- *Nginx* : In costante e ripida ascesa è NGINX, un web server che ha come punto forte quello di offrire funzionalità high performance di *reverse proxy*, *load balancer* e *HTTP cache* .
- *Internet Information Services (IIS)*: Web server prodotto da Microsoft.

Ogni risorsa generica (che sia un immagine o un indirizzo web, un documento, un servizio, ecc) viene identificata da una stringa specifica, **URI** (*Uniform Resource Identifier*) [66], che rende disponibile la stessa in internet.
Il più famoso *resource identifier* è **URL**, *Uniform Resource Locator* [66] che dato

in pasto ad HTTP permette di effettuare un *GET* (di richiedere) quella esatta risorsa su quello specifico web server. Poiché un URI definisce un *path* per una particolare risorsa, alcuni tipi di URI permettono di accedere a diversi servizi mediante diversi protocolli (detti "schemi"). Questo è un esempio di come alcuni URI possono essere utilizzati:

<scheme>://<user>:<password>@<host>:<port>/<path>;<params>?<query>#<frag>

Ad esempio:

- *ftp://user:password@ftp.example.com:2021* – per servizi FTP
- *mailto:myemail@example.com* – per posta elettronica
- *http://www.example.com/page/news-24.html* – per servizi HTTP
- *https://www.myhomebanking.com/login* – per servizi HTTPS

1.1.2 Browser Web

Il browser è l'altro punto fondamentale per la comunicazione oltre al web server. Il suo scopo è quello di permettere all'utente di navigare e dunque di mostrare pagine web, ovvero ciò che viene scaricato dal sito in questione, sotto forma di testo leggibile, coordinato e con tutti gli stili che lo rendono esteticamente gradevole o meno. È il browser stesso che interagendo con il sistema operativo si occupa di stabilire la connessione con il web server ed inoltre essendo le pagine web state costruite mediante diversi linguaggi di programmazione, è compito suo elaborare tutte queste informazioni per creare il risultato finale.

Per permettere la dinamicità ad un sito web viene utilizzato soprattutto il linguaggio Javascript, che viene integrato nelle pagine web ed eseguito *client-side* dal browser stesso. Questo dà agli sviluppatori uno strumento versatile e molto comodo per migliorare le pagine web ma introduce anche una serie di debolezze dovute proprio al fatto che il browser "processa del codice" e di conseguenza se quel codice è "maligno" viene portato ad eseguire operazioni potenzialmente dannose.

Per esempio, se un ipotetico attaccante riuscisse a modificare una pagina web, aggiungendoci per esempio del codice Javascript, **prima** che quest'ultima raggiunga il browser dell'utente finale, esso (l'attaccante) sarebbe così in

grado di far eseguire il proprio pezzo di codice sul browser della vittima che sta tranquillamente navigando.

Questo esempio è chiamato *Man In The Middle Attack* (MITM), ovvero quando un entità (ad esempio un attaccante) si pone in mezzo ad una comunicazione tra due host ed altera le informazioni che vengono trasferite (lo vedremo più nel dettaglio avanti nei capitoli). Questo tipo di attacco può essere sfruttato per una moltitudine di obiettivi. Ciò è reso possibile dal fatto che HTTP intrinsecamente non consenta una autenticazione da parte del server al browser (e viceversa) e non gestisca un controllo sull'integrità di ogni file trasferito. Ovvero non garantisce che il file inviato dal server sia identico a quello che il client riceva e viceversa. Una proprietà che invece HTTPS garantirà come vedremo più avanti.

1.1.3 Web Application (In)Security

Al giorno d'oggi la maggior parte dei siti sono in pratica *web application* che fanno affidamento sul flusso bidirezionale di informazioni scambiate tra server e client (browser). I contenuti presentati all'utente sono il più delle volte generati sul momento, specifici per quell'utente e per ciò che quest'ultimo richiede. Molte informazioni che ci vengono richieste e dunque che circolano in internet sono riservate, ad esempio indirizzi email o di cellulare, dati personali, dati sanitari, dati di carte di credito ecc, e non vogliamo, per ovvi motivi, che vengano mostrate pubblicamente, motivo per cui la sicurezza è un aspetto fondamentale da tenere in considerazione.

Essendo ogni *web application* diversa, ognuna può introdurre differenti debolezze dovute al fatto che siano programmate da sviluppatori, e gli sviluppatori spesso commettono errori. Di conseguenza, con l'incremento costante e sempre più incalzante dei dispositivi mobile quali smartphone e tablet, il range di possibili vulnerabilità, dunque di debolezze sfruttabili da un ipotetico attaccante, si espande a dismisura. Questo perché le debolezze provengono il più delle volte proprio da software con errori di implementazione ed uno in particolare che viene sfruttato per assistere gli attacchi è proprio il browser. Inoltre di quest'ultimo ne esistono tante varianti per tanti dispositivi e Sistemi Operativi (OS) diversi prodotti da altrettanti differenti gruppi di sviluppatori. Il risultato finale è che oltre ad introdurre molteplici metodi per gestire le pagine web, diviene pressoché impossibile poter contare e dunque affidarsi totalmente ad un browser esente da bug.

«La maggioranza delle *web application* è insicura» [21], e quasi certamente ognuna di queste può essere affetta da certe categorie di vulnerabilità come per esempio:

- **Autenticazione debole**: Concerne vari problemi relativi al meccanismo di login che permettono ad un attaccante di "tentare ad indovinare" password deboli, oppure lanciare attacchi *brute-force*, ovvero tentando tutte le possibili combinazioni di password.
- **Controllo degli accessi debole**: Questo coinvolge casi in cui l'applicazione non protegge correttamente l'accesso a dei dati o a delle procedure abilitando potenzialmente l'attaccante a vedere dati sensibili oppure ad eseguire azioni privilegiate.
- **Cross Site Scripting (XSS)**: Questa vulnerabilità insita nell'errata implementazione di codice Javascript integrato nelle web page permette ad un attaccante di interferire con l'applicazione "iniettando" codice malevolo che poi verrà eseguito sul lato client del browser della vittima.
- **Perdita di informazioni** : Questo coinvolge casi in cui l'applicazione lascia trapelare informazioni sensibili che sono sfruttabili da un attaccante, un esempio è quello degli errori ed eccezioni. Se il server o la web application sono mal configurati questo permette all'attaccante di raccogliere informazioni importanti per portare a termine un attacco.
- **Cross-Site Request Forgery (CSRF)**: Questo attacco avviene quando un utente (per esempio loggato) è portato ad inviare richieste maligne (create *ad hoc* da un attaccante) al web server. Qui accade che la richiesta in questione eredita l'identità e i privilegi dell'utente (poichè è lui ad inviare la richiesta), portando cosi l'attaccante a fare eseguire codice arbitrario con i livelli di privilegio di un utente già loggato ad un sito.
- **SQL injection** : Questa vulnerabilità permette ad un attaccante di inviare input perfettamente costruiti alla web application per fare in modo di poter interagire con il database retrostante e dunque eseguire comandi e query arbitrarie.

Nonostante HTTPS possa, se ben configurato, garantire proprietà per una comunicazione sicura tra browser e server, quest'ultimo purtroppo non offre alcuna protezione contro gli attacchi e le debolezze sopra elencati. Questo perchè HTTPS protegge le informazioni durante il loro tragitto e garantisce che non vengano modificate da terze parti, ma non garantisce in alcun modo

sicurezza al browser o alla web application di riferimento se quest'ultima è progettata e programmata male. Questo per mettere in chiaro che sia che un sito utilizzi la crittografia o meno, potrà sempre essere affetto da falle di sicurezza.

1.1.4 Cookie HTTP

I cookie svolgono un ruolo importante per il protocollo HTTP/HTTPS poiché su di essi fanno affidamento molte *web application*. Permettono al server di mandare dati sotto forma di chiave-valore al client il quale li memorizzerà e poi, aspetto importante, **ad ogni successiva richiesta** li ripresenterà al server. Questi dati possono essere le "preferenze" dell'utente oppure il suo stato di login cosicché egli non dovrà, ogni qualvolta ri-accederà al sito, reimmettere le credenziali di accesso. Questo concetto di **sessione** permette di trattare tutte le richieste fatte da un client generico come eseguite da un determinato utente.
Il server dunque potrà impostare dei cookie aggiungendo la direttiva *Set-Cookie* all'header HTTP:

```
Set-Cookie: key1=value1; key2=value2;
```

Il browser client di conseguenza aggiungerà quindi ad ogni richiesta, nell'header HTTP:

```
Cookie: key1=value1; key2=value2;
```

Cosi facendo il server può tenere traccia, memorizzando su database, esattamente di ogni movimento di quello specifico utente, che sia esso loggato o meno. Questo permette ad HTTP di mantenere uno "stato di sessione" riguardante ogni singolo client, cosa che intrinsecamente HTTP non è in grado di fare essendo un semplice protocollo richiesta-risposta in cui ogni coppia di messaggi rappresenta una transazione indipendente. Questo meccanismo è ampiamente sfruttato dalla maggioranza dei siti ma anch'esso non è esente da problemi.

Ricapitolando i cookie sono utilizzati prevalentemente per autorizzare una sessione, come *authentication token* e/o come container di dati.

Oltre ad avere altre opzioni configurabili al momento della creazione, sempre includendole nella riga *Set-Cookie*, possiamo soffermarci su alcune in particolare che ne determinano più o meno un certo grado di sicurezza.

- `domain`: specifica il dominio di validità del cookie. Deve essere lo stesso o il parente del dominio dal quale il cookie è ricevuto.
- `secure`: Se questo attributo è settato, il cookie verrà presentato solamente in richieste HTTPS. Può sembrare a prima vista scontato ma ad oggi non tutti i siti che utilizzano HTTPS utilizzano anche l'attributo *secure* portando così ad esporre importanti informazioni ad un potenziale attaccante.
- `HttpOnly`: Questo attributo, se settato, determina l'impossibilità da parte di script *client-side* di accedere al cookie. Proprietà notevole per la sicurezza poiché essendo i cookie obiettivo ambito, mediante famosi attacchi, primo tra i quali *Cross-Site Scripting* (XSS), un attaccante può accedere ai cookie tramite linguaggio Javascript con un semplice `document.cookie` (riferimento per accedere direttamente al cookie della pagina attuale).

Quindi un ipotetico attaccante può potenzialmente essere in grado di visualizzare ogni cookie di ogni pagina aperta sul browser client? E se le pagine sono sotto HTTPS?

Per questo viene in aiuto SOP *Same-Origin Policy* ([26]) ovvero un critico meccanismo di sicurezza implementato dai browser che permette di isolare potenziali script malevoli.

Questo ci spiega che due pagine hanno "la stessa origine" se e solo se il protocollo, la porta e il dominio di riferimento sono gli stessi. Solo ed unicamente così sarà possibile per uno script interagire con le varie proprietà delle varie pagine "same-origin".

1.1.5 HTTP Connection

I *web server* utilizzano socket TCP e stanno in ascolto su una porta predefinita del sistema: 80 di default e 443 per HTTPS. I web server possono essere configurati per stare in ascolto su porte differenti da quelle di default, di conseguenza per poter raggiungere tali siti sarà necessario specificare il numero di porta nell' URL ad esempio:

http://mywebsite.com:8080/index.html
https://mysecuredwebsite.com:4443/index.html

Utilizzando TCP ad ogni richiesta HTTP corrisponderà una fase iniziale di handshake dopo la quale il client potrà effettuare la prima richiesta e ricevere la prima risposta. Questo ha un costo totale di 2 RTT (Round Trip Time), ovvero 2 "andate e ritorno" dei relativi pacchetti tra client e server.
HTTP permette di gestire due tipi di connessioni:

- *Persistenti*: Mediante l'attributo `Connection` possiamo specificare `keep-alive` per permettere al server ed al client di mantenere attiva la connessione corrente e dunque permettere di scambiare più file sequenzialmente sulla stessa linea.
- *Non-Persistenti*: Mediante attributo `close` indica che ad ogni richiesta il server inizializzerà una nuova connessione la quale verrà terminata non appena il file sarà ricevuto dal client.

Vedremo invece più avanti come funzionano nel dettaglio le connessioni HTTPS, cosa introducono e a cosa devono l'allungamento dei tempi di attesa prima che il client possa inviare la sua prima richiesta.

1.1.6 Problematiche di HTTP

Quale è il problema principale di HTTP? Indubbiamente che ogni byte circola in chiaro, ovvero nulla è crittografato, le informazioni viaggiano così come sono.

Si pensi ai *Cookie* HTTP, moltissimi siti utilizzano proprio questi per mantenere informazioni sugli utenti loggati ad un loro servizio (*Magic Cookie*). Ciò significherebbe che se un attaccante con un semplice strumento di intercettazione del traffico (tcpdump, Wireshark) riuscisse ad entrarne in possesso mediante la cosiddetta tecnica di *eavesdropping*[1], sarà allora in grado di "spacciarsi" per quell'utente legittimo e riuscire ad impersonarsi inviando come cookie, incorporato in una richiesta HTTP, quello appena intercettato. Questo attacco è chiamato *Cookie Stealing* mediante il quale si opera una ***Session Hijacking***.
Per altre informazioni riguardo questi e molti altri attacchi informatici ci si può riferire a *"OWASP, the free and open software security community"* [49]
Sempre mediante intercettazione è ovviamente possibile vedere chiaramente qualsiasi password circoli nella rete che si sta ispezionando. La contromisura si chiama SSL/TLS.

[1] *Eavesdropping* è un termine utilizzato per indicare un intercettazione solitamente non autorizzata di conversazioni private altrui, una forma di eavesdropping si chiama *sniffing*.

D'altronde il protocollo HTTP non è stato strutturalmente progettato includendo gli aspetti relativi alla sicurezza, cosi come lo stesso stack TCP/IP. Quindi è stato fondamentale sviluppare un nuovo set di strumenti all'interno del layer applicativo per provvedere al necessario aumento di privacy e sicurezza delle comunicazioni. I lavori furono iniziati già nel 1995 con SSL versione 2.0 fino ad arrivare alla versione 1.2 di TLS, e tuttora è in lavorazione la prossima versione, la 1.3.

1.2 HTTPS

Correttamente *"HTTP Over SSL/TLS o HTTP Secure"* ([66]) consiste nella comunicazione tramite protocollo HTTP all'interno di una connessione crittografata mediante TLS o dal suo predecessore, deprecato, SSL. HTTPS è essenzialmente lo stesso protocollo a livello applicativo di HTTP ma passa per un tunnel crittografato. SSL e TLS sono protocolli crittografici progettati per garantire una comunicazione sicura sopra una infrastruttura insicura. Ciò significa che se questi protocolli sono correttamente implementati, chiunque può aprire un canale di comunicazione verso un arbitrario servizio su internet ed essere ragionevolmente certo che:

- Sta parlando con il server corretto
- Le informazioni in transito non possono essere lette da terze persone
- Le informazioni scambiate arriveranno perfettamente intatte

Ovvero vengono garantite le proprietà fondamentali di una comunicazione sicura quali autenticazione, riservatezza ed integrità. Il tutto mediante vari algoritmi applicati in maniera complementare e, teoricamente, rigorosa. Il problema che a volte determina il fallimento di tali algoritmi, anche quelli considerati sicuri, risiede proprio nel fatto che vengano indotti a funzionare non "allo stato dell'arte" e la mancanza di opportuni controlli ne conferma la loro violabilità.

Dunque HTTPS è in grado di fornirci un elevato/totale grado di segretezza?
Se la risposta è no, ciò significa che tutte le connessioni sono a rischio?
Cosa comporta questo al giorno d'oggi?

1.2.1 Funzionamento generale e cenni di crittografia

Partiamo da una spiegazione ad alto livello di come funziona HTTPS.

Per capire dove si collocano SSL (*Secure Socket Layer*) e TLS (*Transport Layer Security*) dobbiamo riferirci al modello *Open System Interconnection* OSI [66] dove tutte le funzionalità di rete sono mappate in 7 strati. Al livello più alto, quello più vicino all'utente, si trova lo strato Applicativo e subito sotto troviamo lo strato di Presentazione. Questo ha come compito quello di gestire la rappresentazione, la loro conversione ed eventualmente tutti gli aspetti legati alla crittografia dei dati provenienti dai layer inferiori/superiori, ed è proprio qui che stanno SSL e TLS. Questo è un aspetto importante poiché permette alle applicazioni di "utilizzare" semplicemente tutte le loro funzionalità senza doversi preoccupare di come queste siano poi implementate, un esempio perfetto di *separation of concerns*.
SSL e TLS stanno quindi sopra TCP (*Transport layer*) ma sotto HTTP (*Application layer*).

Il loro funzionamento si fonda sull'utilizzo della crittografia, simmetrica e asimmetrica, ed è in parte demandato ad un altro meccanismo chiave chiamato PKI (*Public Key Infrastructure*) il quale gestisce gli aspetti relativi ai **certificati**.

La crittografia *simmetrica* si basa sul principio di una **chiave unica** in grado di crittografare un blocco di informazioni mediante un opportuno algoritmo ad un'elevata velocità. Cosi facendo tutto ciò che è contenuto nel blocco diventa illeggibile e solamente con la medesima chiave e riapplicando lo stesso algoritmo si potrà tornare ad avere le informazioni originali. Il grado di sicurezza dipende dalla lunghezza della chiave utilizzata ma lo vedremo meglio più avanti. Poiché questa crittografia si basa sull'utilizzo di una chiave unica e condivisa tra due parti, come è possibile sfruttarla tra milioni di host distanti centinaia di km? Il problema detto della distribuzione delle chiavi, ovvero della crittografia asimmetrica che nessuno fino a quel momento aveva neanche ideato, è stato immaginato e risolto per la prima volta nel 1976 da due gruppi distinti, *Whitfield Diffie* e *Martin Hellman* in un gruppo e 3 anni prima *James H. Ellis* nell'altro. .

La crittografia *asimmetrica* comprende una **coppia** di chiavi matematicamente correlate, una tenuta segreta e chiamata *private key* e l'altra resa invece pubblica, *public key*. Le due chiavi permettono di criptare e decriptare ma

con la differenza che un messaggio criptato con una delle due verrà decrittato solamente dall'altra e viceversa. Tutto si basa sul fatto che la chiave privata debba essere unica e ben protetta dal suo possessore, dalla sua non compromissione dipende l'intero sistema.

Ciò rende possibile, siccome la crittografia a chiave pubblica utilizza algoritmi computazionalmente onerosi, di combinare entrambi; il client utilizza la chiave pubblica del server per crittografare un piccolo pezzo di informazione per poi inviarlo al server stesso. A questo punto il server ed il client utilizzano quella informazione come chiave per la crittografia simmetrica (molto più rapida) per poter crittografare tutti i byte che si scambieranno da quel momento in avanti.
Questo esempio si riferisce a come lavora l'algoritmo ***RSA*** di cui parleremo approfonditamente più avanti. Questa fase in cui client e server si scambiano questa "chiave" da utilizzare per la crittografia simmetrica si chiama "***Key-Exchange***". Si noti che in altri algoritmi la chiave simmetrica viene generata e/o derivata in maniere diverse.

Ma come è possibile per un client essere certo che sta parlando con il server con il quale intende dialogare e che questo non sia un impostore? Per questo si utilizzano i certificati.

I certificati sono idealmente "garanzie" dell'identità digitale di un client, dominio o server racchiuse sotto forma di documento in formato *X.509* e dunque con una specifica formattazione. Questi vengono firmati da un autorità competente detta *Certification Authority* che ne garantisce l'integrità e vengono diffusi pubblicamente dalla PKI. Contengono informazioni circa il dominio ed il suo proprietario ma soprattuto contengono la chiave pubblica del server che verrà utilizzata da chiunque voglia instaurare una connessione SSL/TLS con lui.

Vedremo come il meccanismo dei certificati possa funzionare perfettamente in teoria ma in verità sul campo abbia diverse debolezze. Vedremo come vengono instaurate le connessioni HTTPS e quali sono i suoi punti critici e come tutto questo insieme di elementi possa cadere vittima di attacchi di ogni tipo che vanno a mirare i punti più deboli e meno controllati di tutta la catena.

1.3 Importanza delle comunicazioni oggi

Web server e *web application* sono l'obiettivo preferito da malintenzionati poiché al giorno d'oggi custodiscono interi patrimoni sotto forma di dati bancari, identità dei cittadini, informazioni governative, informazioni militari, dati di business online, e-mail ed ogni altra forma di comunicazione e banca dati. In parole brevi, la nostra vita ed il nostro denaro.

Per capire quanto sia fondamentale HTTPS e più in genere SSL/TLS:
cosa succederebbe se le informazioni di carattere militare dei vari stati viaggiassero insicure su internet? Chiunque potrebbe entrarne in possesso, portando così tecnologie top secret alla mercé di tutti. Cosa accadrebbe invece a transazioni e tutte le miliardi di altre operazioni bancarie che vengono tutti i giorni eseguite da web browser? Potrebbero essere modificate "in transito" da un attaccante con conseguenti danni e perdita di fiducia per le banche che cadrebbero all'istante con conseguente crollo dell'economia. Così come, scendendo più nel piccolo, qualunque sito di e-commerce non potrebbe garantire alcuna sicurezza riguardo ai pagamenti essendo le informazioni a rischio intercettazione.

Ci sono due lati della medaglia, come ad intendere una sfida fra due fazioni: chi cerca di sviluppare una crittografia sicura al 100% e chi cerca di "romperla" per dimostrare che quest'ultima non esiste. Questa lotta continua da tantissimi anni ed ha portato a innovazioni incredibili oltre a influenzare in maniera determinante molte guerre accadute in passato.

Questo ruolo che oggi riveste la crittografia non ha fatto che diventare nel tempo sempre più imponente e necessario soprattutto poichè strumenti per catturare il traffico di rete sono di libero reperimento. Software per effettuare attacchi sono spesso totalmente automatizzati e quindi alla portata dei più quindi pur avendo tecnologie all'avanguardia bisogna sempre essere vigili e seguire diligentemente tutte le pratiche di sicurezza e di buona configurazione.

Inoltre, ad oggi, il costo effettivo per poter configurare un sito web con HTTPS è accessibile a tutti e compagnie come Google ne incoraggiano incessantemente l'utilizzo. [43]"*Let's Encrypt*" una nuova Certification Authority offre certificati gratuiti con un procedimento totalmente automatizzato. Per di più i *web site* che utilizzano SSL/TLS possono da tempo godere di alcuni vantaggi come un miglior *ranking* e dunque di conseguenza un migliore

posizionamento sui motori di ricerca. [8]

Tentando quindi di rispondere alle domande sopra poste:
HTTPS è in grado di fornirci un **elevato** grado di sicurezza **solo se** tutte le precauzioni quali aggiornamenti software, configurazioni server e browser correttamente impostate, suite crittografiche ben scelte e costanti controlli su validità certificati ecc siano messe in atto. La mancanza di conoscenza e quindi di attenzione dell'utente finale può inconsapevolmente portarlo ad essere vittima di attacchi alla sicurezza esponendo così tutte le sue informazioni ad un ipotetico attaccante.

Capitolo 2

Public Key Infrastructure

2.1 Introduzione

Per poter ben comprendere come attacchi alla sicurezza di siti web e più in generale attacchi a SSL/TLS possano avere luogo abbiamo prima bisogno di spiegare *step by step* come l'intero sistema funzioni, almeno a grandi linee.

La crittografia a chiave pubblica, ovvero quella asimmetrica come la abbiamo introdotta poco fa, è ciò che ci permette di comunicare in maniera sicura con persone o web server, i quali hanno reso disponibile la loro chiave pubblica.

Per fare qualche esempio ci riferiremo a tre ipotetiche persone: Alice (A) e Bob (B) sono i due principali che hanno la conversazione e Eve (E) invece è un ipotetico attaccante il quale cerca di interferire con la loro comunicazione.

Vediamo con un primo esempio semplificato come l'intero processo ci permetta di iniziare una comunicazione crittografata:

Sono stati tralasciati volutamente certi passaggi poichè verranno spiegati nel dettaglio nei prossimi capitoli, ora l'importante è capire il funzionamento dei certificati e della crittografia a chiave pubblica in generale.

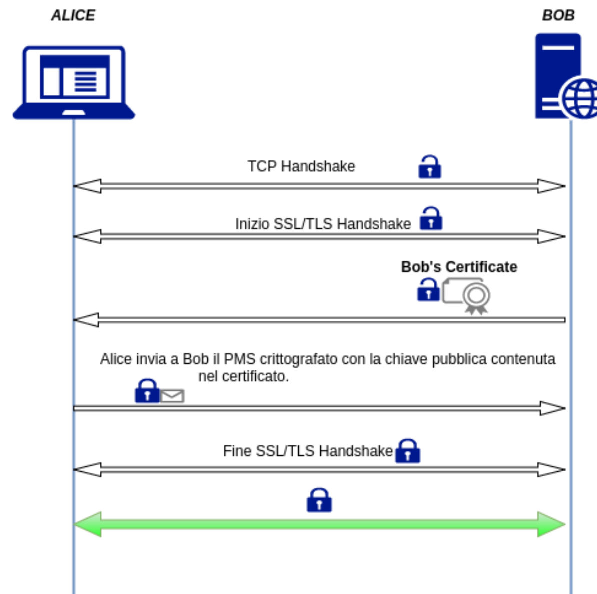


FIGURA 2.1: Esempio di inizializzazione semplificata di una comunicazione sicura in HTTPS

Partiamo spiegando ad alto livello cosa significano i passaggi visti nella figura:

1. Alice e Bob per parlarsi in maniera sicura devono innanzitutto inizializzare una connessione TCP fra di loro.
2. A questo punto entrano in gioco i protocolli di SSL/TLS. Qui dopo una fase iniziale, Bob invia ad Alice il suo certificato contenente la sua chiave pubblica.
3. Alice controlla il certificato e se tutto è andato a buon fine è sicura che l'entità con cui sta parlando sia veramente Bob.
4. Ora Alice utilizza la chiave pubblica contenuta nel certificato per inviare a Bob un *Pre Master Secret* PMS, ovvero un segreto (una stringa casuale) che poi verrà utilizzato per derivare la *chiave di sessione* utilizzata come chiave per la crittografia simmetrica.
5. Termina il SSL/TLS Handshake ora che entrambe le parti hanno ottenuto in maniera sicura una chiave da utilizzare per crittografare simmetricamente ogni dato che si invieranno da quel momento in avanti.

In figura il lucchetto aperto indica che la comunicazione è crittografata, chiuso che è in chiaro.

Essendo questi argomenti pienamente intrecciati tra loro ed essendo impossibile raggrupparli tutti insieme creando così confusione all'utente che

legge, potreste trovare dei termini che non spiegherò nel dettaglio, questo perché verranno tutti ampiamente discussi nei capitoli successivi.

Dunque come abbiamo appena visto dall'esempio chiunque può instaurare una connessione crittografata con un server utilizzando la sua chiave pubblica, ma come ci perviene quest'ultima? Chi ci garantisce che è effettivamente quella, e non quella di un impostore che si stia "spacciando" per il web server? Come facciamo a sapere se quella chiave pubblica sia ancora valida? E tutto ciò, come facciamo ad applicarlo a milioni di server e a miliardi di persone e dispositivi collegati ad Internet?

L'obiettivo principale della PKI è appunto questo, ed anche quello di abilitare una comunicazione protetta tra due parti che non si sono mai "incontrate". Per fare ciò, a fare da Garante tra le due parti si è aggiunto un meccanismo chiave, quello della ***Certification Authority (CA)*** che crea e firma i certificati sui quali ogni volta poniamo la nostra fiducia.

Il termine PKI viene usato per indicare sia l'autorità di certificazione (Certification Authority) ed i relativi accordi, sia, in senso più esteso, l'uso di algoritmi crittografici a chiave pubblica nelle comunicazioni elettroniche. [66]

Le CA sono affiancate anche da ***Registration Authority*** e ***Certificate Server***.

2.1.1 PKI Vista d'insieme

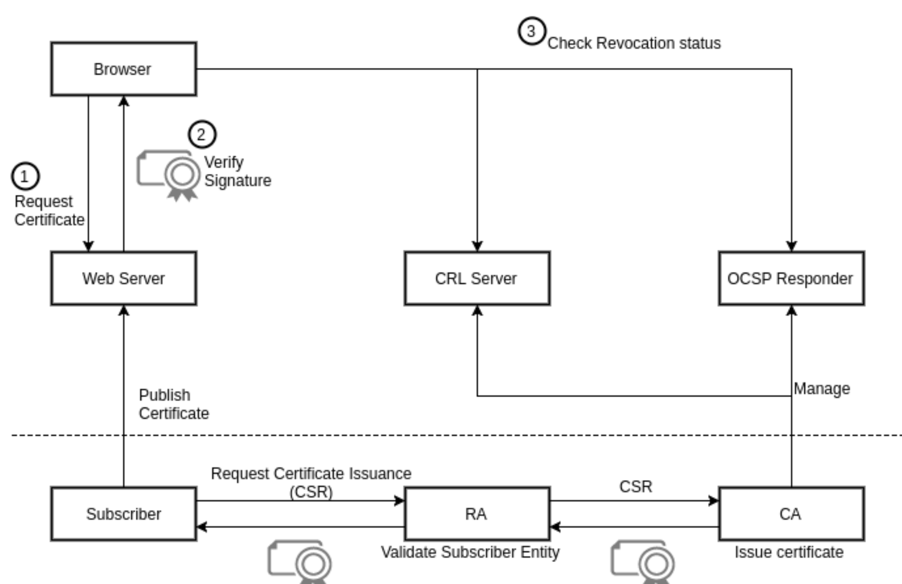


FIGURA 2.2: Internet PKI certificate lifecycle [1]

Questa figura racchiude gli aspetti principali del funzionamento della PKI ed entro la fine del capitolo ogni argomento verrà dettagliatamente spiegato.

Intanto possiamo vedere che la figura racchiude due meccanismi separati concettualmente dalla linea tratteggiata. Nella parte superiore notiamo ciò che coinvolge principalmente un browser nell'utilizzo dei certificati e della loro validazione, mentre nella parte inferiore possiamo vedere il processo di creazione del certificato da parte dell'entità richiedente e la Certification Authority (ovviamente questo passaggio è il primo che viene effettuato).

Gli elementi che compongono la PKI sono:

- **Subscriber**: detto "*end-entity*", è chi richiede il certificato che può essere ad esempio il proprietario di un dominio.
- **Registration Authority (RA)**: ha il dovere di validare l'identità proposta dal subscriber al momento della richiesta di un certificato prima che la richiesta passi alla CA. Svolge dunque funzioni delegate dalla CA. È detta anche Local RA.
- **Certification Authority**: è la terza parte su cui ognuno pone fiducia, ha il compito di "firmare" i certificati garantendo quindi con essi l'identità del Subscriber.
- **Browser**: è la parte finale che utilizza i certificati, può essere come da esempio il browser, ma anche qualsiasi altro applicativo ne faccia uso.

CRL e OCSP Responder li vedremo fra poco.

2.2 Certificati

I certificati costituiscono gli elementi base della PKI ma cosa sono e cosa contengono?

Innanzitutto si basano su uno standard internazionale, lo X.500 il quale fu adattato all'uso sulla Internet PKI dal *PKIX Working Group* nel 1995 e poi riferito come **X.509**, ad oggi versione n.3. [50] La versione X.509 v1 fu inizialmente pubblicata nel 1988 come parte della X.500. Questo determina la struttura di un certificato digitale ed anche le implementazioni delle importanti CRL, ovvero *Certificate Revocation List*[66].

2.2.1 Struttura

La struttura di un certificato X.509 v3 contiene i seguenti campi:

- **Versione**: attualmente si utilizza la n.3
- **Numero Seriale**: inizialmente era un numero positivo intero che identificava unicamente il certificato creato dalla CA ma poi, per aggiungere un ulteriore strato di sicurezza a causa di attacchi che riuscivano a predirne il valore, è diventato un numero non sequenziale che contiene almeno 20 bit di *entropia*.[1]
- **Signature Algorithm**: specifica l'algoritmo utilizzato per la firma del certificato.
- **Ente Emettitore**: contiene il *DN* (Distinguished Name) ovvero un "nome distintivo" di chi rilascia il certificato costituito da tre parti: nazione, organizzazione ed unità organizzativa.
- **Validità**: il periodo di validità di tale certificato specificato con una data di inizio ed una data di fine.
- **Soggetto**: è il nome distintivo dell'entità associata con la chiave pubblica assegnata al certificato stesso. Il *CN* (Common Name) indica l'*hostname* ovvero l'indirizzo web associato. Ex: */CN=www.example.com*. Oggi siccome questo può portare a confusione, quando un certificato è valido per diversi hostname, si utilizza invece l'estensione *SubjectAlternativeName* con la quale si possono indicare in lista diversi hostname. [33]
- **Chiave pubblica**: il campo contiene la chiave pubblica rappresentata dalla struttura *Subject Public-Key Info* che essenzialmente è formata da un ID dell'algoritmo, parametri opzionali e la chiave pubblica stessa.
- **Estensioni**: qui è presente una lista di estensioni che aggiungono flessibilità ai certificati, ognuna consiste di un *object identifier* (OID), un indicatore di criticità ed un valore. Per esempio se una estensione è marcata come *critical* questa deve essere processata correttamente altrimenti l'intero certificato viene respinto.

[1] *Entropia*: si intende come una misura della resistenza della password (quanto è impredicibile). Il numero di bits indicati indicano quanti tentativi sono necessari per esaurire tutte le possibilità durante un attacco "brute-force", ad esempio 40 bit di entropia indicano che sono necessari 2^{40} tentativi. [65]

- **Algoritmo di firma del certificato**: indica l'algoritmo utilizzato per firmare il certificato ad esempio "*PKCS 1 SHA-256 With RSA Encryption*"
- **Firma del certificato**: rappresenta l'intero certificato "firmato" con l'algoritmo sopra identificato.

2.2.2 Catene di Certificati

Nella maggioranza dei casi un certificato da solo non è sufficiente per una validazione corretta, quindi è necessario fornire una *catena* di certificati che ci riporta infine ad una **root CA** reputabile come fidata. Questo significa che il certificato finale viene "firmato" diverse volte, ogni volta da un autorità ad un livello più alto fino a concludere con la root Certification Authority. Ogni certificato utilizzato per la firma che non sia quello del proprietario finale, ne quello della root CA è chiamato certificato intermedio. Questa diventa una catena poichè ogni relativa chiave privata di ogni certificato a partire dalla root CA è utilizzata per firmare il certificato al livello inferiore, fino ad arrivare al certificato finale emesso per l'utente finale.

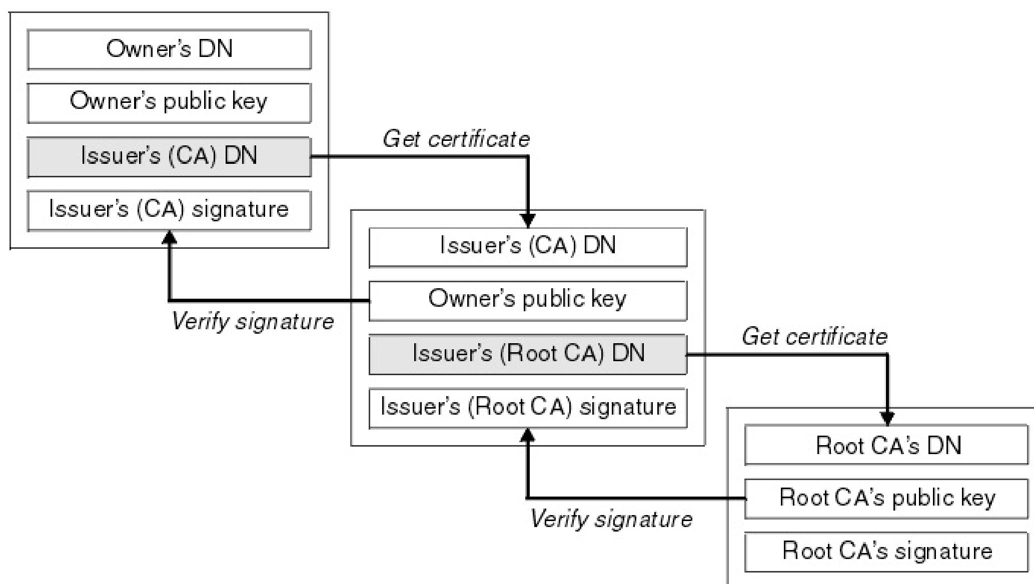


FIGURA 2.3: The below figure illustrates a certification path from the certificate owner to the Root CA, where the chain of trust begins[61]

Quando un browser vuole dialogare con un sito protetto dovrà dunque richiedere tutti e tre i tipi di certificati, e mediante vari meccanismi verificare che questi ultimi siano validi. Solitamente ogni sistema operativo così come

ogni browser contiene già una lista delle più importanti root CA pronte per essere usate per la validazione di certificati.

2.2.3 Ciclo di vita di un certificato

Il ciclo inizia quando un *subscriber*, ovvero un utilizzatore finale, prepara un *Certificate Signing Request (CSR)* e lo presenta alla CA di sua scelta. Il CSR contiene informazioni come la chiave pubblica per la quale il certificato debba essere rilasciato, informazioni identificative come il nome di dominio ed informazioni utili per la protezione dell'integrità come una firma digitale.

Successivamente la CA procederà alla validazione in base al tipo di certificato richiesto che può essere di tre tipologie ognuna delle quali garantisce, in ordine, maggiore fiducia:

- **Domain Validation (DV)**: certificati di questo tipo sono rilasciati dopo aver provato di possedere il controllo effettivo di un dominio. Ad esempio mediante una mail di conferma inviata all'email del dominio stesso. Solitamente questo tipo di procedimento è totalmente automatizzato.
- **Organization Validation (OV)**: richiedono di verificare identità ed autenticità. Solitamente utilizzati da organizzazioni, governi o altre entità.
- **Extended Validation (EV)**: forniscono il più alto livello di fiducia ai visitatori e necessitano di più controlli da parte della CA. I certificati EV permettono al browser di mostrare direttamente sulla barra degli indirizzi il nome della organizzazione in verde.

Dopo aver validato, la CA potrà rilasciare il certificato. In aggiunta, la CA provvederà a fornire anche tutti i certificati intermedi necessari per completare la catena fino alla radice. Solitamente la catena consta di 3 certificati in totale. A questo punto l'utente finale potrà utilizzare il certificato in produzione fino al momento in cui esso scadrà. Altri motivi per il quale il certificato non sarà più sicuro e/o valido sono la compromissione della corrispondente chiave privata e la revoca del certificato.

2.2.4 Revoca dei Certificati

I certificati vengono **revocati** quando le relative chiavi private vengono compromesse oppure quando essi non sono più necessari.

Ci sono due standard per la revoca (ed il controllo di validità) dei certificati:

- **Certificate Revocation List (CRL)**: è una lista contenente tutti i numeri seriali appartenenti ai certificati revocati che non sono ancora scaduti. Ogni CA mantiene una o più liste ed in ogni certificato deve essere indicato il percorso per raggiungere la CRL nell'apposito *CRL Distribution Points* campo. Il problema principale con le CRL è che queste ultime tendono a diventare di dimensioni considerevoli allungando così i tempi per ogni eventuale consultazione.

- **Online Certificate Status Protocol (OCSP)**: è un sistema client-server che permette di ottenere in tempo reale lo stato di revoca di un certificato in maniera più semplice, veloce ed affidabile rispetto allo scaricamento delle CRL. I server OCSP sono detti *OCSP responders* ed il loro indirizzo è codificato nell'estensione *Authority Information Access* del certificato. OCSP ha però anche degli svantaggi poichè essendo ogni richiesta analizzata dal responder, si passa ad esso tutta la cronologia di navigazione e questo è un evidente problema di ***privacy***. Inoltre se entro un timeout dalla richiesta il server non risponde, OCSP verrà ignorato silenziosamente, ed essendo un attaccante in grado di bloccarne la consegna questo è un altro evidente problema di sicurezza.

Un approccio alternativo a OCSP è l'*OCSP stapling* formalmente conosciuto come *TLS Certificate Status Request extension*[66]. Questo permette al server di incorporare una risposta OCSP direttamente dentro al TLS handshake. Vedremo nel prossimo capitolo il suo funzionamento.

2.3 Importanza della root CA

La chiave privata della root Certification Authority è quanto di più prezioso esista per la CA stessa e l'intero ecosistema. Oltre ad avere un immenso valore economico, se la chiave viene compromessa può essere utilizzata per generare qualunque certificato fraudolento per qualunque nome di dominio. In tal caso la chiave dovrebbe essere revocata causando così l'inutilizzabilità di tutti i siti che dipendono da essa. Questi sono motivi per cui per esempio rilasciare un certificato firmato direttamente dalla root CA non è più permesso (*Basement Requirements for the Issuance and Management of Publicly-Trusted Certificates*), inoltre il server dove risiede la CA deve sottostare a rigidissimi

controlli e regole di sicurezza una delle quali l'isolamento quasi totale dalla rete.[45]

2.3.1 Debolezze della PKI

Se osserviamo l'intera PKI possiamo notare che ci sono diverse mancanze dal punto di vista della sicurezza, alcune meno significative ed alcune notevoli. Dobbiamo sempre ricordare che l'Internet non è nato sicuro *per design* ma quest'ultima è stata aggiunta e poi sempre rifinita successivamente. La sicurezza sul web è un compromesso tra Browser, Certification Authority ed esperienza di navigazione per l'utente finale. Le CA hanno sicuramente come scopo quello di aumentare i profitti commerciali, i browser quello di incrementare i loro *market share*, il tutto tentando di mantenere un adeguato livello di sicurezza per l'utente finale. È proprio riguardo a questo livello che si viene ad un compromesso: se si volesse mantenere lo standard più elevato allora molti client che non lo supporterebbero verrebbero tagliati fuori per motivi di compatibilità, all'opposto uno standard eccessivamente permissivo porterebbe solamente a peggiori problemi di sicurezza per tutti. Questo è il motivo per cui spesso è impossibile sfruttare la crittografia più potente e più recente su grande scala, quindi se ne utilizza una a livello inferiore che permetta di allargare il bacino di utilizzatori. Per poter usufruire appieno di un sistema veramente sicuro sarebbe necessario che ogni singolo componente dell'intero ecosistema funzioni a dovere cooperando con tutti gli altri, adoperando sempre l'ultima tecnologia disponibile pena l'inaccessibilità del sito web, dunque lo scontento e la perdita del pubblico.

Vediamo alcuni punti critici:

- **Permesso del proprietario di dominio non richiesto per il rilascio di certificati**: è senza dubbio che la CA debba verificare l'identità del richiedente certificato prima di rilasciarlo, ma è vero anche il contrario? Questo è un problema concettuale, qualsiasi CA può creare un certificato per qualsiasi dominio senza bisogno dell'autorizzazione del proprietario del dominio stesso. Questo poteva non essere un problema quando le root CA erano poche, ma ad oggi che ce ne sono decine e centinaia la sicurezza dell'intera PKI dipende appunto "dall'anello più debole della catena". Che danni si avrebbero se un attaccante riuscisse a prendere controllo di questo meccanismo? Sono accaduti eventi in passato come ad esempio il caso della CA **Trustwave (2012)** la quale

ha ammesso pubblicamente di aver permesso ad una azienda di creare certificati *on the fly* con l'obiettivo di poter quindi ispezionare il traffico di rete protetto diretto a qualsiasi sito web.[5] Possiamo a questo punto essere completamente sicuri che le Certification Authority non agiscano anche sotto direttive dei governi o di enormi e potenti aziende visto che, come terze parti, affidiamo a loro tutta la nostra fiducia e ne saremmo comunque ignari?

- **Debole validazione di dominio (DV certificates)**: questi certificati sono rilasciati sulla prova di proprietà di un dominio, il più delle volte mediante e-mail che potrebbero essere insicure. Tutta la sicurezza di quel dominio si potrebbe ridurre per un attaccante a trovare la password della mailbox appartenente a quel dominio.

- **Il controllo sulla revoca dei certificati non funziona affatto**: ci sono diversi motivi per la quale questo accade. Uno è il ritardo di propagazione del comando di revoca in ogni server della CA, come da baseline, ogni informazione in CRL e OCSP deve rimanere valida per almeno 10 giorni. Ciò significa che c'è bisogno di almeno 10 giorni per propagare totalmente l'informazione. Un altro problema è cosi detto della *soft-fail* policy implementata nei browser; ovvero quando essi tentano di ottenere informazioni riguardo la revoca o meno di un certificato, se questa informazione non giunge in un tempo limite o non giunge proprio allora i browser semplicemente la **ignorano** e proseguono accettando il certificato così com'è. Addirittura ***Google Chrome*** non prova nemmeno a controllare lo stato di revoca dei certificati (tranne che per certificati EV). Un attaccante può mediante *man in the middle* sopprimere le richieste e dunque far accettare alla vittima un certificato falso.[6]Per controllare certificati più importanti Chrome utilizza i **CRLSets** che sono liste CRL aggiornate periodicamente dal browser.[19]

- **Gli avvisi su certificati da parte del browser fanno fallire lo scopo della crittografia**: uno dei più grandi fallimenti della PKI è questo lasso approccio alla verifica della validità dei certificati. Quando il browser riesce a verificare che un certificato è invalido presenta un warning all'utente al quale viene chiesto se vuole bypassarlo o meno con un semplice click. Un click e lo scopo della crittografia, e della sicurezza svanisce. Recentemente uno standard chiamato **HTTP Strict Transport Security (HSTS)** è stato sviluppato per istruire i browser a troncare la connessione e dunque a mostrare un errore invece che un warning.[66]

A causa delle dimensioni tutt'altro che modeste di questo immenso ecosistema quale è la PKI, ricercatori ed esperti continuamente monitorano l'andamento dei certificati e degli algoritmi crittografici utilizzati evidenziandone la loro importanza e tutte le vulnerabilità alle quali sono soggetti.

Già nel 2010 ad una famosa serie di conferenze sulla sicurezza informatica, *Black Hat USA*, *Ivan Ristić* pubblicò una ricerca su milioni nomi di dominio con un analisi dei certificati e dei TLS server [54]. Poco dopo la *Electronic Frontier Foundation (EFF)* annunciò un progetto chiamato *SSL Observatory*, ovvero una ricerca basata sull'intero IPv4 [25]. In Aprile 2012, *SSL Labs* iniziò un progetto chiamato "SSL Pulse" che effettua uno scan mensile dei 150,000 siti web più popolari ottenuti da Alexa top 1 million [51].

Vediamo un veloce confronto tra due scan effettuati da SSL Labs a Febbraio 2014 e ad Ottobre 2017. [52]

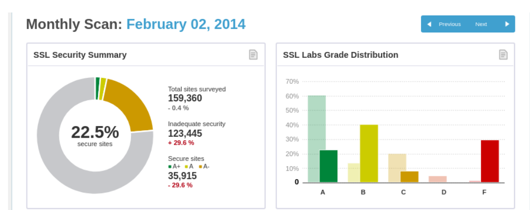


FIGURA 2.4: ssl-pulse survey on Feb 2014

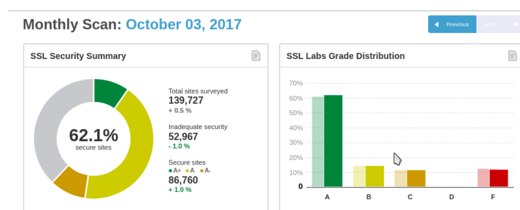


FIGURA 2.5: ssl-pulse survey on Oct 2017

Possiamo notare che negli ultimi tre anni la percentuale di siti che utilizzano una corretta configurazione per garantire un adeguato livello di sicurezza, in media, è quasi triplicato rispetto al 2014. Ciò fa presumere un costante ma lento adeguamento agli standard minimi di sicurezza disponibili. Possiamo anche altrettanto criticamente trasalire pensando "ancora ad oggi almeno il 40% dei siti Internet considerati 'sicuri' è mal configurato e probabilmente vulnerabile?"

2.4 PKI possibili miglioramenti

Durante gli anni ci sono state molte proposte per migliorare lo stato della PKI, soprattutto a partire dal 2011 dopo diverse gravi compromissioni di alcune root Certification Authority. Vediamo alcune delle più importanti e che hanno avuto maggiore successo:

- **Public Key Pinning**: indirizza una delle grandi falle della PKI, ovvero il fatto che qualsiasi CA potesse creare un certificato per qualsivoglia indirizzo web senza l'autorizzazione del proprietario di quest'ultimo. Infatti, con il *pinning*, il gestore del sito web può selezionare una o più CA come fidate, isolando quindi tutte le altre.[24] Meccanismo molto potente quanto "laborioso e pericoloso" da mantenere in quanto, una volta abilitato, lega un'identità crittografica (chiave pubblica/certificato) all'indirizzo web per un periodo di tempo. Se per qualche grave motivo si perdesse il controllo di tale identità, si perderebbe anche il sito web.[14] Simile ma da non confondere con il consueto uso dei certificati, se in questo caso si dovessero compromettere le chiavi, basterebbe revocarlo e richiederne uno nuovo alla CA.

- **DANE**: *DNSSEC* è una serie di specifiche che estende *DNS* con il controllo di integrità. Non aggiunge crittografia dei dati, ma garantisce la validità dell'indirizzo web che l'utente sta visitando. Garantisce che l'associazione tra hostname ed indirizzo IP sia quella originale, troncando così molte opportunità per possibili attaccanti. DANE è un ponte tra DNSSEC e il protocollo SSL/TLS definito nell RFC6698 [40]. Un dominio può essere associato direttamente alla sua corrispondente chiave pubblica, incorporando il tutto nel record DNS. È molto interessante come questo meccanismo, se globalmente implementato, può permettere di evitare la necessità delle Certification Authority. [1]

- **Certificate Transparency**: è un framework per la revisione ed il monitoraggio dei certificati pubblici. Ogni volta che una CA rilascia un nuovo certificato deve altrettanto inviarne una copia "firmata" ad un *log* pubblico dei certificati. Così facendo chiunque può monitorare ogni nuovo certificato emesso e quindi ogni certificato fraudolento potrebbe essere identificato velocemente. Da Febbraio 2015 Chrome richiede la Certificate Transparency per certificati EV. [1] In Ottobre 2016 Chrome ha annunciato che entro l'Aprile 2018 richiederà la Certificate Transparency per **tutti** i nuovi certificati rilasciati.[59]

2.5 Attacchi alla/alle Certification Authority

Ci sono diversi attack vector[2] che possono essere sfruttati per poter farsi rilasciare un certificato senza essere l'autentico proprietario del dominio, il più comune è il processo di validazione. Se convinci una CA di essere il legittimo proprietario allora questa ti rilascerà il certificato.

Vediamo alcuni esempi di incidenti ed attacchi a CA accaduti.

Thawte login.live.com (2008)

Estate 2008, Mike Zusman ricercatore di sicurezza informatica è riuscito ad ingannare il meccanismo di validazione dei certificati di *Thawte* per ottenere un certificato valido per *login.live.com* il quale era ed è un singolo punto di autenticazione Microsoft per milioni di utenti. Due fatti lo resero possibile: Thawte permetteva diverse email per la verifica del domain name, e Microsoft permette a chiunque di registrare un dominio *@live.com*. Una delle email che Thawte accettò per l'autenticazione fu *sslcertificates@live.com*, la quale era anche disponibile alla registrazione come account ordinario Microsoft. Non appena Mike registrò la mail, potè ricevere il certificato. [1]

StartCom Breach (2008)

Nel Dicembre 2008 Mike Zusman bypassò totalmente il meccanismo di domain validation sfruttando una debolezza nel sito web dell'azienda stessa. Ciò gli permise di essere in grado di richiedere ed ottenere certificati per qualsiasi nome di dominio. Solamente per una lista di "high-profile websites" un doppio controllo era innescato, motivo per cui certificati per *paypal.com* e *verisign.com* furono subito bloccati e revocati. [1] Nell'Ottobre 2016 Apple, Chrome e Mozilla hanno deciso ufficialmente di non fidarsi più di qualsiasi nuovo certificato rilasciato da StartCom.[59]

[2] Attack vector: un mezzo, una tecnologia per la quale un accesso non autorizzato può essere ottenuto per un device o una rete da un hacker o un malintenzionato. Esempi sono: allegati email, chat rooms, virus, finestre pop-up, social network, ecc.[60]

CertStar (Comodo) Mozilla Certificate (2008)

Due giorni dopo l'attacco a StartCom, il loro CTO Eddy Nigg scoprì un problema simile con un altra CA, CertStar, un partner affiliato di Comodo la quale rilasciava certificati senza nessuna validazione di dominio. Eddy ottenne certificati validi per la stessa *startcom.org* e per *mozilla.org*. La notizia del certificato fraudolento valido per Mozilla, quale dominio high-profile, finì sui media. [1] [17]

RapidSSL Rogue CA Certificate

Nel 2008 un gruppo di ricercatori guidato da Alex Sotirov e Mark Stevens portarono alla luce un grande proof-of-concept[3] di un attacco contro l'intera PKI, ottennero un certificato falso di una Certification Authority che poteva essere usato per firmare certificati per qualsiasi altro website. Il procedimento è lungo e complesso ma riassumendolo non andando nel dettaglio, si sfruttò la debolezza dell'algoritmo di hashing MD5 per creare dei *colliding certificate*, ovvero due certificati con un hash completamente uguale, ma contenenti informazioni diverse. Questo è uno dei casi eccezionali quando una funzione di hash restituisce due valori uguali per due input totalmente differenti, chiamata appunto **collisione**. Successivamente RapidSSL migrò nel giro di pochissime ore a SHA1, un migliore algoritmo di hashing ma questa scoperta mise in allerta tutte le CA dimostrando a pieni voti l'insicurezza dell'algoritmo MD5. [1] [68]

DigiNotar

Diginotar, una CA olandese, fu la prima CA ad essere completamente compromessa, con certificati fraudolenti utilizzati realmente. I root certificate furono poi revocati con conseguente bancarotta dell'azienda in Settembre 2011. Un totale di 531 certificati furono creati tra i quali moltissimi high-profile website come CA, agenzie governative e grandi aziende del calibro di Google, Facebook, Twitter, ecc. [1] [53]

[3] POC: "prova del concetto", incompleta realizzazione di un progetto con lo scopo di dimostrarne la fattibilità.

Capitolo 3

Cenni di crittografia

La crittografia è l'arte e la scienza della comunicazione sicura. Nonostante si possa comunemente associare ai tempi moderni questa è in realtà utilizzata da migliaia di anni.[1]

Quando la crittografia è correttamente applicata soddisfa i tre requisiti principali della sicurezza: mantenere segreto (*confidentiality*), verificare le identità (*authenticity*) e garantire integrità nel trasporto (*integrity*).

Al livello più basso la crittografia si basa su quelle che vengono chiamate *primitive crittografiche*. Queste funzioni vengono combinate in *schemi e protocolli*.

3.1 Crittografia Simmetrica

Per comunicare in maniera sicura Alice e Bob si accordano su un algoritmo crittografico e poi su una chiave segreta. Successivamente quando uno dei due dovrà scrivere all'altro non farà altro che crittografare il messaggio con la chiave segreta ed inviarlo. Una volta ricevuto il messaggio *cifrato* utilizzando la stessa chiave segreta sarà possibile *decifrarlo* per ottenere il messaggio originale. Eve, cercando di spiare la conversazione, anche se fosse in grado di poter accedere al canale e quindi a tutti i messaggi scambiati, anche conoscendo l'algoritmo, non sarebbe comunque in grado di comprenderli poiché crittografati.

Un algoritmo è considerato buono quando produce del testo che sembra totalmente casuale, il quale non può essere analizzato da un attaccante che con tecniche di *analisi delle frequenze* possa riuscire ad identificare le lettere e quindi a ripristinare il messaggio originale. Se un algoritmo è "sicuro" l'unico

modo per "romperlo" è tentare, per esempio nel caso di una chiave, tutte le possibili combinazioni di chiavi fino ad ottenere quella corretta, quello che si chiama *exhaustive key search*, quello che gli attacchi *brute-force* fanno.

Dunque la sicurezza del messaggio dipende esclusivamente dalla chiave. Se questa è presa da un grande *keyspace*[1] allora "rompere la crittografia" significherebbe letteralmente iterare su un numero proibitivo di combinazioni ed allora si dice che il *cipher*[2] è *computazionalmente sicuro*.

I cipher o *cifrari* si dividono in due gruppi.

3.1.1 Stream Cipher

Questi chiper funzionano come se esistesse un canale unico di bytes in chiaro i quali uno per uno vengono dati in pasto ad un algoritmo crittografico dal quale ne escono cifrati.

Un algoritmo stream cipher produce uno *stream* infinito di dati random chiamato *keystream* a partire da una chiave segreta. Per cifrare, ogni byte del keystream è combinato con un byte del testo in chiaro (*plaintext*) usando un operazione logica chiamata XOR[3]. Siccome lo XOR è una operazione reversibile, per decifrare si effettua nuovamente lo XOR del testo cifrato con lo stesso byte del keystream.

L'intero processo di cifratura è considerato sicuro se un attaccante non è in grado di predire quali possano essere gli elementi del keystream ed in quale posizione essi siano. Per questo motivo è vitale che gli stream ciphers non vengano utilizzati più di una volta con la stessa chiave. Gli attaccanti conoscendo sia testo in chiaro sia testo cifrato possono facilmente confrontarli per risalire al keystream e decifrare successivamente ogni messaggio. Per questo motivo è comune che gli stream ciphers utilizzino chiavi chiamate *one-time* che vengono derivate da chiavi *long-term*.[1]

Un esempio di un algoritmo tra i più utilizzati in passato è *RC4* ma da tempo considerato insicuro ora il suo utilizzo è veramente limitato.

[1]keyspace: insieme di tutte le possibili permutazioni di una chiave. Esempio, se avessimo una chiave lunga 1 byte allora il keyspace è grande 2^8 ovvero 256 possibilità

[2]cipher: è l'algoritmo utilizzato per cifrare e decifrare

[3]XOR: "disgiunzione esclusiva", è un connettivo (o operatore) logico ($\oplus$) che restituisce in uscita VERO (V) se e solo se gli ingressi sono diversi tra di loro, (F) FALSO altrimenti.[66]

3.1.2 Block Cipher

Gli algoritmi di questo tipo cifrano interi blocchi di dati alla volta. Questo è anche un motivo limitante, ovvero la lunghezza dei dati da cifrare deve essere multipla della dimensione del blocco, motivo per cui ci sono degli schemi crittografici utilizzati per poter cifrare dati di arbitraria lunghezza. I block cipher o "algoritmi di cifratura a blocchi", operano come funzioni trasformatrici: prendono un input e producono un output "randomizzato". Il problema è che sono *deterministici* ovvero producono sempre lo stesso output per lo stesso input. In pratica però questi vengono utilizzati con degli schemi crittografici particolari chiamati *block cipher modes* i quali ne eliminano le limitazioni. L'algoritmo a blocchi più popolare al mondo è l'**AES** (*Advanced Encryption Standard*) disponibile a 128, 192 e 256 bits. [66]

Come gestire quindi i casi in cui i dati da cifrare eccedano la grandezza di un blocco? ***Padding***. Il padding consiste nell'aggiungere alla fine del testo in chiaro un po di dati extra in un modo che il ricevente possa poi riconoscerli e scartarli. Tutti i byte di padding sono settati con lo stesso valore della lunghezza del padding stesso.

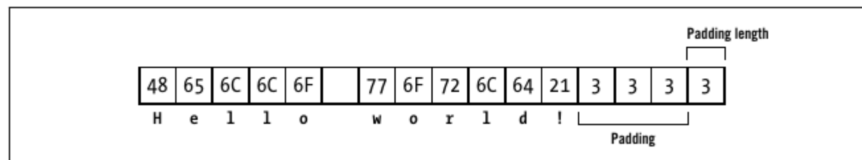


FIGURA 3.1: Esempio di padding

3.1.3 Block Cipher Modes

Lo scopo di queste modalità di funzionamento dei cifrari a blocchi è sempre quello di garantire confidenzialità mentre in alcuni casi è fornita anche autenticazione.

Vediamo alcuni esempi.

Electronic Codebook Mode (ECB)

ECB è uno dei più semplici cifrari a blocchi, funziona solamente trasformando blocchi di 64 bits o multipli in input in altrettanti blocchi in output. Se la dimensione non è esattamente tale allora il padding verrà aggiunto *prima* di

cifrare il blocco. L'algoritmo paga la sua semplicità in termini di sicurezza in quanto produce sempre lo stesso output per lo stesso input. Ciò significa che un attaccante potrebbe sottoporre dell'input arbitrario ad un applicativo web, esempio delle richieste HTTP, e osservare le risposte cifrate. Da qui può notare quando del testo è ripetuto ed infine ricostruire il messaggio in chiaro.[48]

Cipher Block Chaining Mode (CBC)

Qui si cercano di eliminare i difetti di ECB con un algoritmo molto utilizzato che introduce il concetto di *initialization vector* (IV) il quale permette di produrre un output differente anche se l'input rimane lo stesso. L' IV è un vettore generato inizialmente contenente dati random che ha la stessa dimensione del blocco. Si parte effettuando uno XOR del primo blocco in chiaro con l'IV e successivamente crittografando il blocco appena ottenuto. Da questo momento in avanti ad ogni blocco di testo in chiaro successivo verrà effettuato uno XOR con il blocco cifrato precedente (utilizzato quindi come IV) subito prima di essere crittografato.

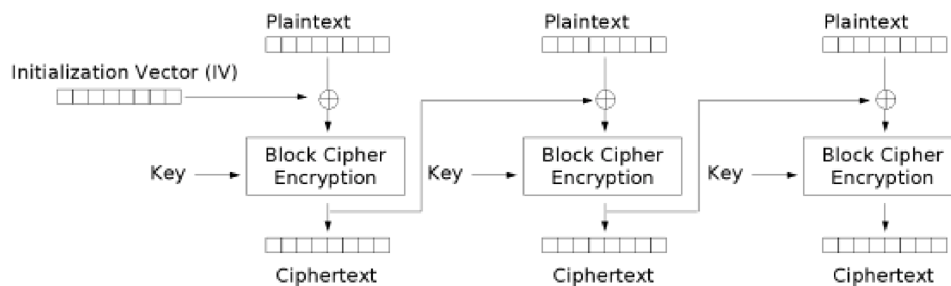


FIGURA 3.2: Cipher Block Chaining (CBC) mode encryption [66]

Purtroppo parlando di sicurezza effettiva, CBC così com'è non è più adatto ad essere utilizzato con HTTPS e dunque TLS. A causa delle sue vulnerabilità, una delle quali la gestione del padding. Diversi attacchi furono scoperti come ***POODLE***, ***BEAST***, ***Padding Oracle Attack*** e ***Lucky Thirteen attack*** che vedremo nei prossimi capitoli.

Come possiamo notare gli algoritmi crittografici basano la loro sicurezza su dei valori generati "casualmente" ed è fondamentale che questi siano impredicibili. Inoltre è **fondamentale** per evitare *tampering*, ovvero manomissioni, che l'intero messaggio sia **autenticato**. Altrettanto importante è

non riutilizzare mai la stessa chiave e lo stesso IV. Come è possibile questo? Utilizzando dei generatori di numeri casuali come vedremo fra poco.

Esistono altre modalità di funzionamento dei cifrari a blocchi con altrettanti pro e contro e queste, per esempio, operano trasformando *block cipher* in *stream cipher*:

- Cipher Feedback (CFB)
- Output Feedback (OFB)
- Counter (CTR)

Galois Counter Mode (GCM)

Questa modalità nota per la sua efficienza e le sue performance [44] ci permette una "*crittografia autenticata*[4]" in quanto fornisce allo stesso tempo confidenzialità, autenticazione ed integrità. GCM opera con blocchi di 128 bits. Ne rivedremo più avanti le potenzialità.

Quali sono quindi le principali differenze tra *block* e *stream cipher*? Principalmente gli stream cipher, oltre a non garantire l'autenticità, permettono di inviare in maniera immediata e veloce i byte senza dover aspettare di riempire ogni blocco, con la sconvenienza che la lunghezza dei dati cifrati inviati rimarrà la stessa dei dati in chiaro, una potenziale informazione per un attaccante. Cosa che non accade con i block cipher con il padding. Esistono però algoritmi, come sopra abbiamo accennato GCM, che nascono come block cipher e possono essere trasformati in stream cipher unendo così le potenzialità di entrambi.

3.2 Hash Functions

Una funzione di *hashing* è un algoritmo deterministico che converte un input di lunghezza arbitraria in un output di dimensione fissa, detto semplicemente *hash*. Non tutte le funzioni hash sono utilizzabili per la crittografia, quelle che lo sono devono soddisfare delle proprietà addizionali:

- **Preimage resistance**: ovvero dato un hash è computazionalmente irrealizzabile costruire un messaggio che lo produca.

[4] *Authenticated Encryption with Associated Data (AEAD* [66]

- **Second preimage resistance**: ovvero dato un hash e il messaggio originale è computazionalmente irrealizzabile trovare un secondo messaggio che produca lo stesso hash.
- **Collision resistance**: è computazionalmente irrealizzabile trovare due messaggi che producano lo stesso hash.

Piuttosto che confrontare due file per intero bit per bit è molto più comunemente usato confrontare i loro valori di hash. Lo stesso vale per la verifica e storage delle password. Per questo le funzioni di hash vengono spesso utilizzate per *fingerprint o message digest*.[1]

Ad oggi molte sono le funzioni hash conosciute, tra cui le più discusse ed utilizzate:

- **MD5**: realizzata nel 1991 da *Ronald Rivest* uno dei tre membri della RSA [66]. Produce un output di lunghezza fissa di 128 bit ed è stato ampiamente utilizzato, anche tutt'ora che è considerato completamente **insicuro**. Insicuro perchè si è trovato il metodo efficiente per generare stringhe che *collidono* ovvero che producono lo stesso hash, con conseguenze veramente pericolose per i sistemi (CA e certificati per esempio) sui quali fanno affidamento.
- **SHA-1**: da *Secure Hash Algorithm* [66] è un algoritmo nettamente migliorato rispetto a MD5 che produce un *digest* o output di 160 bits, ed è uno dei più diffusi algoritmi dell'intera famiglia SHA. Ad oggi anche questo è considerato insicuro in quanto in Febbraio 2017 Google ha annunciato di essere riuscita a completare la prima *collisione*. [56] Ciò ne prova la sua violabilità e dunque suggerisce un cambio verso un altro algoritmo più sicuro, come ad esempio SHA-256.
- **SHA-256** fa parte della suite SHA-2 così come le versioni più avanzate *SHA-384, SHA-512*. SHA-256 come da nome produce un digest di 256 bits ed è attualmente la migliore e più utilizzata scelta dopo SHA-1. [66]

Dunque prendendo SHA-256 come esempio, il numero totale di output che può generare è 2^{256} ovvero *1.1579209e+77* possibilità.

Un paradosso da tenere in considerazione parlando di sicurezza di una funzione hash è il ***Birthday Attack***. Questo è un fenomeno statistico che permette di fare brute-force su funzioni di hash in maniera più semplice, esattamente riducendo del **50%** i tentativi necessari. Perchè questo? Cerchiamo di capirlo in maniera semplice.

Il nome *birthday* è riferito proprio all'esempio preso in considerazione per dimostrare questo fenomeno. Facendo finta di avere una stanza da riempire di persone, quante devono essere queste per avere una chance del 50% che almeno due condividano lo stesso compleanno? **23**. Del 100%? **57**.
Questo perchè partendo da 0 ogni persona aggiunta, dovendo evitare quelle già presenti, deve evitare anche l'insieme di *probabilità* possibili. La probabilità del 50% viene fuori sommando in sequenza tutte le probabilità di ogni persona aggiunta, ed in questo caso, dopo 23 somme il risultato è quasi esattamente del 50%. [42]

Dunque rivalutando la effettiva efficacia delle funzioni hash, se una è a 128 bit, dunque 2^{128} possibilità, in realtà si avrà una possibilità maggiore del 50% di trovare una collisione già dal 2^{127} tentativo.

3.3 Message Authentication Codes (MAC)

Prendiamo ad esempio un messaggio ed il suo valore di hash che ne "riassume" univocamente il contenuto. Se Alice volesse inviare entrambi a Bob, Bob comunque non sarebbe in grado di fidarsi in quanto un attaccante potrebbe aver modificato il messaggio e poi aver rigenerato il corrispettivo valore hash. Per questo motivo c'è bisogno di estendere l'hashing con una autenticazione da parte del mittente in modo che solo chi possieda la chiave possa produrre quello che è chiamato MAC. Il valore MAC quindi protegge l'integrità del messaggio ed allo stesso tempo ne garantisce l'autenticazione. La funzione prende in ingresso la chiave segreta ed il messaggio e restituisce un MAC, il ricevente per verificarlo non farà altro che ricalcolarlo sul messaggio originale e vedere se coincide. Questo permette di tagliare fuori possibili attaccanti in quanto se non si è a conoscenza della chiave è impossibile ricostruire un MAC valido per il messaggio. Qualsiasi funzione hash può essere usata come base per MAC utilizzando una modalità detta HMAC (*keyed-hash message authentication code*[66]). Essenzialmente HMAC funziona combinando la chiave ed il messaggio in maniera sicura.[1]

3.4 Random Number Generation

Come stiamo vedendo la crittografia si basa su lunghe chiavi che altro non sono che numeri generati casualmente. Il problema è che a volte è possibile, ma non semplice, predire tali valori. Veri numeri casuali possono essere ottenuti osservando e collezionando porzioni di *entropia*, come ad esempio movimenti del mouse, orari, interazione con periferiche o device o hard disk, ecc. Questo metodo è chiamato *true random number generator (TRNG)* ma non è affidabile abbastanza per essere usato direttamente poiché spesso questa "entropia collezionata" non è sufficiente a garantire la sicurezza richiesta. Per questo motivo generalmente si utilizzano invece *pseudo-random number generator (PRNG)* i quali utilizzano piccole parti di dati random chiamate *seed* come input ad un algoritmo che è in grado poi di generare lunghe sequenze di dati apparentemente casuali. Anche qui un problema è che se si riesegue l'algoritmo con lo stesso valore di seed si avrà in output la stessa identica sequenza di numeri "pseudocasuali". Questo è il motivo per cui non si utilizzano direttamente in crittografia ma se ne utilizza un altro tipo chiamato *cryptographic pseudo-random number generators (CPRNG)* il quale rende la sequenza anche impredicibile.[66]

3.5 Firma digitale

La firma digitale è uno schema crittografico che rende possibile dimostrare l'autenticità di un documento o un messaggio. MAC è un esempio di firma digitale. In generale il meccanismo si basa sull'uso della crittografia asimmetrica, un documento (o il suo hash) firmato con una chiave privata è decifrabile da chiunque possieda la sua chiave pubblica. Questo significa che se Alice firma il suo messaggio con la sua chiave privata allora chiunque abbia la sua chiave pubblica potrà verificare che quel messaggio sia veramente proveniente da Alice.

Facciamo un esempio di come questo funzioni:

1. Si calcoli l'hash del documento che alla fine avrà dimensione fissa, ad esempio 256 bits per SHA256.
2. Si aggiungano le informazioni necessarie al ricevente per ricalcolare l'hash come appunto l'algoritmo di hashing utilizzato.

3. Si crittografi il tutto utilizzando la propria chiave privata, il risultato è la **firma digitale** che funge pienamente da prova di autenticità da aggiungere separatamente al messaggio.

Per verificare la firma il destinatario potrà e dovrà utilizzare esclusivamente la chiave pubblica del mittente per decifrare la firma, a questo punto utilizzerà lo stesso algoritmo di hashing indicato per *hashare* il documento e se il risultato è identico a quello contenuto nella firma allora il messaggio è integro ed autentico.

Non tutti gli algoritmi di firma digitale funzionano allo stesso modo questo è un esempio riguardante *RSA*[5] che permette sia di cifrare sia di firmare contemporaneamente.

3.6 Conclusione

Ci sono diversi modi per attaccare la crittografia. Possiamo partire considerando le primitive crittografiche, potremmo provare a fare *brute-force* su una chiave se questa è corta ma se non lo è il processo richiederebbe un tempo infinito oltre ad una eccezionale potenza di elaborazione. Inoltre spesso gli algoritmi essendo pubblici sono visibili a chiunque dal momento in cui sono stati pubblicati, motivo per cui se sono considerati sicuri da molte persone, molto probabilmente lo sono.

Ci sono d'altra parte attacchi alla *implementazione* di questi ultimi, in parole povere *bug* nei programmi che li utilizzano. Il più delle volte sono utilizzati linguaggi di basso livello per migliorare al massimo le performance, ma questi richiedono anche delle buone capacità di programmazione, il che aumenta la probabilità di fare errori.

Per questi motivi il più delle volte si dice che la crittografia è aggirata e non attaccata poiché le primitive sono veramente solide e resistenti, ma non vale lo stesso per l'ecosistema che le utilizza.

[5] RSA: algoritmo di crittografia asimmetrica, inventato nel 1977 da Ronald Rivest, Adi Shamir e Leonard Adleman utilizzabile per cifrare o firmare informazioni.[66]

Capitolo 4

SSL e TLS

In questo capitolo descriveremo principalmente il protocollo TLS versione 1.2 per poi farne un confronto con le versioni precedenti e quella successiva ancora in fase di sviluppo.

Uno dei modi migliori per capire il funzionamento del protocollo è quello di osservare come si comporti *real-time*, quindi con programmi di cattura ed analisi del traffico come *Wireshark* potremo velocemente vederne degli esempi.

TLS è formato principalmente da 2 livelli di protocolli. Il **Record protocol** sta alla base e fornisce funzionalità essenziali a TLS stesso per la gestione dei pacchetti. Sopra il Record protocol avremo distintamente l'**Handshake protocol** il quale si occupa si tutto ciò che concerne l'instaurazione del canale protetto, e poi successivamente l'**Application Data protocol** con il quale si opera il trasferimento continuo di dati. *Alert protocol* e *Change Cipher Spec protocol* sono altri due protocolli che operano sopra il Record protocol.

4.1 Record Protocol

Il Record protocol funge, subito sopra TCP, da mezzo di trasporto a più basso livello per i messaggi scambiati all'interno di una connessione protetta.

Ogni *TLS Record* inizia con un corto *header* il quale conterrà informazioni relative al sotto-protocollo (*type* e *version*) e la lunghezza in byte (*length*).

Vediamo quali sono questi campi contenuti nell'header (RFC5246) [35].

```
struct {
        uint8 major;
        uint8 minor;
```

```
      } ProtocolVersion;

enum {
          change_cipher_spec(20),
          alert(21),
          handshake(22),
          application_data(23), (255)
      } ContentType;

struct {
          ContentType type;
          ProtocolVersion version;
          uint16 length;
          opaque fragment[TLSPlaintext.length];
      } TLSPlaintext;
```

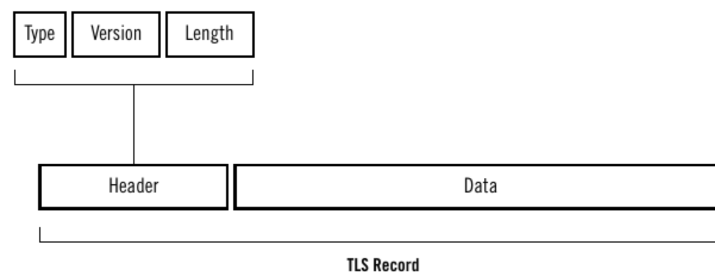


FIGURA 4.1: TLS Record [1]

Come possiamo vedere abbiamo 1 byte per descrivere il *content type* ed indicare quale sotto-protocollo è utilizzato, 2 byte per la versione ed i rimanenti per i dati o *TLSPlaintext* la quale lunghezza non deve assolutamente superare i 2^{14} byte o *octet*.

Il Record protocol è utile per diversi importanti aspetti della comunicazione:

- **Trasporto messaggio**: se il messaggio supera la lunghezza limite di 16,384 byte il protocollo si occuperà di frammentarlo in pezzi più piccoli e successivamente si occuperà anche di ri-assemblare i frammenti per comporre il messaggio originale.
- **Crittografia ed integrità**: all'inizio della comunicazione il protocollo invia messaggi privi di protezione. Alla fine della fase di negoziazione si occuperà della crittografia e della validazione dell'integrità di ogni messaggio scambiato.

- **Compressione**: del messaggio se necessaria. In pratica non è stata più utilizzata per due motivi: primo è già abilitata a livello applicativo (Ex. HTTP) e secondo a causa di vari attacchi scoperti che ne sfruttavano le debolezze.
- **Estendibilità**: questo protocollo si occupa di crittografare e validare l'integrità ma delega le altre funzioni ai sotto-protocolli rendendo cosi TLS molto estendibile e flessibile.

TLS identifica 4 sotto-protocolli: *handshake protocol, change cipher spec protocol, application data protocol e alert protocol*.[1]

4.2 Handshake Protocol

Questa è la parte più elaborata di TLS che ha come compito quella di negoziare dei parametri di connessione e di effettuare l'autenticazione di uno o entrambe le parti della comunicazione. Solitamente questa fase richiede dai 6 ai 10 messaggi per completarsi, a seconda delle opzioni e dei casi.

Possiamo distinguere 3 casi principali:

full handshake with server authentication

abbreviated handshake which resumes session

handshake with client and server authentication

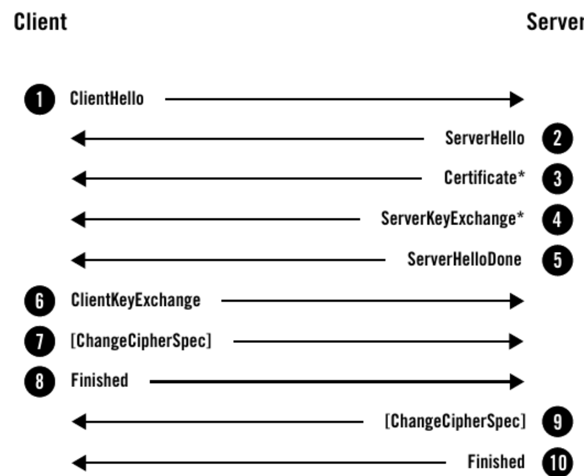
4.2.1 Full Handshake

Questa è la fase che ogni client deve seguire se non ha già stabilito in precedenza una sessione con il server, e si chiama appunto *full handshake*.

Qui il compito principale è quello di preparare entrambe le parti ad una connessione sicura accordandosi sugli algoritmi crittografici, le chiavi ed i protocolli da utilizzare. Questa prima fase è chiamata anche di *key exchange* o *key establishment*.

1. Il client inizia un nuovo handshake inviando il *Client Hello message*
2. Il server seleziona alcuni parametri di connessione e risponde con *Server Hello*.
3. Il server invia il certificato (e la sua catena).

FIGURA 4.2: TLS Handshake with server authentication [1]



4. A seconda del *key exchange* selezionato il server invia altro materiale necessario alla generazione delle chiavi.

5. Il server indica che ha finito la sua parte iniziale della negoziazione.

6. Il client invia altre informazioni necessarie alla generazione delle chiavi.

7. Il client informa il server che passerà all'utilizzo della crittografia.

8. Il client avvisa che ha terminato la sua parte.

9. Il server informa il client che passerà all'utilizzo della crittografia

10. Il server avvisa che l'handshake è terminato ed invia un MAC di tutti i messaggi ricevuti ed inviati.

Alla fine dell'handshake sia client che server, controllando il MAC di tutti i messaggi che si sono scambiati hanno il compito di verificare che la comunicazione non è stata alterata in alcun modo da una terza parte e quindi si presuppone possa continuare in maniera sicura.

Da questo momento client e server hanno generato la stessa chiave simmetrica e sono in grado di comunicare crittografando ogni messaggio inviato.

Vediamo ora il significato di ogni messaggio nel particolare.

Client Hello

Questo è sempre il primo messaggio all'interno dell'handshake ed è utilizzato dal client per comunicare al server l'insieme di algoritmi crittografici chiamati *cipher suites* che può supportare e le estensioni TLS utilizzabili.

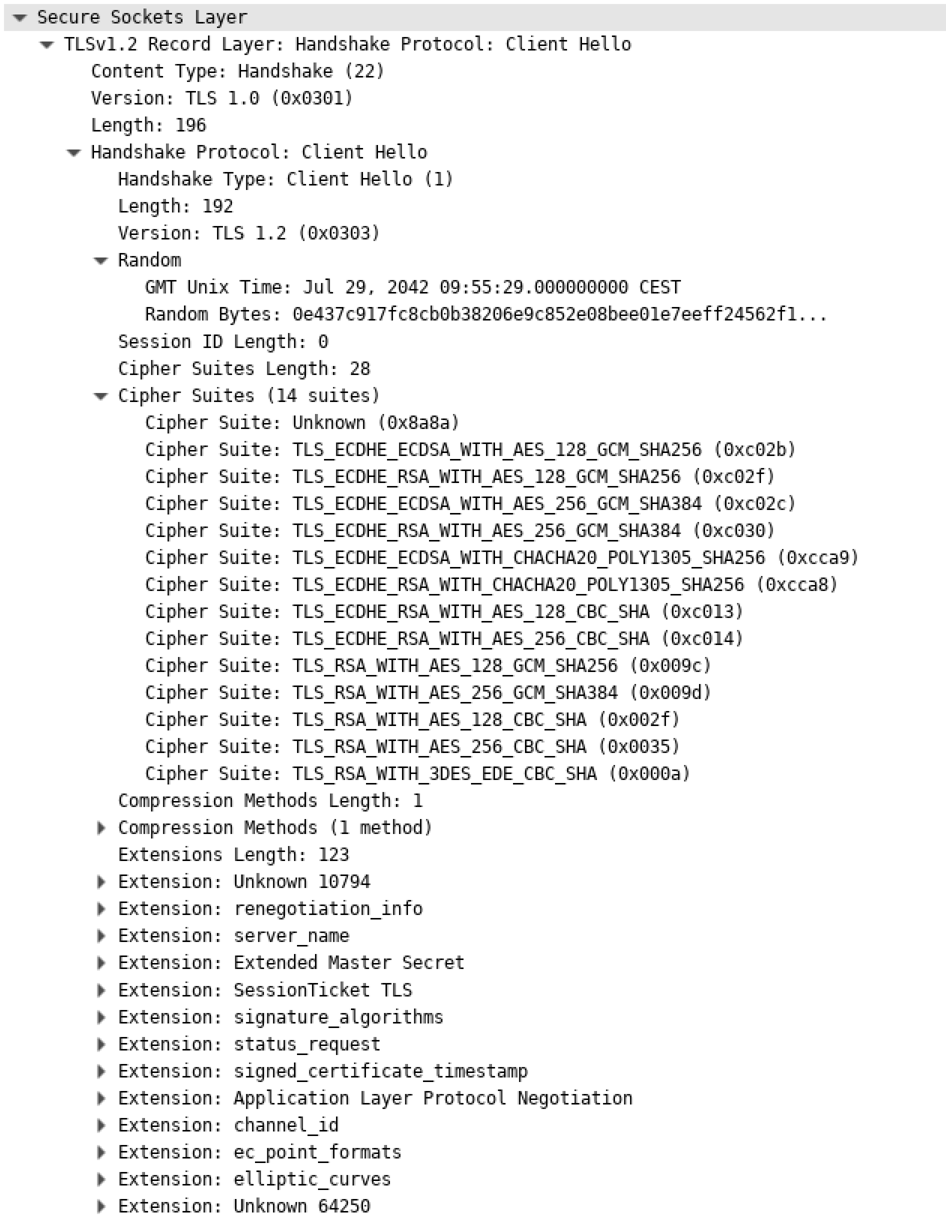


FIGURA 4.3: TLS Client Hello example

Come possiamo vedere per forza di cose i messaggi iniziali vengono scambiati in chiaro e dunque potenzialmente visibili a chiunque. La potenza della fase di key exchange è proprio questa. Far arrivare sia server che client ad

una stessa chiave, parlandosi e scambiandosi in chiaro le informazioni per ricavarla.

La struttura del messaggio è alquanto semplice da analizzare. Vediamo subito che l'intero messaggio è costruito sopra il *TLS Record Layer*, sul quale è identificato il sotto-protocollo Handshake Protocol con codice (22), lunghezza del pacchetto 196 byte e versione TLS 1.0. Ora inizia il Client Hello:

Version: identifica la massima versione di SSL/TLS supportata dal client

Random: contiene 32 byte di dati generati randomicamente e necessari ad aggiungere un fattore di unicità. Questi giocano un ruolo importante in quanto aiutano a prevenire *replay attack*, ovvero attacchi che, ripresentando gli stessi dati, cercano di simulare l'identità del mittente originario. [66]

Session ID: quando un client si sta connettendo per la prima volta è vuoto, quando invece vuole riprendere una sessione già iniziata contiene l'identificatore unico di quella sessione per permettere al server di reimpostare i parametri di quella comunicazione.

Cipher Suites: contiene la lista di tutte le suite crittografiche supportate dal client.

Compression: indica il metodo di compressione utilizzato, ormai di prassi settato a null, ovvero non viene utilizzata alcuna compressione.

Extensions: contiene un numero arbitrario di estensioni TLS che aggiungono ulteriori dati.

Server Hello

Lo scopo di questo messaggio è quello di comunicare al Client gli algoritmi crittografici scelti dal server per la comunicazione, a differenza del Client Hello qui ogni campo contiene un unico valore.

```
▼ Secure Sockets Layer
  ▼ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 84
    ▼ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 80
        Version: TLS 1.2 (0x0303)
      ▼ Random
          GMT Unix Time: Mar 14, 2096 22:48:21.000000000 CET
          Random Bytes: ca47218eba59e84760806acceb2bbb96748a969730ecc8ff...
        Session ID Length: 0
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
        Compression Method: null (0)
        Extensions Length: 40
      ▶ Extension: server_name
      ▶ Extension: renegotiation_info
      ▶ Extension: ec_point_formats
      ▶ Extension: SessionTicket TLS
      ▶ Extension: status_request
      ▶ Extension: Application Layer Protocol Negotiation
```

FIGURA 4.4: TLS Server Hello example

Come possiamo notare il server ha accettato la richiesta del client di connettersi mediante TLS 1.2 ed ha accettato una delle cipher suites selezionate. Importante sapere che il server può accettare tutti i protocolli che lui stesso implementa e rende disponibili. Se un server ha ancora disponibile il protocollo SSLv3 ed un client chiederà di connettersi mediante lo stesso, allora la connessione verrà instaurata normalmente. Lo stesso vale per le cipher suites, il server selezionerà con il suo ordine di priorità quella che è disponibile in entrambe le liste. Motivo per cui è **fondamentale** per il server rimuovere tutte le cipher suites contenenti algoritmi insicuri così come negare la possibilità di connettersi a client che non utilizzino almeno una certa versione di TLS.

Certificate

Questo messaggio altro non fa che trasportare la catena di certificati del server. È importante che il server invii il certificato appropriato alla suite crittografica scelta in quanto alcuni algoritmi si basano su delle informazioni codificate nel certificato stesso.

Nonostante la maggior parte delle connessioni lo utilizzi, questo messaggio è opzionale poiché non tutte le suite utilizzano autenticazione ed inoltre ci sono alcuni metodi che non necessitano di certificati.[1]

In caso OCSP Stapling venga utilizzato allora il server invierà anche un altro tipo di messaggio chiamato *Certificate Status* il quale conterrà direttamente una copia della risposta ricevuta dall'OCSP responder contenente l'indicazione di validità o meno del certificato.

Server Key Exchange

Lo scopo di questo messaggio è di trasportare dati addizionali indispensabili al client per la *derivazione* delle chiavi. Un esempio sono i parametri per lo scambio di chiavi *Diffie-Hellman*.

Server Hello Done

È il segnale che il server ha inviato tutti i messaggi iniziali dell'handshake ed è ora in attesa di risposta dal client.

Client Key Exchange

Anche qui il client ha il compito di inviare al server parametri addizionali necessari alla derivazione delle chiavi e strettamente dipendenti dalla cipher suite scelta.

Change Cipher Spec

Questo messaggio, facente parte un protocollo a sé, indica che il client ha ora tutte le informazioni necessarie per generare la chiave di cifratura e che da questo momento in avanti crittograferà ogni messaggio. Sia Server che Client si inviano tale messaggio.

Finished Message

È il messaggio che indica il completamento dell'handshake ed è il primo ad essere crittografato con la chiave simmetrica di sessione. Inoltre contiene nel campo `verify_data` un hash di tutti i messaggi precedentemente scambiati dalle due parti e la appena creata chiave di cifratura [35].

4.2.2 Client authentication

Qui abbiamo visto come funziona l'handshake TLS con autenticazione da parte del server. È altrettanto possibile richiedere anche l'autenticazione da parte del client, in tal caso il server effettuerà una **certificate request** (con possibilità di definire le CA ammissibili) ed egli risponderà con il messaggio **client certificate**. Inoltre il client dovrà dare ulteriore esplicita prova di possedere il certificato inviando nel messaggio **certificate verify** un hash firmato con la propria chiave privata di tutti i messaggi di handshake precedentemente scambiati con il server.[35]

4.2.3 Session Resumption

Session resumption o *abbreviated handshake* permette di ristabilire un canale tra server e client che si erano già "parlati" risparmiando quindi messaggi e tempi di calcolo. Questo è permesso poiché entrambi memorizzano i parametri dell'ultima connessione effettutata per un certo periodo di tempo durante il quale sia server che client mantengono degli identificatori di sessione chiamati *session ID*. Non appena un client voglia riprendere una comunicazione esistente, altro non farà che incorporare nel Client Hello l'appropriato Session ID.

A questo punto se il server accetta di riprendere la sessione, entrambi rigenereranno un nuovo set di chiavi partendo dal master secret precedentemente utilizzato.

Così facendo i messaggi scambiati si riducono a 6.

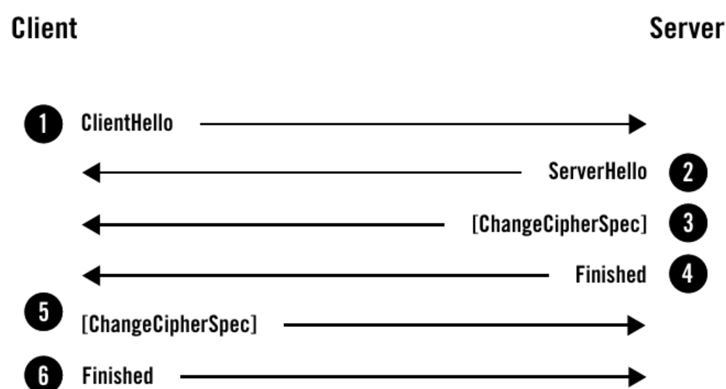


FIGURA 4.5: TLS Abbreviated Handshake [1]

Altrimenti una estensione di TLS chiamata *session tickets* (RFC 5077) permette al server di evitare di mantenere in memoria lo stato di ogni connessione, riversando il compito sul solo client. Il server ingloberà lo stato di sessione (master secret e cipher suite) in un "ticket", crittograficamente protetto, e lo invierà al client che lo memorizzerà (come un cookie HTTP). Successivamente quando il client vorrà riprendere la sessione specificherà il ticket nel Client Hello message.

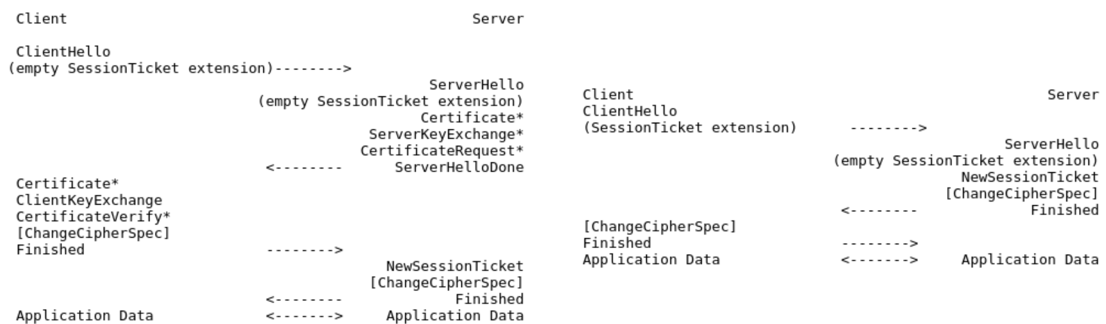


FIGURA 4.6: TLS Session Tickets examples.[39]
Left: session ticket issuance. Right: resume with session ticket.

4.3 Key Exchange

In TLS la sicurezza della sessione dipende pesantemente su una chiave condivisa di 48 byte (384 bit = 2^{384} combinazioni) che è il *master secret*. Il ruolo di questa fase è di generare il *premaster-secret* dal quale poi il master secret viene costruito.

TLS supporta diversi tipi di algoritmi di key exchange tra cui i più utilizzati sono: **RSA**, **DHE_RSA**, **ECDHE_RSA**, **ECDHE_ECDSA**. La prima sigla indica l'algoritmo di key exchange, la seconda indica l'algoritmo di firma utilizzato.

4.3.1 RSA

RSA proveniente dalle iniziali dei tre inventori *Ronald Rivest*, *Adi Shamir* e *Leonard Adleman* è un noto algoritmo standard di autenticazione e, a differenza di altri, anche di cifratura. [66]

Lo scambio di chiavi RSA è piuttosto semplice da spiegare: il client genera un premaster-secret di 46 byte, lo cifra con la chiave pubblica del server e lo invia nel messaggio *ClientKeyExchange*. Il server ottiene il PMS semplicemente decifrando il messaggio. Matematicamente l'algoritmo sfrutta l'elevata complessità computazionale del calcolo della fattorizzazione in numeri primi, che impedirebbe ad un attaccante di effettuare brute-force sulla chiave in tempo "utile". [66]

Questa semplicità purtroppo è il suo punto debole, se un domani la chiave privata del server venisse compromessa allora si potrebbero recuperare **tutti** i PMS utilizzati e dunque successivamente decifrare tutte le comunicazioni avvenute e registrate fino a quel momento.

Un algoritmo che non permette questo si dice che supporti la **forward secrecy**[1]. Motivo per cui attualmente RSA non è più utilizzato da solo bensì affiancato ad altri algoritmi di key exchange.

4.3.2 Diffie-Hellman Key Exchange

DH o *Diffie-Hellman* key exchange è un *key agreement* protocol ovvero un protocollo che permette a due o più parti di accordarsi su una chiave in modo sicuro da attacchi passivi. [66]

NOTA: Non protegge da attacchi attivi in quanto in una situazione di *man-in-the-middle* un attaccante potrebbe spacciarsi per l'altra parte, motivo per cui è fondamentale utilizzare DH con **autenticazione** (esempio DHE_RSA).

L'algoritmo si basa su una funzione matematica veramente semplice da calcolare in un verso ed estremamente onerosa computazionalmente da calcolare nell'altro.

DH richiede 6 parametri:

- **2 Domain Parameters** chiamati dh_p e dh_g selezionati dal server.
- **4 Parametri addizionali** 2 dal server e 2 dal client generati durante la negoziazione.

[1] PFS: Perfect Forward Secrecy è una proprietà dei protocolli di negoziazione delle chiavi che assicura che se una chiave di cifratura a lungo termine viene compromessa, le chiavi di sessione generate a partire da essa rimangono riservate. Se la chiave usata per cifrare i dati non viene usata per generare altre chiavi, e a loro volta le chiavi a lungo termine da cui è stata generata non sono usate per derivare altre chiavi di sessione, la compromissione di una sola chiave di sessione permetterà l'accesso solo ai dati protetti da tale chiave.[66]

Successivamente quando entrambe le parti hanno ricevuto tutti i parametri saranno allora in grado di generare la chiave condivisa.

Si chiama DHE, E sta per *Ephemeral*, quando nessuno dei parametri viene riutilizzato.

La sicurezza dell'algoritmo quindi si basa strettamente sulla qualità di questi parametri scambiati. Può capitare che un server accetti, per errata configurazione, dei parametri deboli ed insicuri portando cosi alla luce attacchi come ***Triple Handshake Attack***. Inoltre in passato gli Stati Uniti hanno ristretto l'utilizzo di una crittografia che utilizzasse chiavi superiori ad una certa lunghezza per poter essere in grado, in caso di necessità, di poter decifrare ed analizzare le comunicazioni.[66] Questo ha portato a "troncare" la sicurezza (in termini di lunghezza delle chiavi) di alcune cipher suite soprannominandole *export cipher suites* ed ha permesso di sfruttarne le debolezze in attacchi come ***FREAK*** e ***Logjam*** che riprenderemo in seguito.

4.3.3 Elliptic Curve DH Key Exchange

Ephemeral ECDH key exchange è sostanzialmente simile a DH classico ma utilizza alle fondamenta una funzione matematica diversa, dal nome *elliptic curve*.

In pratica il server si preoccupa di scegliere (allo stesso modo delle cipher suites) una *elliptic curve* e un EC *point* che fungerà da Domain parameter. Il client dalla sua parte sceglierà ed invierà i suoi parametri ed entrambi potranno arrivare al premaster secret.

4.3.4 Elliptic Curve Digital Signature Algorithm

Non fa parte degli algoritmi di key-exchange ma di firma digitale. Estende DSA (*Digital Signature Algorithm* [66]) con l'utilizzo delle elliptic curve. È considerata una scelta migliore rispetto a RSA non tanto in quanto veloce e "next generation" quanto RSA è destinata prima o poi a diventare insicura. Questo perché essendo basata sulla scomposizione in fattori primi, è solo

un problema di tempo affinché nuovi metodi e hardware più performanti o addirittura *computer quantistici*[2] possano renderla inutilizzabile.

L'autenticazione in crittografia è la proprietà più importante per poter garantire l'integrità e la non repudiabilità della comunicazione e di ogni singolo messaggio. Infatti ogni algoritmo di key exchange è affiancato da uno di autenticazione:

4.4 Application Data Protocol

Questo protocollo ha il compito di trasportare messaggi dal livello applicativo i quali, per quanto riguarda TLS, sono solamente buffer di dati. Tutti i messaggi sono frammentati e cifrati dal Record layer secondo gli attuali parametri di sicurezza.

4.5 Alert Protocol

Gli *alert* sono utilizzati come meccanismo di notifica per informare l'altra parte di circostanze eccezionali. Generalmente sono utilizzati per indicare messaggi di errore con l'eccezione di *close_notify* la quale è usata per concludere normalmente una connessione. Ogni alert è provvisto anche di un livello di severità che può essere *warning* o *fatal*. Messaggi con severità *fatal* risultano in un immediata terminazione della connessione e successivo invalidamento del SessionID e/o SessionTicket. Tutti gli alert sono cifrati secondo gli attuali parametri della connessione.[35]

```
enum {
          close_notify(0),
          unexpected_message(10),
```

[2]nuovo dispositivo per il trattamento ed elaborazione delle informazioni che, per eseguire le classiche operazioni sui dati, utilizza i fenomeni tipici della meccanica quantistica, come la sovrapposizione degli effetti e l'entanglement.[66] Il suo avvento introduce inoltre il concetto di *crittografia quantistica*

```
        bad_record_mac(20),
        decryption_failed_RESERVED(21),
        record_overflow(22),
        decompression_failure(30),
        handshake_failure(40),
        no_certificate_RESERVED(41),
        bad_certificate(42),
        unsupported_certificate(43),
        certificate_revoked(44),
        certificate_expired(45),
        certificate_unknown(46),
        illegal_parameter(47),
        unknown_ca(48),
        access_denied(49),
        decode_error(50),
        decrypt_error(51),
        export_restriction_RESERVED(60),
        protocol_version(70),
        insufficient_security(71),
        internal_error(80),
        user_canceled(90),
        no_renegotiation(100),
        unsupported_extension(110),
        (255)
    } AlertDescription;
```

Questi sono possibili errori che portano a generare alert message.

Come abbiamo detto, per terminare una connessione in maniera corretta, entrambe le parti invieranno un alert di *close_notify*. Ogni messaggio scambiato dopo l'alert verrà ignorato. Questo è necessario per impedire *truncation attack* ovvero attacchi che mirano a terminare la comunicazione e/o bloccare ogni ulteriore messaggio scambiato. Senza di questo semplice protocollo di spegnimento le due parti non potrebbero sapere se siano effettivamente sotto attacco o se la connessione sia genuinamente terminata. [1]

4.6 Cifratura

Tutti e tre i tipi ovvero *stream, block e authenticated encryption* sono supportati da TLS. Alcuni esempi di algoritmi sono 3DES, AES, ARIA, CAMELLIA, RC4, ChaCha20. Vediamone schematicamente i passaggi principali.

Stream Encryption

1. **Autenticazione**: si esegue MAC del *Plaintext*, header e sequence number.
2. **Cifratura**: si cifrano plaintext e MAC insieme diventando *ciphertext*.
3. Si ingloba il ciphertext in un record layer aggiungendo l'header.

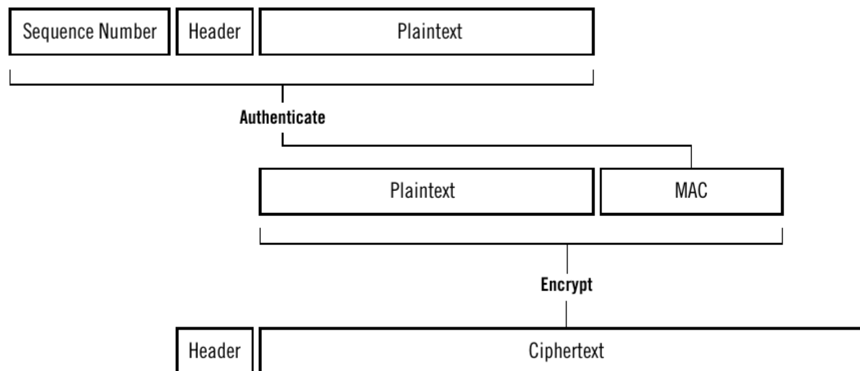


FIGURA 4.7: Stream encryption [1]

L'approccio di effettuare prima un MAC dei dati in chiaro e poi cifrare il tutto è denominato *mac-then-Encrypt*.

Block Encryption

1. **Autenticazione**: calcola il MAC di sequence number, header e plaintext.
2. Costruisce il *padding* necessario per raggiungere la lunghezza multipla del blocco.
3. Genera un impredicibile *Inizialization Vector* della stessa lunghezza del blocco.
4. Utilizza, una *block mode* per cifrare plaintext, MAC e padding.
5. Invia ciphertext ed IV insieme.

Qui a causa del padding l'approccio è meglio detto *mac-then-pad-then-encrypt*. Proprio a causa di comuni errori, in alcune implementazioni, nella gestione del padding, questo è diventato una vulnerabilità sfruttabile da alcuni tipi di attacchi. Questo perché il padding **non è incluso** nel MAC, dunque non può essere considerato *trusted*.

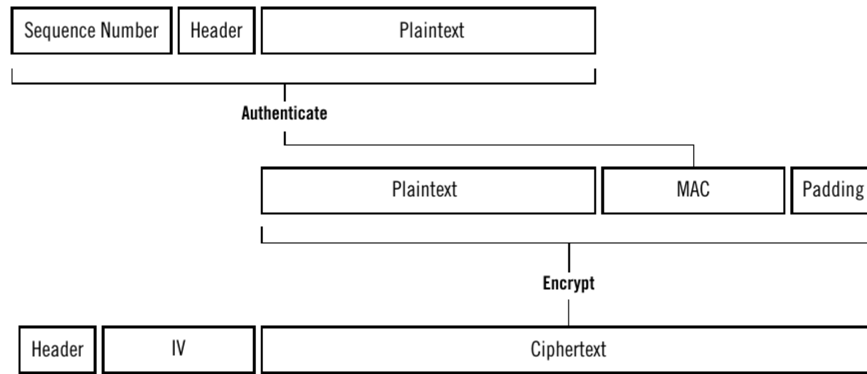


FIGURA 4.8: Block encryption [1]

Un diverso approccio infatti chiamato *encrypt-then-mac* si assicura di effettuare prima una cifratura del plaintext e del padding e solamente poi si effettua il MAC del tutto.

Crittografia autenticata

Gli *authenticated ciphers* combinano insieme cifratura e validazione di integrità in un unico algoritmo. Il nome completo è *authenticated encryption with associated data* (AEAD) [66]. Sono una via di mezzo tra stream e block cipher, non utilizzano padding ed inizialization vector bensì usano invece un valore speciale chiamato *nonce* il quale deve essere sempre unico.

Proprio perché riunisce i vantaggi ed elimina gli svantaggi delle due modalità precedenti, gli authenticated cipher offrono attualmente la miglior scelta a TLS.

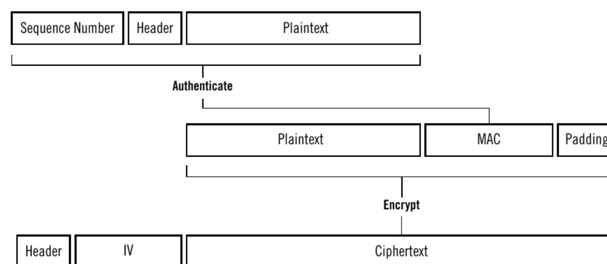


FIGURA 4.9: Authenticated encryption [1]

4.7 Rinegoziazione

Quando una *rinegoziazione* della connessione TLS è richiesta, allora un nuovo handshake deve essere eseguito per accordarsi su nuovi parametri di sicurezza.

Se il client vuole richiedere la rinegoziazione non farà altro che inviare un nuovo *ClientHello* message. Questo è chiamato *client-initiated-renegotiation*. Se il server invece vuole rinegoziare allora invierà un *HelloRequest* message indicando al client di smettere qualsiasi operazione e reiniziare con il processo di handshake. Questo è chiamato *server-initiated-renegotiation*.

La prima cosa da capire è che la rinegoziazione avviene all'interno del canale crittografico già esistente, dunque il suo contenuto è "protetto". Alla fine del nuovo handshake, client e server hanno generato dei nuovi parametri che si sostituiscono a quelli preesistenti.

Ma quando questa torna utile?

- **Cambio requisiti di sicurezza**: quando, ad esempio, all'interno di un website si voglia passare ad un'area che necessiti di un ancor più elevato livello di protezione. A questo punto il server rinegozierà la connessione con il client impostando nuovi parametri di sicurezza.

- **Certificato client richiesto**: quando, similmente a come appena detto, il server richiede una autenticazione da parte del client per entrare in certe aree di un sito web, dunque chiederà di rinegoziare.

NOTA: questo tipo di approccio ha un importante vantaggio, quello dell'*information hiding*. Questo perché il secondo handshake è crittografato con il significato che un attaccante passivo non è nemmeno in grado di osservare il certificato del client con conseguente sua "anonimità". Ad esempio il protocollo Tor può utilizzare la rinegoziazione in questo modo.[1]

Purtroppo di per sé la rinegoziazione in TLS ha avuto un effetto negativo in quanto contenente una vulnerabilità chiamata *TLS authentication gap* [15]. Un ipotetico attaccante in una posizione di man-in-the-middle potrebbe essere in grado di iniettare dei dati all'interno del canale protetto, ne rivedremo fra poco una spiegazione. Questo è il motivo per cui una estensione di TLS chiamata *secure renegotiation* (RFC 5746) [37] è stata successivamente aggiunta.

4.8 Operazioni crittografiche

Fino ad ora abbiamo parlato di vari aspetti del protocollo TLS senza andare nel dettaglio di cosa sia effettivamente il *master secret* e come esso venga generato.

In TLS si utilizzano delle funzioni chiamate *pseudorandom* (PRF) che sono in grado di generare una quantità arbitraria di dati effettivamente random. Queste funzioni prendono in input un *seed*, una stringa unica detta *label* ed un *secret*. Da TLS 1.2 in avanti ogni cipher suite deve esplicitare quale sia la funzione PRF utilizzata, il più delle volte basata su HMAC e SHA256. [35]

TLS definisce una funzione detta di *data expansion* chiamata *P_hash* basata su HMAC che serve a generare la quantità di dati richiesti.

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
```

Dove A() è definita come:

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

In TLS la PRF è creata applicando P_hash:

```
PRF(secret, label, seed) = P_hash(secret, label + seed)
```

L'introduzione di seed e label permette allo stesso secret di essere riutilizzato in contesti differenti producendo sempre output diversi.

Inoltre TLS utilizza sei diverse chiavi per crittografare simmetricamente, le quali vengono tutte derivate da questo master secret comune. Queste chiavi sono:

- client_write_MAC_key$[SecurityParameters.mac_key_length]$
- server_write_MAC_key$[SecurityParameters.mac_key_length]$
- client_write_key$[SecurityParameters.enc_key_length]$
- server_write_key$[SecurityParameters.enc_key_length]$
- client_write_IV$[SecurityParameters.fixed_iv_length]$
- server_write_IV$[SecurityParameters.fixed_iv_length]$

Le "MAC_key" vengono utilizzate per l'autenticazione dei messaggi, le "write_key" per la cifratura/decifratura e le ultime due sono inizialization vector (usati in base all'algoritmo scelto). Dunque su sei valori generati le chiavi effettive sono quattro, poiché utilizzate in primitive crittografiche, le altre due essendo inizialization vector non vengono considerate "chiavi".

Vediamo ora come si arriva al *master secret*.

Master secret

Come abbiamo visto prima, alla fine della fase di key-exchange entrambe le parti avranno un premaster secret. Questo valore viene processato utilizzando la PRF per generare un master secret di 48 byte.

```
master_secret = PRF(pre_master_secret, "master␣secret",
                    ClientHello.random + ServerHello.random)
```

Generazione delle chiavi di sessione

Queste chiavi vengono derivate da una singola invocazione di PRF, sono basate sul master secret e hanno come seed i valori random inviati da client e server.

```
key_block = PRF(SecurityParameters.master_secret, "key␣expansion",
                  SecurityParameters.server_random +
                  SecurityParameters.client_random);
```

Il "key_block" generato viene poi diviso nelle sei chiavi corrispondenti. Perché ci sono sei chiavi e non tre usate simmetricamente sia da server che da client? Per avere la certezza che un messaggio prodotto da una parte, utilizzando il suo proprio set di chiavi, non possa essere attribuito all'altra.

Quando si riprende una sessione viene utilizzato lo stesso master secret per la generazione delle chiavi. In ogni caso vengono aggiunti come seed i valori casuali inviati da client e da server e dunque il key_block sarà sempre differente.

4.9 Cipher suites

Una cipher suite è una selezione di primitive crittografiche ed altri parametri che definiscono esattamente come la sicurezza verrà implementata. Ogni suite è definita da diversi attributi:

- Metodo di autenticazione
- Metodo di key exchange
- Algoritmo di cifratura
- Lunghezza della chiave di cifratura
- Cipher mode
- Algoritmo MAC
- PRF utilizzata (solo da TLS 1.2)
- Funzione hash utilizzata per il *finished message*
- Lunghezza del campo *verify_data*

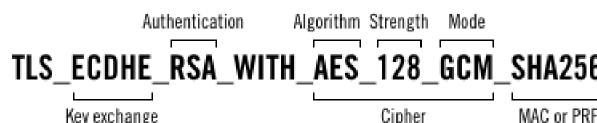


FIGURA 4.10: Cipher suite name construction [1]

Da notare che le suite che utilizzano crittografia autenticata non mostreranno l'algoritmo MAC nell'ultimo segmento ma quello PRF. Questo perché intrinsecamente queste cipher modes, come ad esempio GCM o CCM, uniscono controllo di integrità (MAC) e cifratura dentro un unico livello.

Quando vengono usate delle *elliptic curve* allora queste vengono specificate in fondo, ad esempio:

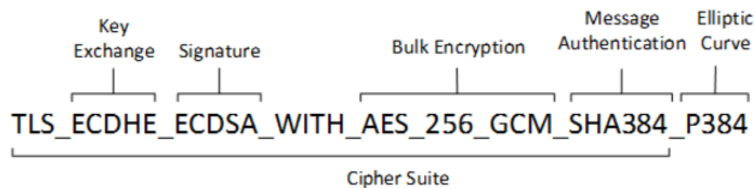


FIGURA 4.11: Cipher suite name construction [46]

Perché la crittografia estesa con le elliptic curve (EC) ha avuto, sta tuttora avendo successo e molto probabilmente ne avrà ancora più in futuro? Perché

permette di generare chiavi di lunghezza molto minore ad RSA e DH (per esempio) pur mantenendo lo stesso livello di sicurezza. Un esempio lampante: una chiave da 163 bit ECC equivale in termini di sicurezza ad una da 1024 bit DH/DSA/RSA. [2]

Symmetric	ECC	DH/DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

FIGURA 4.12: Comparable key sizes (in bits) [2]

Key Length (bits)		Time (s)	
RSA	ECC	RSA	ECC
1024	163	0.16	0.08
2240	233	7.47	0.18
3072	283	9.80	0.27
7680	409	133.90	0.64
15360	571	679.06	1.44

FIGURA 4.13: Key generation Performance [55]

È un algoritmo che non necessita di grande potenza computazionale e si adatta molto bene anche ad hardware di device mobile. [55] Nonostante per certe operazioni sia leggermente più lento di RSA, sembra che il mondo della crittografia e la *National Security Agency* (NSA) con la *NSA Suite B* [20] ne stiano incoraggiando sempre più l'utilizzo, forse anche in previsione di quelli che saranno *computer quantistici*.

4.10 TLS Extensions

Vediamo ora alcune estensioni di TLS, le quali permettono di aggiungere funzionalità senza dover cambiare ogni volta il protocollo stesso. Le estensioni sono aggiunte sotto forma di blocchi alla fine dei ClientHello e ServerHello message.

Server Name Indication (SNI)

Type: server_name (0x0000)
Permette al client di indicare al server il nome di dominio che sta cercando di

contattare, questo per facilitare la connessione a server che ospitano diversi *virtual host*.

Trusted CA Indication

Type: trusted_ca_keys (0x0003)
Permette ai clients di indicare una lista di root CA che supportano.

Certificate Status Request

Type: status_request (0x0005)
Utilizzato dai clients per indicare il supporto per *OCSP stapling*. Questa è una funzionalità del server che permette di inviare direttamente al client i dati riguardanti la revoca dei certificati. Il server quindi risponderà incorporando una *OSCP response*.

Signature Algorithms

Type: signature_algorithms (0x0013)
Permette al client di comunicare il supporto per vari algoritmi di hashing e firma sotto forma di coppia *signature-hash*.

Heartbeat

Type: heartbeat (0x0015)
Heartbeat è un estensione che aggiunge il supporto per il *keep-alive*, ovvero un controllo costante che l'altra parte sia sempre raggiungibile e per il *path maximum transmission unit discovery (PMTU)* di TLS[3] e DTLS[4]. Nonostante TCP abbia già una funzionalità *keep-alive*, Heartbeat si rivolge maggiormente a UDP, come protocollo inaffidabile.

[3] MTU: è la più grande dimensione di dati che può essere spedita singolarmente. Quando due parti comunicano direttamente possono scambiarsi i loro valori di MTU. Quando però la comunicazione passa attraverso diversi *hops* allora è necessario un meccanismo che li percorra tutti e scopra quale sia l'MTU effettivo valido per ogni tratto.[1]

[4] DTLS: Datagram Transport Layer Security, ovvero TLS per connessioni basate su *datagrammi* come UDP.

Heartbeat è un sotto-procotollo di TLS, ovvero i suoi messaggi possono essere scambiati affianco a messaggi di *application data*. Inoltre RFC stabilisce che questi messaggi possano essere scambiati solamente una volta terminato l'handshake.

Una terribile falla nell'implementazione di questo protocollo ha portato ad un attacco chiamato **Heartbleed** dalle conseguenze potenzialmente catastrofiche per il protocollo TLS stesso.

Application Layer Protocol Negotiation

Type: application_layer_protocol_negotiation (0x0010)
Questa è una estensione che abilita la negoziazione di diversi protocolli applicativi sopra una connessione TLS. Con ALPN un server attivo su porta 443 può offrire l'utilizzo di HTTP 1.1 ma anche altri protocolli come SPDY o HTTP 2.0.

Extended Master Secret

Type: Extended Master Secret (0x0017)
Quando il client utilizza questa estensione durante un *full handshake* allora il master secret viene calcolato come:

```
master_secret = PRF(pre_master_secret, "extended␣master␣secret",
                    session_hash)
```

Dove per "session hash" si intende un hash di tutti i messaggi di handshake. Questo dipende anche da i valori random scambiati tra client e server, in aggiunta a tutti gli altri parametri. Questa scelta riflette la raccomandazione di legare ogni materiale crittografico al contesto esclusivo che lo calcola e lo utilizza. [38]

Secure Renegotiation

Type: renegotiation_info (0xff01)
Questa estensione migliora TLS con la verifica che la rinegoziazione venga portata a termine tra le stesse due parti che inizialmente hanno completato l'handshake.

Durante il primo handshake le due parti si comunicano che supportano la rinegoziazione sicura semplicemente inviandosi l'estensione vuota. Ad ogni successivo handshake l'estensione è utilizzata per dare prova di essere la stessa parte dell'handshake precedente. Questo è ottenuto reinviando il valore *verify_data* contenuto nel *Finished message* precedente. Un attaccante non potrebbe ottenere questi dati poichè lo stesso Finished message è crittografato.

Eliptic curves

Type: elliptic_curves (0x000a)
Il client può indicare quali curve è in grado di supportare, anche se le più supportate e consigliate dal *National Institute of Standards and Technology (NIST)* sono due: `secp256r1 (P-256)` e `secp384r1 (P-384).`

4.11 Differenze tra varie versioni SSL/TLS

Ora che abbiamo visto una panoramica del funzionamento di TLS 1.2 vediamone le differenze con i protocolli precedenti.

SSLv3

Rilasciato nel 1996 e riprogettato per evitare le vulnerabilità presenti nella versione precedente, SSL 3 stabilì un design che seppur con grandi cambiamenti è rimasto alla base nei protocolli successivi. Nel Giugno 2015 è stato ufficialmente deprecato con l'RFC 7568 in quanto ormai largamente considerato insicuro. Alcune delle possibili vulnerabilità possono essere mitigate con le estensioni, ma queste ultime sono compatibili solamente con TLS e non SSL. Oltre alle limitate capacità, il protocollo non è stato in grado di offrire sicurezza e dunque molte vulnerabilità hanno spinto a progettare il successivo TLS 1.0.[32]

TLSv1.0

Rilasciato nel 1999 [34] include i seguenti cambiamenti:

- Per la generazione del master secret si utilizza una PRF invece di una funzione custom. Questa PRF è la prima ad effettuare una combinazione di XOR con HMAC-MD5 e HMAC-SHA1:

```
PRF(secret, label, seed) = P_MD5(S1, label + seed) XOR
                           P_SHA-1(S2, label + seed);
```

- Il campo *verify_data* è ora basato su una PRF invece di una funzione custom.
- Il controllo di integrità (MAC) ora usa l'ufficiale HMAC.
- Cambia il formato del padding.
- Le suite crittografiche *FORTEZZA* vengono rimosse.

Le differenze con il protocollo SSLv3 non erano elevate ma tali da rendere i due protocolli incompatibili. Per questo TLS 1.0 possiede un meccanismo per effettuare un *downgrade* del protocollo a SSLv3.

TLSv1.1

Rilasciato in Aprile 2006 include diverse migliorie alla versione precedente:

- Per difendersi da attacchi al padding il protocollo richiede di utilizzare l'alert *bad_record_mac*.
- La modalità CBC ora utilizza espliciti inizialization vectors inclusi in ogni record TLS.
- Prima versione di TLS a supportare le estensioni.

TLSv1.2

Rilasciato in Agosto 2008 [1] include vari ed importanti miglioramenti a TLS 1.1:

- Aggiunto il supporto per la crittografia autenticata (AEAD)
- Aggiunto supporto per HMAC-SHA256.
- Le cipher suite IDEA e DES sono rimosse.
- La combinazione MD5/SHA1 per la PRF sono rimpiazzate da SHA256.
- Le cipher suite permettono di specificare la loro PRF.

- la coppia MD5/SHA1 usata per la firma digitale viene rimpiazzata con un hash singolo, SHA256.
- L'estensione signature_algorithms permette ai clients di specificare l'algoritmo di firma preferito.
- La lunghezza del campo *verify_data* nel Finished message può essere ora specificata dalla cipher suite.

4.11.1 TLSv1.3

TLS versione 1.3 è tutt'ora in stato di *working draft*[5] ma alcune compagnie come Cloudflare stanno già iniziando a renderlo disponibile ai loro clienti. [10] Rispetto alla versione precedente, TLS v1.3 porta miglioramenti in termini di velocità e di sicurezza.

Per quanto riguarda l'instaurazione della connessione si è cambiato l'approccio rispetto a TLSv1.2 ed invece di dover aspettare due *round trip* per completare la fase di hanshaking se ne aspetta solamente uno.

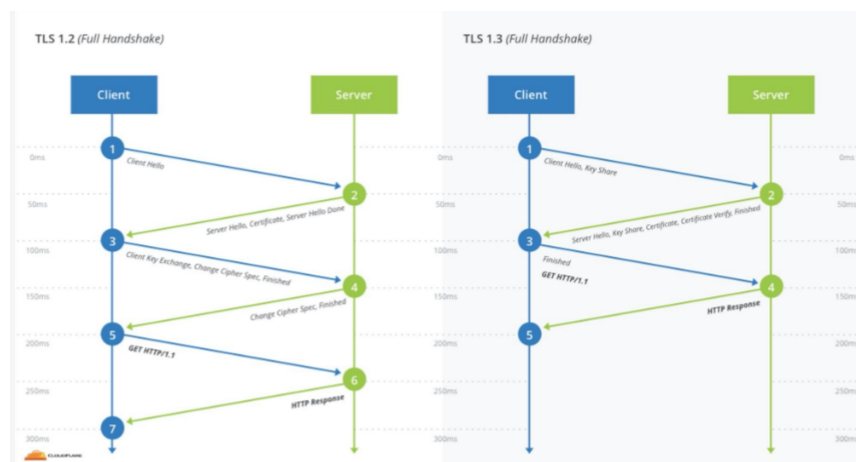


FIGURA 4.14: Differenze tra handshake TLSv1.2 e TLSv1.3 [10]

Inoltre TLS 1.3 supporta una funzionalità speciale: se un client ha già visitato un sito web, allora potrà essere in grado di inviare dati già dal primo messaggio. Questo si chiama *"zero round trip" mode* (0-RTT) e risulterà in connessioni ancora più veloci. Se si combina TLS 1.3 0-RTT e HTTP/2 la differenza in termini di velocità sarà notevole, soprattutto per dispositivi mobile.

[5]Internet draft: documento pubblicato dalla Internet Engiinering Task Force (IETF) contentente informazioni preliminari e risultati di ricerche ed altre informazioni tecniche.[66]

La velocità della modalità 0-RTT viene però ad un costo, ovvero quella di essere suscettibile ad attacchi di tipo *replay*. Se un attaccante potesse registrare i messaggi 0-RTT inviati dal client per poi rinviarli successivamente, il server non noterebbe la differenza e li processerebbe come validi. [11]

Per quanto riguarda il lato sicurezza TLS 1.3 rimuove tutti gli algoritmi obsoleti e considerati insicuri di TLS 1.2 come ad esempio:

- RSA *key transport* perché non garantisce *forward secrecy*
- SHA-1 per hashing (in favore di SHA-2)
- Stream cipher che utilizzino RC4
- Stream cipher che utilizzino DES/3DES
- MD5 insicura e deprecata
- AES-CBC responsabile per BEAST e Lucky 13
- EXPORT-CIPHER responsabili per FREAK e Logjam

Rimuovendo vecchi algoritmi e cipher suite e semplificando il protocollo stesso, la configurazione e l'implementazione verranno di conseguenza agevolate e rese meno prone ad errori di programmazione. [10]

Capitolo 5

Attacchi conosciuti a SSL/TLS

Durante gli anni la sicurezza dei protocolli SSL e poi TLS è stata continuamente messa alla prova portando alla luce sempre nuovi tipi di attacchi. Lo studio e la ricerca che hanno permesso questi ultimi sono il fattore trainante allo sviluppo ed avanzamento dei protocolli crittografici stessi.

In questo capitolo vedremo alcuni attacchi a SSL e TLS ed anche qualche caso in cui degli errori prettamente implementativi hanno portato a conseguenze davvero preoccupanti sul lato della sicurezza.

5.1 Insecure renegotiation

Questa debolezza, detta anche *TLS Authentication Gap* intrinseca del meccanismo di rinegoziazione stessa è stata scoperta nell'Agosto 2009. [1] La vulnerabilità esiste poiché a livello di protocollo, non c'è continuità tra il vecchio stream TLS ed il nuovo, anche se entrambi vengono generati sulla stessa connessione TCP.

Questo avviene per colpa di TLS che crittograficamente non lega l'esistente canale TLS con il nuovo handshake lasciando così una finestra aperta per un ipotetico attaccante. In parole povere, il server non controlla che la stessa entità sia dietro ad entrambe le conversazioni prima e dopo la rinegoziazione. Vediamone brevemente il funzionamento.

1. L'attaccante intercetta la richiesta di connessione dalla vittima al server.
2. Lui stesso apre una connessione TLS con il server ed invia un *attack payload*[1].

[1]attack payload: ciò che permette all'attaccante di portare a termine con successo il suo attacco, può essere ad esempio del codice che verrà eseguito o delle semplici stringhe. In questo caso è una porzione di richiesta HTTP.

3. Da questo momento in avanti agirà da semplice proxy facendo passare le richieste dal client al server, compreso il ClientHello iniziale.

4. Il client non è a conoscenza che sta effettuando una rinegoziazione ed il server "crederà" che i messaggi ricevuti precedentemente siano comunque provenienti dal client stesso.

5. A questo punto il *payload* inviato dall'attaccante verrà riconosciuto valido ed "appeso"[2] sopra al primo messaggio inviato dal client originale.

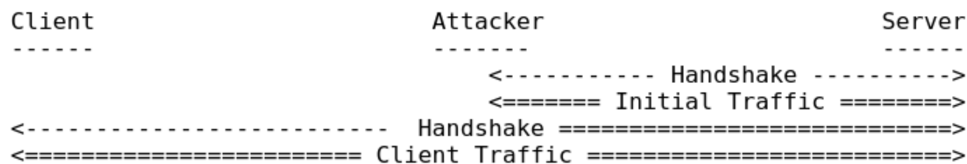


FIGURA 5.1: Renegotiation attack [37]

Facendo un esempio concreto con delle richieste HTTP (GET). Se un attaccante, prima di inoltrare il ClientHello message proveniente dal client, inviasse:

```
GET /pizza?top=olive;addr=attackersaddr HTTP/1.1
X-Ignore-This:
```

Ed il client come prima richiesta inviasse:

```
GET /pizza?top=sausage;addr=victimssaddr HTTP/1.1
Cookie: victimscookie
```

Il risultato al server diverrebbe:

```
GET /pizza?top=olive;addr=attackersaddr HTTP/1.1
X-Ignore-This: GET /pizza?top=sausage;addr=victimssaddr
HTTP/1.1
Cookie: victimscookie
```

In questo modo l'attaccante ha violato l'integrità dei dati ed è in grado di iniettare dati arbitrari all'inizio della connessione, motivo per cui una estensione di TLS chiamata *secure renegotiation* (RFC 5746) [37] è venuta in soccorso.

Ovviamente questo tipo di vulnerabilità apre molte possibilità ad un attaccante il quale è ora in grado di modificare le prime righe dell'header HTTP (come nell'esempio) della prima richiesta all'interno di una connessione HTTPS.

[2]"appeso" poichè spesso accade che in HTTP il server mantenga nel buffer i dati ricevuti dal client prima della rinegoziazione per ripresentarli al livello applicativo appena questa è terminata.[1]

Un attaccante potrebbe:

- **Redirigere l'utente verso un altro sito**: sempre controllato dall'attaccante e solitamente per motivi di *phishing*[3].
- **Downgrade della connessione**: a HTTP in chiaro se un attaccante è in grado di redirigere la connessione del client verso qualunque altro sito web.
- **Catturare le credenziali dell'utente**: mediante una richiesta POST "rediretta". Nel caso il sito potesse utilizzare il codice di redirect *307*[4] allora l'attaccante potrebbe essere in grado di redirigere totalmente la richiesta POST fatta dall'utente, e quindi contenente le informazioni da lui immesse, verso un sito a sua scelta.

Abbiamo capito che la rinegoziazione così come è stata originariamente progettata non è assolutamente utilizzabile. Un immediato possibile *fix* potrebbe essere quello di disabilitarla totalmente ma questo porta anche a sconvenienti in quanto la rinegoziazione è talvolta necessaria. Quindi l'opzione successiva è diventata esclusivamente l'estensione di TLS chiamata *secure renegotiation*.

5.2 BEAST

Nel 2011 un attacco verso i protocolli TLS 1.0 ed inferiori fu scoperto e denominato BEAST (*Browser Exploit Against SSL TLS*) [66], il quale, sfruttando le debolezze della modalità CBC (Cipher Block Chaining) e degli IV (inizialization vector), permette di estrarre porzioni di testo dal canale crittografico. [13]

Il problema è che in TLS 1.0 ed inferiori, un attaccante, essendo in grado di osservare i messaggi cifrati, verrà anche a conoscenza di ogni IV utilizzato. L'attaccante sarà quindi sempre in grado di conoscere l'initialization vector utilizzato per il messaggio successivo azzerandone l'impredicibilità. Questo permette di degradare la modalità CBC a semplice ECB (Electronic Code Book), la quale è notoriamente insicura.

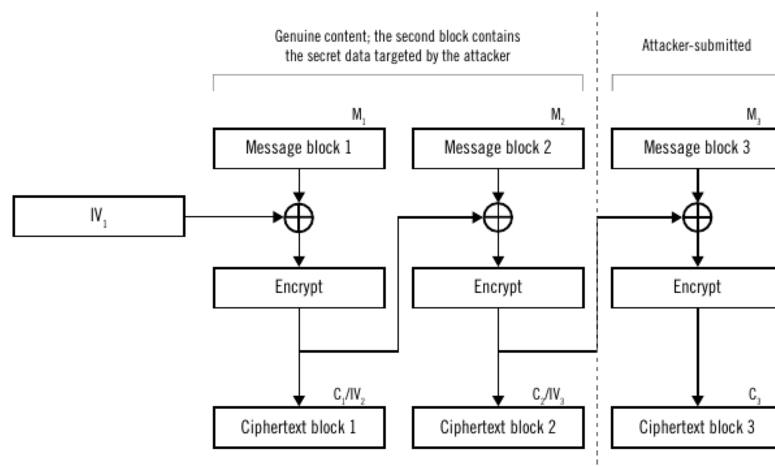
[3]phishing: quando si tenta di far credere all'utente di essere sul sito originale quando in realtà sta navigando su un sito maligno copiato a regola d'arte e gestito da un attaccante. Si effettua phishing per estorcere credenziali o altre informazioni all'utente finale.

[4]307 Temporary redirect: ovvero HTTP indica che la risorsa è temporaneamente indicata sotto un diverso URI e quindi lo user agent NON deve cambiare il metodo di richiesta ed eseguire una re-direzione automatica verso quel URI.

La differenza tra CBC ed ECB sta nel fatto che CBC utilizzi gli IV per mascherare ogni messaggio prima di essere crittografato. Questo rende il testo cifrato sempre differente anche se l'input fosse lo stesso, ma affinché funzioni gli IV devono essere unici. TLS 1.0 ed SSL utilizzano un solo blocco di dati random come IV per il primo messaggio, successivamente ogni messaggio cifrato sarà l'IV del messaggio successivo.

Un attaccante attivo in grado di inviare testo arbitrario da far crittografare al server può osservare i messaggi cifrati prodotti ed adattare quindi l'attacco. Il suo obiettivo è quello di rivelare il contenuto del secondo blocco, non il primo poiché il primissimo IV non è conosciuto, per poi procedere con quelli successivi. Dopo aver ricevuto i primi due blocchi l'attaccante si intromette ed inizia con i suoi tentativi. Per ogni tentativo che effettua, l'attaccante, conoscendo gli IV, può creare un messaggio *ad hoc* in modo che gli effetti di un mascheramento con l'IV vengano eliminati completamente. Quando un tentativo ha successo significa che la versione cifrata del messaggio è identica a quella prodotta dall'attaccante.

FIGURA 5.2: BEAST attack against CBC with predictable IV [1]



Vediamo matematicamente come procede l'attacco. (M3 è controllato ed inviato dall'attaccante):

$C_2 = E(M_2 \oplus IV_2) = E(M_2 \oplus C_1)$
$C_3 = E(M_3 \oplus IV_3) = E(M_3 \oplus C_2)$

Siccome l'attaccante conosce sia gli IV (quindi C_1 e C_2) può creare un M_3 in modo da neutralizzare il mascheramento. Assumendo M_g sia il tentativo:

$M_3 = M_g \oplus C_1 \oplus C_2$

Quindi la cifratura di M_3 sarà:

$C_3 = E(M_3 \oplus C_2) = E(M_g \oplus C_1 \oplus C_2 \oplus C_2) = E(M_g \oplus C_1)$

Dunque se il tentativo sarà corretto ed $M_g = M_2$, allora il blocco cifrato C_2 sarà uguale al blocco C_3. [1]

$C_3 = E(M_g \oplus C_1) = E(M_2 \oplus C_1) = C_2$

Ci sono altri fattori necessari all'attaccante per portare a termine con successo l'attacco, come ad esempio il fatto di essere in grado di inviare traffico arbitrario, il fatto di dover indovinare esattamente l'intero blocco ed altri. Ciononostante BEAST ha portato alla luce le varie debolezze del protocollo ed ha caldamente incoraggiato la produzione di TLS 1.1 il quale utilizza finalmente un IV casuale per ogni TLS Record.

La soluzione utilizzata per mitigare BEAST fu quella di adottare un meccanismo di split 1/n-1 dove il primo byte veniva inviato in un record a sé mentre tutti gli altri n-1 byte nel record successivo, creando così dei IV random ogni volta. Questo garantiva la sicurezza del secondo blocco lasciando maggiormente esposto il primo che in ogni caso conteneva solamente un byte di dati. Contenendo solo 1 byte, gli altri (per esempio) 15 erano randomizzati all'interno del blocco, rendendo così molto meno probabile per l'attaccante scoprirne il valore.

5.3 Compression attacks

Questa tipologia di attacchi prendono di mira la compressione dei dati utilizzata a livello TLS. In ordine di sviluppo sono stati creati CRIME, TIME e BREACH.

In generale la compressione lavora riducendo i bit ridondanti come ad esempio i caratteri ripetuti più di due volte. Si dice che esiste un *oracle* quando un attaccante ha la possibilità di inviare dei dati arbitrari nello stesso contesto dei dati da mantenere segreti ed è in grado di sapere se la risposta è corretta oppure no. Quale è lo scopo? Se una certa parola o frase è ripetuta, allora verrà salvata una sola volta. Un attaccante può prendere di mira per esempio un cookie HTTP ed inviare ogni volta dei caratteri sperando che siano gli stessi del cookie. Osservando poi l'output potrà sapere se i tentativi sono corretti o meno, come? Guardando la dimensione. Se il tentativo

è giusto allora la compressione diminuisce la dimensione dell'output, altrimenti se il tentativo introduce dati random allora la compressione non viene utilizzata e la dimensione aumenta.

5.3.1 CRIME

CRIME (*Compression Ratio Info-leak Made Easy* è un attacco *client-side* del 2012 operato dagli stessi autori di BEAST. [3] CRIME utilizza la compressione a livello TLS e Javascript per estrarre i cookie in un attacco di man-in-the-middle. Inoltre gli autori hanno dichiarato che l'attacco è in grado di recuperare un carattere utilizzando solamente sei richieste. [1] L'attaccante ha comunque bisogno di osservare la dimensione delle richieste e delle risposte.

L'attacco può essere preventivato semplicemente evitando l'uso della compressione a livello TLS. Il client nel ClientHello specificherà la lista di algoritmi di compressione supportati tra i quali il 'null' ovvero l'indicazione di non utilizzare la compressione (obbligatoria). Il server allora sceglierà il metodo da utilizzare e, siccome il client è forzato ad inviare sempre il byte 0 indicante "no-compression", potrà sempre rifiutare la compressione.

5.3.2 TIME

TIME è una versione migliorata di CRIME presentata in Marzo 2013 al Black Hat Europe. [9] Un fattore restrittivo di CRIME è che l'attaccante deve avere accesso alla stessa rete locale per poter osservare le dimensioni dei pacchetti. TIME invece, dal nome, utilizza le differenze tra le tempistiche per misurare la grandezza dei pacchetti.

L'attacco prende forma interamente nel browser della vittima mediante un tag html <*img*> utilizzato per inizializzare le richieste e del codice Javascript per misurarne il timing. Con questo cambiamento ora l'attacco può potenzialmente essere eseguito contro qualunque sito web. È da notare che l'attacco funziona osservando differenze di un byte nell'output compresso, quindi per riuscire ad avere un'accuratezza così elevata bisogna considerare e sfruttare le proprietà del livello di trasporto TCP come il controllo di congestione e le *congestion window* o "finestre di congestione".

In pratica TIME non ebbe particolare successo in quanto l'attacco è piuttosto intricato da eseguire in *real-life* a causa dei possibili ritardi nella comunicazione e dei meccanismi di controllo del flusso di TCP.

5.4 Lucky 13

Nel Febbraio 2013 *AlFardan* e *Paterson* pubblicarono un documento dove veniva spiegata la tecnica per poter recuperare parti di testo cifrato a patto che sia utilizzata una suite con modalità CBC. Queste "piccole parti di testo" sono solitamente sempre riferite a cookie HTTP. [4]

Il problema alla base di tutto è che il padding utilizzato nei cifrari a blocchi e nella modalità CBC **non è autenticato**. Infatti il meccanismo standard è denominato *mac-then-pad-then-encrypt*. Questo permette al padding di essere manomesso nella sua forma crittografata senza nessuna "violazione" di integrità. Questo attacco, utilizzando Javascript nel browser della vittima, permette di recuperare un byte di testo in chiaro in appena 8,192 richieste HTTP. [1]

Tutte le versioni di SSL e TLS sono affette da questo problema e potenzialmente vulnerabili quando utilizzano la modalità CBC. Ma come si arriva a recuperare i byte in chiaro?

Come prima abbiamo parlato di BEAST e del mascheramento del *plaintext* prima di essere crittografato, qui allo stesso modo se un attaccante riuscisse a rivelare la maschera allora potrebbe decifrare il testo. Facendo un passo indietro, l'obiettivo dell'attaccante è quello di inviare dei tentativi con lo scopo di scoprire un byte della maschera usato per la decifratura, una volta in possesso è possibile decifrare un byte di ciphertext. Ed il processo va avanti così. La chiave per eseguire un attacco è dunque in primo luogo inviare molti tentativi, in secondo trovare un modo per determinare se questi hanno avuto successo o meno.

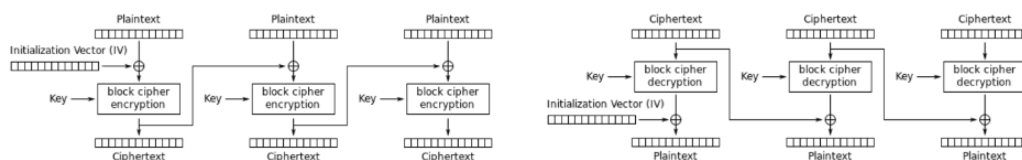


FIGURA 5.3: CBC mode of operation on the left is encryption, on the right decryption [66]

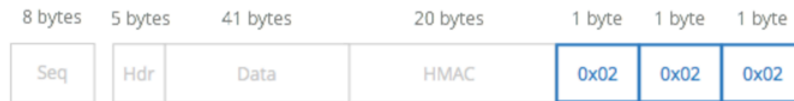


FIGURA 5.4: Struttura di un TLS Record con padding[12]

Dunque facciamo un riassunto, per cifrare un blocco:

1. Si calcola HMAC di plaintext + header e sequence number.
2. Si accostano plaintext, HMAC e si aggiunge il padding.
3. Si cifra il tutto.

Di conseguenza per decifrare:

1. Si decifra il blocco.
2. Si elimina il padding controllando il valore dell'ultimo byte.
3. Si calcola HMAC del plaintext e si confronta con l'HMAC in possesso, se sono identici il blocco è valido.

Per funzionare, l'attaccante deve essere in posizione di man-in-the-middle e quindi intercettare e modificare il traffico criffografato. L'attacco si basa su *padding oracle attack* conosciuto già dal 2002, ma questa volta migliorato in *timing padding oracle* aggiungendoci il fattore 'tempo impiegato'. Spieghiamolo meglio.

Padding Oracle Attack

Questo attacco si basa sul fatto che esista un *oracle* ovvero che dato un tentativo, l'attaccante possa ricevere conferma se tale tentativo sia giusto o meno. Qui l'obiettivo è di mascherare il padding nella modalità di cifratura CBC per poi essere in grado di decifrare il testo un byte alla volta.

Conoscendo il funzionamento di CBC e quindi delle varie operazioni di XOR effettuate per arrivare al plaintext, un attaccante, inviando al server una serie di tentativi nei quali ha opportunamente modificato il blocco precedente a quello da decifrare (si riguardi lo schema CBC), è in grado in base alla risposta del server di sapere se il padding tentato sia corretto o meno.

A questo punto se il padding risulta corretto l'attaccante è in grado di scoprire un byte di cleartext e continuare con lo stesso procedimento fino a conoscere l'intero blocco.

Tornando ora a Lucky13, qui si introduce il fattore tempo, ovvero si analizza quanto tempo il server impieghi per rispondere ad un tentativo fatto dall'attaccante. In base a quello si è in grado di sapere se il padding è corretto o meno. Questo perché nel caso in cui il padding sia corretto allora l'algoritmo procederà con l'effettuare il calcolo HMAC del plaintext ed impiegherà del tempo, invece se il padding si rivela errato il server invierà subito l'alert e terminerà così la connessione. Il numero 13 del nome deriva dal fatto che l'attacco sfrutti il meccanismo MAC di TLS che include nel calcolo i primi 13 byte contenenti l'header. [7]

5.5 POODLE

Nell'Ottobre 2014 viene riportato un altro attacco chiamato POODLE (*Padding Oracle On Downgraded Legacy Security*) ai danni di SSL 3.0 e la modalità CBC. La vulnerabilità è presente sia nei browser che nei server ed è insita nel modo in cui SSL 3.0 gestisce il padding dei blocchi CBC. [62] Il problema è che alcuni server detti *legacy*, sebbene utilizzino anche la più recente versione di TLS, ancora supportano il vecchio protocollo SSL 3.0. Inoltre TLS implementa un meccanismo di *downgrade* in caso il client non riesca a connettersi con l'ultimo protocollo offerto dal server oppure nel caso l'handshake fallisca ripetutamente. A questo punto il client scalerà il protocollo di una versione e tenterà così di ri-collegarsi al server. Il problema sta nel fatto che questo meccanismo può essere innescato volontariamente da un attaccante per effettuare il downgrade della connessione fino ad SSL 3.0, se il server ancora lo supporta. A questo punto sappiamo che SSL 3.0 può utilizzare un cifrario a flusso come RC4 oppure un cifrario a blocchi in CBC. Nel secondo caso allora si può procedere allo stesso modo dell'attacco BEAST per decifrare i blocchi di informazione. [16]

Quale è il rimedio? **TLS_FALLBACK_SCSV**, oltre a disabilitare SSL 3 ovviamente.

TLS Fallback Signaling Cipher Suite Value (SCSV) è una cipher suite che previene attacchi di tipo *protocol downgrade* sul protocollo TLS e DTLS. [36] Questo meccanismo vieta a due parti di accordarsi su un protocollo inferiore al più alto supportato da entrambi. Ad esempio, se client e server supportano TLS 1.2, TLS 1.1 e TLS 1.0, allora non sarà permesso loro accordarsi su un protocollo inferiore al più alto supportato da entrambi, ovvero TLS 1.2.

TLS_FALLBACK_SCSV non è esattamente una cipher suite, ma la sua presenza nel ClientHello è utilizzata come segnale per il server. Inoltre è stato aggiunto un alert di tipo *fatal* chiamato *inappropriate_fallback(86)* il quale viene scaturito dal server quando un client cerca di negoziare una versione del protocollo inferiore a quella disponibile.

E se invece si scegliesse di utilizzare un cifrario a flusso come RC4?

Qui purtroppo non ci sono buone notizie in quanto RC4 è stato analizzato e più volte è stato definito insicuro. Dopo la scoperta di BEAST e Lucky13 molti hanno smesso di utilizzare CBC e cifrari a blocchi e si sono riversati su RC4 rendendolo il più utilizzato del tempo. Già nel Marzo 2013 però cominciano ad essere resi pubblici attacchi sempre più pratici verso RC4. [47]
Il problema di fondo è che RC4, nel caso in cui debba cifrare lo stesso testo ripetute volte, non garantisce una assoluta diversità dei ciphertext corrispondenti. Questo permette ad un attaccante di monitorare i messaggi cifrati e, con alcune accurate tecniche, risalire al testo in chiaro. Quando può capitare che RC4 debba cifrare sempre le stesse informazioni? Quando, ad esempio, il client invia richieste HTTP contenenti i cookie, in quel caso gran parte del testo in chiaro è già conosciuto ad esclusione solamente del cookie. [29]

Inoltre *Mathy Vanhoef* e *Frank Piessens* hanno presentato un attacco a RC4 detto *RC4 NOMORE* il quale necessita di 52 ore per essere portato a termine con successo decifrando cookie web. [63]

5.6 Heartbleed

A volte può capitare, non per colpa di falle intrinseche ad un protocollo, che alcune implementazioni dello stesso contengano degli errori di programmazione. Questi possono rimanere nascosti per molto tempo fino al momento in cui un nuovo attacco viene scoperto ed è il momento di correre ai ripari.

Un esempio lampante è accaduto con *Heartbleed* che ha colpito la popolare libreria *OpenSSL* nel 2014.

Questo bug è una seria vulnerabilità per la libreria crittografica OpenSSL in quanto permette ad un attaccante di sottrarre informazioni riservate nel sistema vittima. Questo dà la possibilità a chiunque in Internet di leggere direttamente la memoria del sistema protetto dalla versione vulnerabile di OpenSSL, con conseguente potenziale compromissione delle chiavi private

utilizzate dal server stesso. Questo è quanto di peggio possa accadere per SSL/TLS in quanto una volta in possesso delle chiavi, un ipotetico attaccante avrebbe pieno controllo su ogni comunicazione e la potenza della crittografia si sgretolerebbe per intero.

Il problema è portato dalla estensione TLS di nome Heartbeat utilizzata per mantenere attiva la connessione SSL/TLS con il server. Qui è mancato in OpenSSL un controllo sulla dimensione di un valore nella specifica linea di codice:

```
memcpy(bp, pl, payload);
```

Dove il payload e la sua lunghezza sono potenzialmente controllabili dall'utente. Mettiamo caso un attaccante modifichi il messaggio Heartbeat sostituendo nel *payload_length* il valore con il valore massimo ottenibile.

La struttura di un messaggio Heartbeat secondo l'RFC 6520 è la seguente:

```
struct {
      HeartbeatMessageType type;
      uint16 payload_length;
      opaque payload[HeartbeatMessage.payload_length];
      opaque padding[padding_length];
   } HeartbeatMessage;
```

Allora il server (o il sistema verso la quale si sta inviando la richiesta) crederà che la dimensione del messaggio sia quella indicata (ad esempio 65535 byte o 64KB) e allocando la memoria per essi risponderà copiando, insieme ai dati ricevuti, anche tutto ciò che è contenuto nelle celle di memoria adiacenti fino ad arrivare alla quantità richiesta. OpenSSL non fa un controllo sulla effettiva dimensione del payload del messaggio Heartbeat e dunque permette di rispedire al mittente la stessa quantità di dati indicata nel campo *payload_length* anche se l'effettivo payload fosse di un paio di byte. Questo causa una perdita di informazioni in quanto restituendo al client una quantità di informazioni superiore rispetto a quella effettivamente richiesta, il server invia letteralmente ciò che ha "in memoria temporanea" in risposta al mittente. [31]

5.7 Conseguenze della Export-Cryptography

Possiamo parlare di *export cryptography* riferendoci agli anni 90' nella quale gli Stati Uniti hanno ristretto l'utilizzo di una crittografia superiore ad un certo livello. [66] Il limite delle chiavi era fissato a 40 bits per la crittografia

simmetrica e 512 per quella asimmetrica. Il problema è che queste cipher suite dette *export cipher suite* rimasero in uso per molto tempo dopo l'inizio del secondo millennio.

5.7.1 FREAK

Factoring RSA Export Keys ovvero FREAK è un attacco scoperto nel 2015 il quale permette ad un attaccante in posizione di man-in-the-middle di far instaurare una connessione tra un server che supporta le export cipher ed un client il quale non le supporta. Naturalmente ci sono alcuni requisiti per un attaccante da soddisfare: essere nella stessa rete locale per operare da man-in-the-middle, trovare un server che utilizzi le vecchie cipher suite e che riutilizzi la stessa chiave per un periodo prolungato. Dunque nel caso in cui un server accetti di negoziare una export cipher suite, un attaccante in posizione di man-in-the-middle potrebbe intromettersi e far accettare l'utilizzo di chiavi deboli. Per esempio, le chiavi RSA dovevano essere inferiori a 512 bits, questo le rendeva già nel 2015 fattorizzabili in circa sette ore spendendo un centinaio di dollari in cloud computing. [30]

Ma la domanda è: quanti server ancora nel 2015 supportano delle cipher suite vecchie di 15-20 anni?

> *"No matter how bad you think the Internet is, it can always surprise you."* [28]

Infatti in Marzo 2015 in base ad uno scan su milioni di siti web con certificati fidati da browser è stato rilevato che circa 36.7% di questi supportava export-RSA cipher. [18]

5.7.2 Logjam

L'attacco Logjam, riportato nel Maggio 2015, consente a un attaccante man-in-the-middle di degradare una connessione TLS vulnerabile ad utilizzare la crittografia di esportazione a 512 bit. Questo consente all'attaccante di leggere e modificare tutti i dati in transito sulla connessione protetta. L'attacco ricorda FREAK, ma è dovuto ad un difetto nel protocollo TLS anziché a una vulnerabilità di implementazione e attacca lo scambio di chiavi Diffie-Hellman anziché lo scambio di chiavi RSA. L'attacco influisce su qualsiasi server che supporta i cipher DHE_EXPORT e riguarda tutti i browser

web moderni. L'8,4% dei siti web della lista Top 1 million erano inizialmente vulnerabili. [23]

Vediamo in alcuni semplici passi come funziona questo *downgrade attack*:

1. Un attaccante MITM intercetta la richiesta di connessione dal client e rimpiazza tutte le cipher suites con solamente quelle DHE_EXPORT
2. Il server, se vulnerabile, sceglie dei parametri deboli a 512-bits, fa i suoi calcoli ed invia tutto il materiale al client assieme al certificato.
3. L'attaccante rimpiazza il nome della cipher suite scelta rimuovendo la sigla EXPORT ed inoltra gli altri dati intatti.
4. Il client a questo punto è portato a credere che il server abbia deciso volontariamente di utilizzare dei parametri deboli per DH. Dal suo punto di vista, non può immaginarsi che un attaccante abbia invece richiesto una cipher suite di tipo EXPORT.
5. A questo punto l'attaccante è in grado di "rompere" i parametri DH, recuperare le chiavi di sessione e procedere singolarmente con la connessione TLS verso il client dimenticandosi totalmente del server.

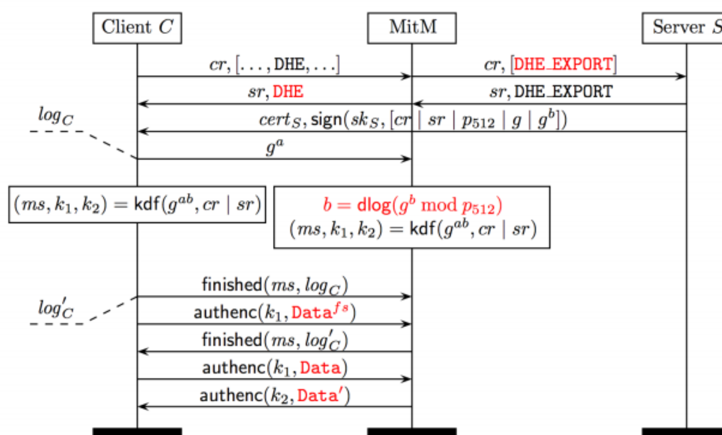


FIGURA 5.5: Logjam attack. [22]

Il rimedio a tutto ciò è semplice. Disabilitare qualunque cipher suite che utilizzi la EXPORT cryptography, ed aggiornare ogni device e software.

5.8 Triple handshake attack

In maggio 2014 un nuovo attacco di tipo *client impersonation* chiamato *Triple Handshake Attack* è stato presentato ai danni del protocollo TLS. [41] In breve

l'attaccante sfrutta una combinazione di debolezze di RSA, Diffie-Hellman, session resumption e rinegoziazione per poter poter agire da man-in-the-middle e successivamente avere accesso a tutto il canale crittografato.

Assumiamo che un client **C** si voglia connettere ad un server **S** e ci sia un attaccante **A**.

Debolezze del protocollo TLS

In questo attacco vengono identificate e sfruttate quattro potenziali debolezze:

1. Nell'hansdshake RSA un client C invia il pre-master-secret (PMS) al server A crittografato con la sua chiave pubblica. Se A fosse un ipotetico attaccante invece, potrebbe agire come se fosse lui stesso il client ed inviare lo stesso PMS su una nuova connessione al server S. A può utilizzare gli stessi valori random e identificatori di sessione inviati da C e S e sincronizzare così le due comunicazioni ottenendo due sessioni che condividono lo stesso master secret, le stesse chiavi ma con estremi diversi.
2. Nell'handshake DHE il server A sceglie dei parametri per DH. Se A fosse un ipotetico attaccante allora può scegliere dei parametri non primi in modo che il PMS risultante sia totalmente sotto il suo controllo. Successivamente come per RSA, l'attaccante può agire da man-in-the-middle tra client C e server S per ottenere le due sessioni con gli stessi parametri.
3. La *session resumption* su una nuova connessione utilizza quello che abbiamo chiamato handshake abbreviato il quale verifica solamente che entrambi le parti condividano lo stesso master secret, cipher suite ed identificatore di sessione. È da notare che in questa fase non si effettua nuovamente l'autenticazione di client e/o server quindi l'attaccante A può allo stesso modo di prima riesumare le sessioni precedenti semplicemente inoltrando l'handshake abbreviato intatto da una connessione all'altra. L'estensione di TLS *secure renegotiation* lega solamente handshake sulla stessa connessione ma non viene utilizzato se la sessione viene ripristinata su una nuova connessione.

4. Durante la rinegoziazione sia il certificato server che quello client possono cambiare. Questo è permesso da TLS ma non è indicato altrettanto come le applicazioni debbano reagire a questo cambio improvviso. [41]

Vediamo ora come avviene l'attacco.

Dal nome stesso vedremo che tre handshake verranno utilizzati in due fasi diverse. Nella prima fase accade quanto scritto nei punti 1. e 2. mentre nella seconda fase avremo il ripristino della connessione e quindi i punti 3. e 4.

Vediamoli con le figure appropriate.

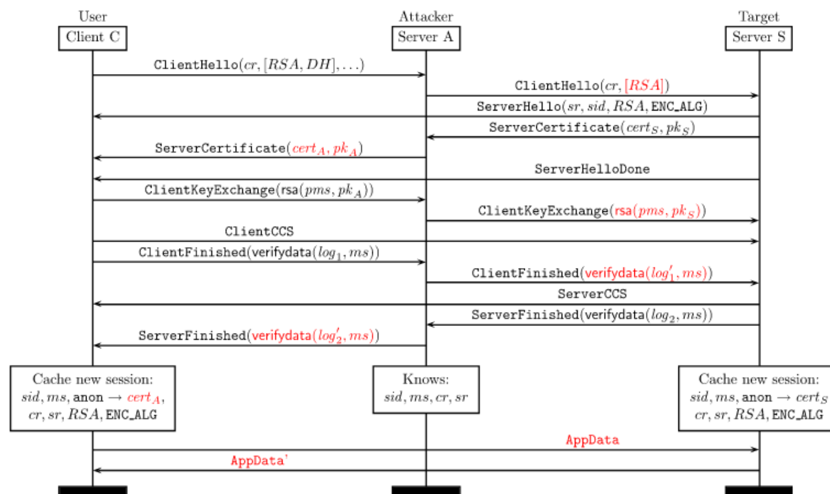


FIGURA 5.6: Prima fase del Triple Handshake Attack [41]

Come possiamo vedere qui l'attaccante si interpone nella comunicazione creando due connessioni totalmente indipendenti ma che condividono gli stessi identici parametri di sicurezza. Da notare che il Finished message sarà altrettanto diverso.

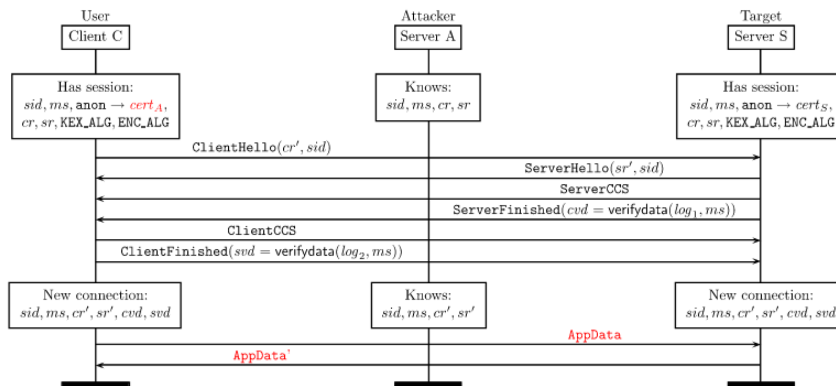


FIGURA 5.7: Seconda fase del Triple Handshake attack [41]

Nella seconda fase invece possiamo vedere che quando C tenta di ricollegarsi a S, in quel momento l'attaccante altro non farà che inoltrare il messaggio cosi com'è e dunque senza alcuna modifica. Da quel momento, cosa più importante, sia server che client avranno anche lo stesso Finished message. Questo è permesso proprio perché, pur cambiando il certificato utilizzato, alcune applicazioni potrebbero reagire semplicemente *ignorando* tale cambiamento.

A questo punto anche le prossime rinegoziazioni avranno successo poichè i campi *verify_data* saranno gli stessi da entrambe le parti della comunicazione.

Un'iniziale misura di sicurezza sarebbe quella di istruire i browser a rifiutare qualsiasi cambio di certificato da parte del server. Altri cambiamenti possono comportare un miglioramento nel calcolo del master secret rendendolo dipendente dall'handshake iniziale, ed un miglioramento nella procedura di handshake abbreviato. [41]

Capitolo 6

TLSScan

TLSScan è il nome del tool sviluppato per automatizzare l'analisi passiva di un sito web che utilizzi HTTPS. L'obiettivo è quello di valutare la sicurezza della comunicazione con tale sito tenendo in considerazione parametri di sicurezza fondamentali e conosciute debolezze che porterebbero ad eventuali attacchi.

6.1 Funzionalità

TLSScan è in grado di svolgere diverse attività:

- Effettuare una scansione dei protocolli SSL/TLS supportati dal server per le connessioni HTTPS. SSL2, SSL3, TLS1.0, TLS1.1 e TLS1.2 sono supportati da TLSScan.
- Effettuare tentativi di connessione con protocolli inferiori ai più alti supportati includendo la SCSV signal cipher suite.
- Effettuare un parsing della catena di certificati ricevuta dal server ed analizzare data di scadenza, algoritmo di firma, indirizzi CRL e OCSP ed altri parametri.
- Effettuare una scansione parallela per determinare il supporto del server, in base ai vari protocolli supportati, delle cipher suite abilitate all'uso. Inoltre le cipher suite verranno elencate in maniera da riflettere le preferenze del server.
- Verificare il supporto del server per EXPORT cipher suite e cipher suite che utilizzino ancora RC4.

- Effettuare degli handshake di test per determinare il supporto del server a varie estensioni TLS come per esempio Heartbeat, Secure Renegotiation, Session Ticket ecc, ecc.
- Effettuare un tentativo di connessione per verificare la disponibilità del server ad utilizzare metodi di compressione.
- Verificare il supporto da parte del server di OCSP Stapling.
- Verificare la risposta del server in caso si richieda un SNI (Server Name Indication) errato.
- Verificare se gli attacchi POODLE, BEAST, Heartbleed siano possibili.
- Possibilità di salvare report su file.
- Possibilità di salvare un full packet capture dello scan (solo se non anonimizzato)
- Possibilità di redirigere il traffico su proxy SOCKS5 TOR, anonimizzando tutto il traffico, richieste DNS comprese.
- Possibilità di aggiungere un timing per poter effettuare le richieste a distanza di n millisecondi.
- Verificare se il sito HTTPS utilizzi o meno HSTS.

6.2 Requisiti

Il tool è stato sviluppato in ambiente Arch Linux e testato con distro Ubuntu-based ma essendo basato su Python 2.7 dovrebbe svolgere il suo lavoro in maniera equivalente su tutte le altre distribuzioni Linux.

Una lista dei moduli necessari a TLSScan per l'esecuzione è contenuta nel file *requirements.txt*.

Per poter installare i moduli in maniera del tutto automatizzata si utilizza il gestore dei pacchetti di Python chiamato pip (pip2 per python 2.7). Con questo comando si potranno scaricare ed installare tutti i moduli necessari al tool per funzionare:

```
$ pip2 install -r requirements.txt
```

6.2.1 Analisi dei requisiti

Obiettivo	Analizzare, mediante le risposte ricevute dal server, l'utilizzo o meno di protocolli ed algoritmi crittografici considerati insicuri simulando il possibile comportamento di un web browser.
Precondizioni	Connessione ad Internet
	Moduli necessari installati
	Servizio Tor installato se si sceglie di anonimizzare il traffico
Successo esecuzione	Report stampato su terminale/file a seconda della scelta.
Insuccesso esecuzione	Problemi di connessione
	Host non disponibile
	Arresto imprevisto dell'applicativo

6.3 Struttura

TLSScan utilizza un framework chiamato Scapy-SSL-TLS [27] che permette una migliore gestione e creazione dei pacchetti di rete, SSL e TLS. Questo ci permette di poter lavorare in maniera agevole su richieste e risposte in ambito di una connessione crittografata SSL/TLS modificando facilmente i parametri interni ai pacchetti.

Il tool è strutturato in due file *tlsscan.py* e *TLSScanner.py*:

1. **tlsscan** è il launcher dell'applicativo, controlla i parametri necessari, crea la classe TLSScanner ed esegue il metodo per far partire la scansione.

2. **TLSScanner** contiene la medesima classe *TLSScanner* la quale si occuperà di gestire la scansione ed i suoi risultati. Tutte le funzionalità sono implementate in questo file.

Congiuntamente a questi, vengono importati altri moduli:

- **requests** per richieste HTTP/HTTPS semplificate.
- **futures** per computazione parallela mediante processi.
- **asn1crypto** per il parsing dei certificati codificati in ASN.1.

- **pysocks** per l'utilizzo di proxy SOCKS5 necessari ad utilizzare il protocollo Tor.

Un ulteriore piccolo file chiamato *colors.py* contiene le funzionalità per poter stampare a schermo caratteri colorati.

6.4 Descrizione dello strumento

La schermata di aiuto mostra subito all'utilizzatore l'insieme di opzioni utilizzabili per avviare la scansione.

```
$ python2 tlsscan.py -h
usage: tlsscan.py <website> [options]

SSL/TLS website passive analyzer.

positional arguments:
  website               website to scan.

optional arguments:
  -h, --help            show this help message and exit
  -p PORT, --port PORT  TCP port to test (default: 443).
  --fullscan            start a full scan of the website.
  --ciphers             start a scan of server supported cipher suites.
  --suppproto           perform only a scan of supported protocols version.
  --certscan            perform only a scan of the server certificate.
  -d DELAY, --delay DELAY
                        wait N milliseconds between each request.
  -vv, --verbose        show verbose information.
  -w, --write           write scan output to file.
  -s, --sniff           save full packet capture in .pcap format. (NEED SUDO
                        PRIVILEGES)
  -t, --torify          make the script running under Tor network.
  -v, --version         show program version.
```

Vediamo adesso le funzionalità date da ogni opzione:

- **-p PORT, –port PORT** : permette di impostare manualmente il numero di porta da utilizzare.
- **–fullscan**: esegue la scansione del sito dato. È l'opzione più completa.
- **–ciphers**: esegue solamente la scansione delle cipher suite accettate dal server, ed ordinate per sua preferenza.
- **–supproto**: sta per "supported protocols", effettua solamente una scansione dei protocolli SSL/TLS supportati dal server.

- **–certscan**: effettua solamente una scansione dei protocolli supportati e stampa tutte le informazioni relative ai certificati SSl/TLS ricevuti.
- **-d DELAY, –delay DELAY**: permette di impostare un tempo espresso in millisecondi che il tool attenderà tra una richiesta e l'altra.
- **-vv, –verbose**: aggiunge alla stampa dettagli relativi ai certificati.
- **-w, –write**: permette di salvare il report finale su file.
- **-s, –sniff**: permette di salvare il full packet capture dello scan su file .pcap (non quando utilizzato insieme a -t)
- **-t, –torify**: permette di utilizzare la rete TOR per anonimizzare il traffico
- **-v, –version**: stampa la versione attuale del tool.

Per esempio:

```
$ python2 tlsscan.py www.archlinux.org --fullscan
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
|                                             |
|   ############  ###         #############   |
|   ############  ###         #############   | Scan a website and analyze
|       ###       ###         ###             | HTTPS configurations and
|       ###       ###         #############   | certificates.
|       ###       ###         #############   |
|       ###       ###                   ###   |
|       ###       ###                   ###   | Find misconfigurations
|       ###       ##########  #############   | which could lead to
|       ###       ##########  #############   | potential attacks.
|                                             |
| and SSL for HTTPS  Passive Security Scanner |
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

TARGET: www.archlinux.org resolved to 138.201.81.199:443
Date of the test: 2017-11-17 09:40:18.887950



Starting SSL/TLS test on www.archlinux.org --> 138.201.81.199:443
TYPE SCAN:  FULLSCAN


scanning for supported protocol...                    done. in -- 1.60s --
loading certificate chain...                          done. in -- 0.00s --
scanning for compression support...                   done. in -- 0.10s --
scanning for secure renegotiation extension..         done. in -- 1.19s --
ordering cipher suites based on server preference...  done. in -- 14.62s --
checking ocsp response..                              done. in -- 1.26s --
checking bad sni response...                          done. in -- 1.18s --
checking TLS heartbeat extension...                   done. in -- 1.18s --
checking TLS session ticket support..                 done. in -- 1.23s --
looking for HSTS header...                            done. in -- 0.24s --
```

Successivamente verranno stampati i risultati della scansione. Ne vedremo fra poco degli esempi.

6.5 Analisi del problema

6.5.1 Protocolli supportati

Per testare correttamente un sito web o più precisamente il web server che lo ospita in maniera passiva possiamo procedere nella maniera seguente.

Nel caso in cui un client tentasse di collegarsi richiedendo una versione del protocollo non supportata dal server allora esso invierà un ClientHello contenente all'interno del Handshake message la versione di SSL/TLS che vuole utilizzare. A questo punto il server può procedere con l'handshake inviando il suo ServerHello e successivi, altrimenti può rifiutare la connessione e procedere in tre diversi modi:

- **TLSAlert (Level: fatal, Description: Protocol Version**: il protocollo non è supportato dal server. Il server chiuderà la connessione inviando un pacchetto FIN.
- **TLSAlert (Level: fatal, Description: Inappropriate Fallback**: nel caso in cui il client abbia incluso la SCSV Signaling cipher suite (codice = 0x5600) e stia tentando di collegarsi con un protocollo inferiore a quello più alto supportato da entrambi. Il server riconoscerà un tentativo volontario di downgrade del protocollo e chiuderà la connessione inviando un pacchetto FIN.
- **TLSAlert (Level: fatal, Description: Handshake Failure**: nel caso in cui il server non riesca a negoziare dei parametri di sicurezza minimi e dunque è costretto a terminare la connessione.

A questo, punto procedendo con l'invio di particolari pacchetti Record Protocol contenenti le varie versioni di SSL/TLS con e senza SCSV ed analizzando le risposte, possiamo velocemente risalire ai protocolli supportati dal server.

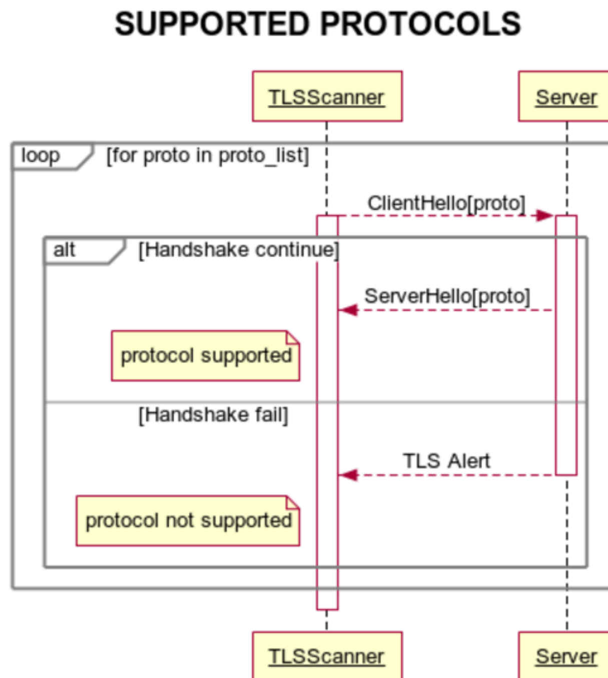


FIGURA 6.1: Diagramma di sequenza per la scansione dei protocolli supportati

6.5.2 TLS Extension e compressione

Per verificare che il server supporti la compressione a livello TLS il metodo è alquanto semplice.

Si procede inviando un ClientHello contenente nel campo *compression_methods* una lista di possibili metodi escluso il "null" con codice (0) indicante il non utilizzo della compressione. A questo punto si aspetta la risposta del server, se esso procederà con l'handshake allora indicherà il suo supporto per la compressione altrimenti risponderà con un TLSAlert **fatal, Decode Error**.

A questo punto dobbiamo verificare il supporto del server alle più importanti estensioni TLS come ad esempio: secure renegotiation, ocsp stapling, heartbeat, session ticket.

Il metodo è il medesimo. Includere l'estensione nel ClientHello, se il server risponderà in maniera positiva includendo egli stesso le estensioni indicate, allora supporterà tali estensioni.

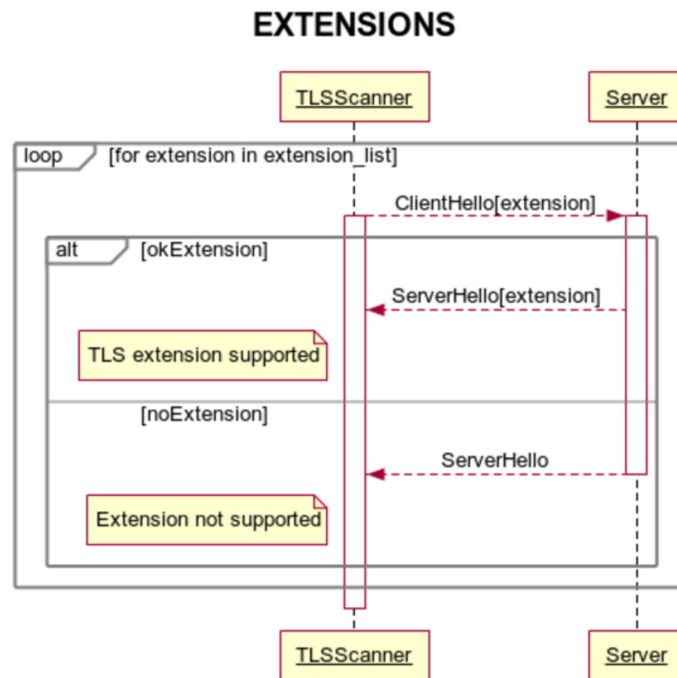


FIGURA 6.2: Diagramma di sequenza per la verifica del supporto delle estensioni TLS

6.5.3 HTTP Strict Transport Security

Per verificare il supporto del server a questa funzionalità si esegue una richiesta GET o HEAD (ho preferito HEAD) all'indirizzo indicato e si analizzano gli header in risposta.

Nel caso in cui il server supporti tale funzionalità esso includerà l'header:

Strict-Transport-Security:max-age=31536000; includeSubdomains; preload

I vari parametri come *max-age*, *includeSubdomains* e *preload* sono personalizzabili e gli ultimi due facoltativi.

6.5.4 Cipher suite

Per verificare le cipher suite disponibili dal server procediamo nella maniera seguente:

1. Presentiamo al server un ClientHello con versione di protocollo valida contenente l'intera lista di cipher suite disponibili.
2. Il server risponderà scegliendone una, la sua preferita.
3. Rimuoviamo tale cipher suite dalla lista e ripartiamo dal punto 1.

4. Alla fine avremo una lista ordinata di cipher suite supportate dal server, ordinate in base alla sua preferenza, per ogni protocollo supportato.

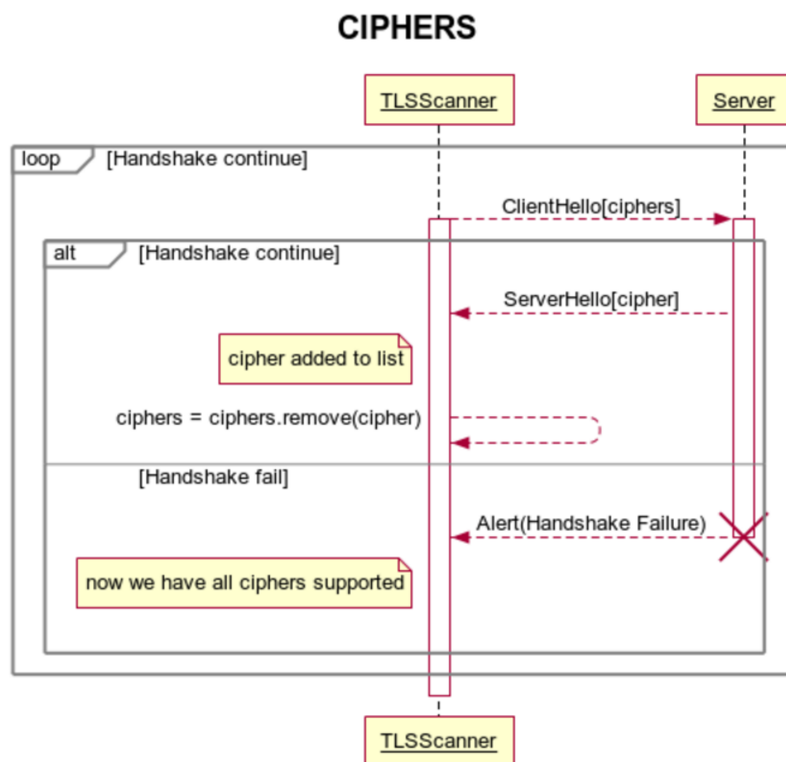


FIGURA 6.3: Diagramma di sequenza per la modalità CIPHERS

6.6 Implementazione

6.6.1 tlsscan

Il file *tlsscan* è il punto di partenza dell'applicazione, contiene i vari controlli sull'input, si occupa di creare l'oggetto TLSScanner e del salvataggio su file del report e del full packet capture.

- Il salvataggio del report è fatto con un semplice redirect dello stdout su file.
- Il salvataggio del full packet capture è reso possibile grazie alla funzionalità integrata in Scapy di nome **sniff**. Tale funzionalità affiancata alla funzione **wrpcap** e ad un filtro permette il salvataggio su file dei pacchetti circolati. Sono necessari i permessi di amministratore per operare a questo livello, velocemente verificati con *os.geteuid()*. Tale funzionalità opererà su un processo separato e parallelo.

Successivamente si creerà l'oggetto scanner con i parametri in input. Scelta la modalità di scansione la si farà partire richiamandola sull'oggetto con un *.scan(MODE)* dove MODE è la modalità scelta e disponibile tra *FULLSCAN*, *SUPPROTO*, *CIPHERS* e *CERTSCAN*.

A questo punto passiamo al file principale *TLSScanner*.

6.6.2 TLSScanner

In TLSScanner uno dei compiti importanti è svolto dalla funzione che invia al server il ClientHello message:

```
1  def send_client_hello(tls_scan_obj):
2          compression = range(1,0xff) if tls_scan_obj.tls_compression else 0x00
3          sock = TCPConnect(tls_scan_obj.target)
4          packet = TLSRecord(version=tls_scan_obj.outer_version)/\
5                  TLSHandshake()/\
6                  TLSClientHello(version=tls_scan_obj.version,
7                          compression_methods=compression,
8                          cipher_suites=tls_scan_obj.cipher_list,
9                          extensions = [
10                                 TLSExtension()/\
11                                 TLSExtServerNameIndication(server_names=
12                                 [TLSServerName(data=tls_scan_obj.server_name)])
13                         ])
14         if tls_scan_obj.tls_sec_reneg:
15                 packet.getlayer(TLSClientHello).extensions
16                         .append(TLSExtension()/TLSExtRenegotiationInfo())
17         if tls_scan_obj.tls_heartbeat:
18                 packet.getlayer(TLSClientHello).extensions
19                         .append(TLSExtension()/TLSExtHeartbeat())
20         if tls_scan_obj.ocsp:
21                 ocsp = TLSExtension(type="status_request", length=5)
22                 ocsp = str(ocsp).encode("hex") + "0100000000"
23                 ocsp = ocsp.decode("hex") #adding ocsp request manually
24                 packet.getlayer(TLSClientHello).extensions.append(SSL(ocsp))
25         if tls_scan_obj.session_ticket:
26                 packet.getlayer(TLSClientHello).extensions
27                         .append(TLSExtension()/TLSExtSessionTicketTLS())
28         sleep(float(tls_scan_obj.time_to_wait)/1000)
29         sock.sendall(str(packet))
30         resp = recvall(sock)
31         if tls_scan_obj.tls_heartbleed:
32                 p = TLSRecord(version=tls_scan_obj.version)/\
33                         TLSHeartBeat(length=2**14-1, data='bleeding...')
34                 sock.settimeout(1)
35                 sock.sendall(str(p))
36                 try:
37                         resp2 = sock.recv(8192)
38                 except (socket.timeout, socket.error) as msg:
39                         return (tls_scan_obj.version, "ERROR")
40                 sock.close()
41                 return (tls_scan_obj.version, resp2)
```

```
42        sock.close()
43        ssl_p = SSL(resp)
44        if ssl_p.haslayer(TLSServerHello) or ssl_p.haslayer(TLSAlert)
45                        or ssl_p.haslayer(TLSCertificate):
46                return (tls_scan_obj.version,resp)
47        else:
48                return None
```

Ogni metodo invocato dalla classe TLSScanner che necessiterà dell'invio di ClientHello si rifarà alla funzione sopra esposta. Ogni funzionalità verrà considerata supportata in base alla risposta del server. L'oggetto *tls_scan_obj* è un istanza di una classe contenente solamente i parametri necessari alla funzione per la personalizzazione del pacchetto.

Quando parte l'analisi delle cipher suite utilizzate dal server, essendo maggiore il numero di richieste necessarie ho optato per l'utilizzo del *ProcessPoolExecutor* il quale permette di parallelizzare effettivamente il lavoro su più processi o *workers*. Perché non i Thread? Perché in Python essi non sono pienamente *thread-safe* e dunque il GIL, *Global Interpreter Lock* [67], ne impedisce la totale concorrenza.

Se l'utente dovesse scegliere di utilizzare TOR per anonimizzare il traffico dello script, la porzione di codice è la seguente. Si assume che TOR sia installato sulla macchina locale ed in ascolto di nuove connessioni sulla porta 9050. TOR offre un proxy SOCKS5, dunque possiamo facilmente redirigere anche tutte le *requests* ad esso.

```
1  if not checkConnection(("127.0.0.1", 9050)):    #assuming a local TOR proxy
2          print "TOR␣PROXY␣NOT␣RUNNING\n"
3          exit(1)
4  else:
5          socks.set_default_proxy(socks.PROXY_TYPE_SOCKS5, "127.0.0.1", 9050)
6          print "TOR␣PROXY␣RUNNING␣ON␣PORT␣9050\n"
7          #setting global socket to use socks.socksocket
8          socket.socket = socks.socksocket
9          #socks5h for forcing DNS resolution through proxy
10         self._requests_session.proxies = {
11                 'http':'socks5h://127.0.0.1:9050',
12                 'https':'socks5h://127.0.0.1:9050'
13         }
```

Nota: anche le richieste DNS verranno dirottate sulla rete TOR non lasciando quindi trapelare nulla in chiaro.

Successivamente verrà stampato a schermo (o su file) il risultato della scansione.

6.7 Esempi di utilizzo

Prendiamo ora in considerazione degli esempi sui quali *tlsscan* ha operato. In questi esempi non è stata utilizzata l'opzione –verbose per non ingigantire troppo l'output mantenendo comunque visibili tutti i risultati importanti.

6.7.1 Esempio 1

Vediamo i risultati di una scansione effettuata su uno degli istituti di credito italiano più famosi, *www.*********.it*. (il numero di asterischi non riflette il numero di caratteri del nome di dominio)

```
#####  PRINTING RESULTS  ####

Total number of certificates received:  3

##### CERTIFICATE INFORMATION for www..*********.it #####

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2018-11-07 23:59:59
[*]Hostname match CN or SUBJECT_ALTERNATIVE_NAME? YES
(Requested)  www.*********.it  (Certificate) www.*********.it
[*]Hostname matches with alternative name:  www.*********.it
[*]Is a CA certificate? NO
[*]Is a self-signed certificate?  NO

##### CERTIFICATE INFORMATION for Symantec Class 3 Secure Server CA - G4 #####

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2023-10-30 23:59:59
(Requested)  www.*********.it
(Certificate) Symantec Class 3 Secure Server CA - G4
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  NO

#### CERTIFICATE INFORMATION for VeriSign Class 3
        Public Primary Certification Authority - G5 ####

Signature Algorithm:    sha1_rsa INSECURE
[*]Is certificate EXPIRED?  NO, valid until 2036-07-16 23:59:59
(Requested)  www.*********.it
(Certificate) VeriSign Class 3 Public Primary Certification Authority - G5
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  YES

################# PROTOCOLS SUPPORTED #################

SUPPORTED PROTOCOLS FOR HANDSHAKE:  TLS_1_2

################## CIPHER SUITES #################
```

```
Accepted cipher-suites ( 8 / 333 ) Ordered by server preference.

TLS_1_2 supports 8 cipher suites.

Protocol: TLS_1_2 -> RSA_WITH_AES_256_GCM_SHA384 (0x9d) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_GCM_SHA256 (0x9c) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA256 (0x3d) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA256 (0x3c) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) supported. non FS
[*]ALERT:  SHA could be upgraded to SHA-2
                        (even though used in case does not pose a real threat)
[*]ALERT:  DES/3DES considered weak, could disable.

        FS = Forward Secrecy (should use only cipher suite with it)


################# SECURITY OPTIONS #################

[*]TLS_FALLBACK_SCSV supported?  Unknown, only one protocol supported
[*]TLS COMPRESSION enabled?  False
[*]SECURE RENEGOTIATION supported? False
[*]TLS Heartbeat extension supported? No
[*]OCSP stapling supported? No
[*]HTTP Strict Transport Security enabled on https://www.*********.it ?  NO
        Server returned redirect header to:  https://www.*********.it/it.html
[*]Incorrect Server Name Indication alert?  NO
[*]TLS session tickets resumption?  NO

################# ATTACKS #################

[*]POODLE attack (SSLv3):  not vulnerable,
                SSLv3 disabled and/or TLS downgrade protection supported.
[*]BEAST attack?  NO
[*]EXPORT ciphers enabled?  NO
[*]Heartbleed vulnerable? NO
[*]RC4 supported?  NO

################# MISC #################

Request to https://www.*********.it/
Status code: 301 Moved Permanently
Server: BigIP
Connection: Keep-Alive

 ---------- SCAN FINISHED ----------

Finished in --- 52.3582379818 seconds ---
```

Come possiamo vedere nel report sono riportate informazioni riguardo la catena di certificati utilizzata (da notare che la ROOT CA Verisign utilizzi sha1_RSA come algoritmo di firma il quale, ad oggi è considerato insicuro), i

protocolli supportati, le cipher suite utilizzabili e vari parametri ed estensioni TLS supportati. Inoltre nella sezione finale sono riportate considerazioni sugli attacchi conosciuti e qualche informazione sullo stato della richiesta HTTPS effettuata verso il dominio selezionato.

Considerazioni

Riguardo a questo scan possiamo affermare che il server supporti unicamente TLS 1.2 come protocollo TLS, il che è valutato positivamente, ma d'altro canto utilizzi solamente cipher suite RSA che intrinsecamente non supportano la Forward Secrecy. Ciò rende tutte le comunicazioni con il server potenzialmente a rischio, nel qual caso un ipotetico attaccante riuscisse a scoprire il materiale crittografico utilizzato dal server. Un punto a sfavore è l'utilizzo di cipher suite con 3DES come algoritmo crittografico, il quale garantisce un basso livello di sicurezza in confronto a tutte le altre opzioni disponibili. Possiamo notare che il server non supporti la compressione TLS quindi nessun attacco del tipo di CRIME e TIME può avere successo. Inoltre non è supportata l'estensione Heartbeat e di conseguenza anche l'attacco Heartbleed è da principio evitato. Inoltre il server non supporta OCSP Stapling per la convalida dei certificati, ne utilizza HTTP Strict Transport Security sul link preso in oggetto (questo ha riportato codice 301 con redirect successivo). Ultima nota da considerare, seppur non sia responsabilità del sito preso in questione, è l'algoritmo di firma utilizzato dalla root CA *Verisign* che essendo *rsa_sha1* non è, ad oggi, considerata la miglior scelta in quanto sha1 è stato dimostrato vulnerabile.

Si noti che la profondità di questo scan non è considerevole e tiene conto solamente di aspetti passivi, ma permette di disegnare un buon quadro iniziale dello stato del server.

6.7.2 Esempio 2

Vediamo un'altro risultato ottenuto ora analizzando *www.*********.it*.

```
###########  PRINTING RESULTS  ###########

Total number of certificates received:  2

##### CERTIFICATE INFORMATION for www.*********.it #####
```

```
Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2018-07-30 23:59:59
[*]Hostname match CN or SUBJECT_ALTERNATIVE_NAME? YES
(Requested)  *********.it  (Certificate) www.*********.it
[*]Hostname matches with alternative name:  *********.it
[*]Is a CA certificate? NO
[*]Is a self-signed certificate?  NO
[*]CRL url:  http://sr.symcb.com/sr.crl
[*]OSCP url:  http://sr.symcd.com

#### CERTIFICATE INFORMATION for Symantec Class 3 EV SSL CA - G3 ###

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2023-10-30 23:59:59
(Requested)  *********.it  (Certificate) Symantec Class 3 EV SSL CA - G3
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  NO
[*]CRL url:  http://s1.symcb.com/pca3-g5.crl
[*]OSCP url:  http://s2.symcb.com

################# PROTOCOLS SUPPORTED #################

SUPPORTED PROTOCOLS FOR HANDSHAKE:  TLS_1_2

################## CIPHER SUITES #################

Accepted cipher-suites ( 6 / 333 ) Ordered by server preference.

TLS_1_2 supports 6 cipher suites.

Protocol: TLS_1_2 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA256 (0x3c) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA256 (0x3d) supported. non FS
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  DES/3DES considered weak, could disable.

        FS = Forward Secrecy (should use only cipher suite with it)

################## SECURITY OPTIONS #################

[*]TLS_FALLBACK_SCSV supported?  Unknown, only 1 protocol supported
[*]TLS COMPRESSION enabled?  False
[*]SECURE RENEGOTIATION supported? True
[*]TLS Heartbeat extension supported? No
[*]OCSP stapling supported? No
[*]HTTP Strict Transport Security enabled on https://*********.it ?  NO
[*]Incorrect Server Name Indication alert?  NO
[*]TLS session tickets resumption?  NO

################# ATTACKS #################

[*]POODLE attack (SSLv3):  not vulnerable,
                SSLv3 disabled and/or TLS downgrade protection supported.
```

```
[*]BEAST attack?  NO
[*]EXPORT ciphers enabled?  NO
[*]Heartbleed vulnerable? NO
[*]RC4 supported?  YES

Finished in --- 30.6416840553 seconds ---
```

Considerazioni

Possiamo notare anche qui che l'unico protocollo supportato sia TLS 1.2 oltre al fatto che le uniche cipher suite disponibili siano quelle che utilizzino RSA e quindi prive di Forward Secrecy. Un aspetto del tutto rilevante è l'utilizzo di RC4 come cifrario a flusso, notoriamente insicuro e quindi molto pericoloso. Ciò può rendere le comunicazioni con tale istituto di credito totalmente decifrabili da un ipotetico attaccante. Il supporto per OCSP Stapling e HSTS è mancante anche in questo caso.

6.7.3 Esempio 3

```
###########  PRINTING RESULTS  ###########

Total number of certificates received:  4

##### CERTIFICATE INFORMATION for www.*********.it ####

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2019-08-07 23:59:59
[*]Hostname match CN or SUBJECT_ALTERNATIVE_NAME? YES
[*]Hostname matches with alternative name:  www.*********.it
[*]Is a CA certificate? NO
[*]Is a self-signed certificate?  NO
[*]CRL url: http://crl.comodoca.com/COMODORSAExtendedValidationSecureServerCA.crl
[*]OSCP url: http://ocsp.comodoca.com

## CERTIFICATE INFORMATION for COMODO RSA Extended Validation Secure Server CA ##

Signature Algorithm:    sha384_rsa
[*]Is certificate EXPIRED?  NO, valid until 2027-02-11 23:59:59
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  NO
[*]CRL url: http://crl.comodoca.com/COMODORSACertificationAuthority.crl
[*]OSCP url: http://ocsp.comodoca.com

### CERTIFICATE INFORMATION for COMODO RSA Certification Authority ####

Signature Algorithm:    sha384_rsa
[*]Is certificate EXPIRED?  NO, valid until 2020-05-30 10:48:38
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  NO
```

```
[*]CRL url:  http://crl.usertrust.com/AddTrustExternalCARoot.crl
[*]OSCP url:  http://ocsp.usertrust.com

### CERTIFICATE INFORMATION for AddTrust External CA Root ###

Signature Algorithm:    sha1_rsa INSECURE
[*]Certificate signature scheme:  rsassa_pkcs1v15
[*]Is certificate EXPIRED?  NO, valid until 2020-05-30 10:48:38
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  YES
[*]CRL url:  NO CLR
[*]OSCP url:  NO OCSP

################# PROTOCOLS SUPPORTED #################

SUPPORTED PROTOCOLS FOR HANDSHAKE:  TLS_1_2

################## CIPHER SUITES #################

Accepted cipher-suites ( 21 / 333 ) Ordered by server preference.

TLS_1_2 supports 21 cipher suites.

Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_256_GCM_SHA384 (0x9f) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_128_GCM_SHA256 (0x9e) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_256_CBC_SHA256 (0x6b) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_128_CBC_SHA256 (0x67) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_GCM_SHA384 (0x9d) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_GCM_SHA256 (0x9c) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA256 (0x3d) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA256 (0x3c) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_256_GCM_SHA384 (0xc030) supported. FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f) supported. FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_256_CBC_SHA384 (0xc028) supported. FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_128_CBC_SHA256 (0xc027) supported. FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012) supported. FS
[*]ALERT:  SHA could be upgraded to SHA-2
                        (even though used in case does not pose a real threat)
[*]ALERT:  DES/3DES considered weak, could disable.

        FS = Forward Secrecy (should use only cipher suite with it)

################## SECURITY OPTIONS #################

[*]TLS_FALLBACK_SCSV supported?  Unknown, only 1 protocol supported
[*]TLS COMPRESSION enabled?  False
[*]SECURE RENEGOTIATION supported? True
[*]TLS Heartbeat extension supported? No
[*]OCSP stapling supported? No
[*]HTTP Strict Transport Security enabled on https://www.*********.it ?  NO
```

```
[*]Incorrect Server Name Indication alert?  NO
[*]TLS session tickets resumption?  NO

################# ATTACKS #################

[*]POODLE attack (SSLv3):  not vulnerable,
                        SSLv3 disabled and/or TLS downgrade protection supported.
[*]BEAST attack?  NO
[*]EXPORT ciphers enabled?  NO
[*]Heartbleed vulnerable? NO
[*]RC4 supported?  NO

Finished in --- 79.4522228241 seconds ---
```

Considerazioni

Qui possiamo notare che oltre a non utilizzare HSTS ed OCSP Stapling *********.*it* permette l'utilizzo di cipher suite con FS, supporta la Secure Renegotiation ed utilizza esclusivamente TLS 1.2.

6.7.4 Esempio 4

Vediamo come si comporta una banca di minore dimensione a livello nazionale, *********.*it*.

```
#####  PRINTING RESULTS  #####

Total number of certificates received:  2

#### CERTIFICATE INFORMATION for *.*********.it #####

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2018-09-10 23:59:59
[*]Hostname match CN or SUBJECT_ALTERNATIVE_NAME? NO
(Requested)  www.*********.it  (Certificate) *.*********.it
[*]Hostname matches with alternative name:  Nothing
[*]Is a CA certificate? NO
[*]Is a self-signed certificate?  NO

#### CERTIFICATE INFORMATION for RapidSSL SHA256 CA ####

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2022-05-20 23:45:51
(Requested)  www.*********.it  (Certificate) RapidSSL SHA256 CA
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  NO

################# PROTOCOLS SUPPORTED #################

SUPPORTED PROTOCOLS FOR HANDSHAKE:  TLS_1_2 TLS_1_1 TLS_1_0 SSL_3_0

################# CIPHER SUITES #################
```

```
Accepted cipher-suites ( 61 / 333 ) Ordered by server preference.

SSL_3_0 supports 19 cipher suites.

Protocol: SSL_3_0 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
Protocol: SSL_3_0 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: SSL_3_0 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: SSL_3_0 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x88) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_SEED_CBC_SHA (0x96) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_SEED_CBC_SHA (0x9a) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_RC4_128_SHA (0xc011) supported. FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  CBC and SSL_3 are vulnerable to POODLE attack!
[*]ALERT:  SSL_3_0 and CBC could lead to BEAST attack!
                (if not client mitigated)
[*]ALERT:  DES/3DES considered weak, could disable.

TLS_1_1 supports 14 cipher suites.

Protocol: TLS_1_1 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
Protocol: TLS_1_1 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: TLS_1_1 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: TLS_1_1 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: TLS_1_1 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: TLS_1_1 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) supported. non FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  DES/3DES considered weak, could disable.

TLS_1_0 supports 14 cipher suites.

Protocol: TLS_1_0 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
```

```
Protocol: TLS_1_0 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: TLS_1_0 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: TLS_1_0 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: TLS_1_0 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: TLS_1_0 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) supported. non FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  TLS_1_0 and CBC could lead to BEAST attack!
                (if not client mitigated)
[*]ALERT:  DES/3DES considered weak, could disable.

TLS_1_2 supports 14 cipher suites.

Protocol: TLS_1_2 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA256 (0x3c) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA256 (0x3d) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_128_CBC_SHA256 (0x67) supported. FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  DES/3DES considered weak, could disable.

        FS = Forward Secrecy (should use only cipher suite with it)

################## SECURITY OPTIONS #################

[*]TLS_FALLBACK_SCSV supported?  True
[*]TLS COMPRESSION enabled?  False
[*]SECURE RENEGOTIATION supported? True
[*]TLS Heartbeat extension supported? Yes
[*]OCSP stapling supported? No
[*]HTTP Strict Transport Security enabled on https://www.*********.it ?  NO
[*]Incorrect Server Name Indication alert?  NO
[*]TLS session tickets resumption?  YES

################## ATTACKS #################
```

```
[*]POODLE attack (SSLv3):  Vulnerable!
[*]BEAST attack?  not mitigated server side.
[*]EXPORT ciphers enabled?  NO
[*]Heartbleed vulnerable? NO
[*]RC4 supported?  YES

################# MISC #################

Request to https://www.*********.it/
Status code: 200 OK
Server: Apache
Connection: Keep-Alive

 ---------- SCAN FINISHED ----------

Finished in --- 35.7183520794 seconds ---
```

Considerazioni

Possiamo notare che qui il panorama offerto non è dei più rosei.

Innanzitutto vediamo che il server web accetta quasi tutti i protocolli SSL/-TLS ovvero: TLS_1_2, TLS_1_1, TLS_1_0 ma ancora più grave, **SSL_3_0**. Inoltre cipher suite che utilizzano **RC4** e **CBC** come modalità di cifratura sono presenti, il che aggrava molto la situazione, rendendo potenzialmente realizzabile l'attacco **POODLE**!. Oltre a non supportare HTTP Strict Transport Security non è offerto nemmeno OCSP Stapling. Infine il server supporta un grande numero di cipher suite, cosa non raccomandabile, ma degne di nota sono le cipher suite che utilizzino ancora **MD5** come algoritmo di hashing/MAC.

6.7.5 Esempio 5

```
###########  PRINTING RESULTS  ###########

Total number of certificates received:  2

##### CERTIFICATE INFORMATION for www.*********.it ####

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2018-03-30 23:59:59
[*]Hostname match CN or SUBJECT_ALTERNATIVE_NAME? YES
(Requested)  www.*********.it  (Certificate) www.*********.it
[*]Hostname matches with alternative name:  www.*********.it
[*]Is a CA certificate? NO
[*]Is a self-signed certificate?  NO
```

```
#### CERTIFICATE INFORMATION for GeoTrust SSL CA - G3 ####

Signature Algorithm:    sha256_rsa
[*]Is certificate EXPIRED?  NO, valid until 2022-05-20 21:36:50
(Requested)  www.*********.it  (Certificate) GeoTrust SSL CA - G3
[*]Is a CA certificate? YES
[*]Is a self-signed certificate?  NO

################# PROTOCOLS SUPPORTED #################

SUPPORTED PROTOCOLS FOR HANDSHAKE:  TLS_1_2 TLS_1_1 TLS_1_0 SSL_3_0

################## CIPHER SUITES #################

Accepted cipher-suites ( 71 / 333 ) Ordered by server preference.

SSL_3_0 supports 19 cipher suites.

Protocol: SSL_3_0 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
Protocol: SSL_3_0 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: SSL_3_0 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: SSL_3_0 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x88) supported. FS
Protocol: SSL_3_0 -> RSA_WITH_SEED_CBC_SHA (0x96) supported. non FS
Protocol: SSL_3_0 -> DHE_RSA_WITH_SEED_CBC_SHA (0x9a) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012) supported. FS
Protocol: SSL_3_0 -> ECDHE_RSA_WITH_RC4_128_SHA (0xc011) supported. FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  CBC and SSL_3 are vulnerable to POODLE attack!
[*]ALERT:  SSL_3_0 and CBC could lead to BEAST attack!
                (if not client mitigated)
[*]ALERT:  DES/3DES considered weak, could disable.

TLS_1_1 supports 17 cipher suites.

Protocol: TLS_1_1 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
Protocol: TLS_1_1 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: TLS_1_1 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: TLS_1_1 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
```

```
Protocol: TLS_1_1 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: TLS_1_1 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: TLS_1_1 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x88) supported. FS
Protocol: TLS_1_1 -> RSA_WITH_SEED_CBC_SHA (0x96) supported. non FS
Protocol: TLS_1_1 -> DHE_RSA_WITH_SEED_CBC_SHA (0x9a) supported. FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  DES/3DES considered weak, could disable.

TLS_1_0 supports 18 cipher suites.

Protocol: TLS_1_0 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
Protocol: TLS_1_0 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: TLS_1_0 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: TLS_1_0 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: TLS_1_0 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: TLS_1_0 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_CAMELLIA_256_CBC_SHA (0x84) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_CAMELLIA_256_CBC_SHA (0x88) supported. FS
Protocol: TLS_1_0 -> RSA_WITH_SEED_CBC_SHA (0x96) supported. non FS
Protocol: TLS_1_0 -> DHE_RSA_WITH_SEED_CBC_SHA (0x9a) supported. FS
Protocol: TLS_1_0 -> ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012) supported. FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  TLS_1_0 and CBC could lead to BEAST attack!
                (if not client mitigated)
[*]ALERT:  DES/3DES considered weak, could disable.

TLS_1_2 supports 17 cipher suites.

Protocol: TLS_1_2 -> RSA_WITH_RC4_128_MD5 (0x4) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_RC4_128_SHA (0x5) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_IDEA_CBC_SHA (0x7) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_3DES_EDE_CBC_SHA (0xa) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x16) supported. FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA (0x2f) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_128_CBC_SHA (0x33) supported. FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA (0x35) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_256_CBC_SHA (0x39) supported. FS
Protocol: TLS_1_2 -> RSA_WITH_AES_128_CBC_SHA256 (0x3c) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_AES_256_CBC_SHA256 (0x3d) supported. non FS
Protocol: TLS_1_2 -> RSA_WITH_CAMELLIA_128_CBC_SHA (0x41) supported. non FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_CAMELLIA_128_CBC_SHA (0x45) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_128_CBC_SHA256 (0x67) supported. FS
Protocol: TLS_1_2 -> DHE_RSA_WITH_AES_256_CBC_SHA256 (0x6b) supported. FS
```

```
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013) supported. FS
Protocol: TLS_1_2 -> ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014) supported. FS
[*]ALERT:  cipher (4): MD5 is deprecated and considered insecure
[*]ALERT:  RC4 in known to be insecure and deprecated
[*]ALERT:  SHA could be upgraded to SHA-2
                (even though used in case does not pose a real threat)
[*]ALERT:  DES/3DES considered weak, could disable.

        FS = Forward Secrecy (should use only cipher suite with it)

################## SECURITY OPTIONS #################

[*]TLS_FALLBACK_SCSV supported?  True
[*]TLS COMPRESSION enabled?  False
[*]SECURE RENEGOTIATION supported? True
[*]TLS Heartbeat extension supported? Yes
[*]OCSP stapling supported? No
[*]HTTP Strict Transport Security enabled on https://www.*********.it ?  NO
[*]Incorrect Server Name Indication alert?  NO
[*]TLS session tickets resumption?  YES

################## ATTACKS #################

[*]POODLE attack (SSLv3):  Vulnerable!
[*]BEAST attack?  not mitigated server side.
[*]EXPORT ciphers enabled?  NO
[*]Heartbleed vulnerable? NO
[*]RC4 supported?  YES

################## MISC ##################

Request to https://www.*********.it/
Status code: 200 OK
Server: Apache/2.2.15 (CentOS)
Connection: Keep-Alive

 ---------- SCAN FINISHED ----------

Finished in --- 108.927115917 seconds ---
```

Considerazioni

Anche su questo istituto di credito possiamo vedere che la situazione è molto simile se non identica a quella dell'esempio precedente.

6.8 Valutazioni finali

Dopo aver effettuato una scansione sulla sicurezza delle comunicazioni di molti istituti di credito italiano, oltre a quelli sopra mostrati, è possibile affermare che, mediamente, il panorama offerto non è tale da permettere particolari elogi e congratulazioni ai gestori e manutentori dei vari server Web. Vorrei ricordare che stiamo parlando di siti Web riguardanti banche, dunque uno dei servizi nei quali riponiamo la nostra fiducia ed i nostri soldi, uno dei servizi che dovrebbe garantire i massimi standard di sicurezza disponibili. Il fatto sconcertante è che tali configurazioni, ad oggi (2017), sono semplificate a tal punto che un tecnico con un minimo di conoscenza sull'argomento, sarebbe in grado di rendere "sicuro" un server web in molto meno tempo di quanto ci si aspetterebbe.

Conclusioni

Con *tlsscan* è possibile avere in pochi secondi un utile strumento di verifica delle impostazioni fondamentali per quanto riguarda un server Web abilitato ad HTTPS. Come abbiamo visto, tlsscan permette di avere un immagine iniziale chiara dei possibili problemi a cui un server Web mal configurato potrebbe andare incontro. Dai risultati di una scansione è inoltre possibile agire in maniera mirata per correggere tali errori in un breve lasso di tempo. In questa prima versione, il tool non comprende un'analisi di tutti i parametri di sicurezza ad oggi conosciuti, cosa che auspico avverrà in un prossimo futuro. Molte sono le opzioni migliorabili, ad esempio l'integrazione con TOR può essere perfezionata, ed eventualmente può essere aggiunto un formato di salvataggio del report facilmente analizzabile per un numero elevato di scansioni. Questo potrebbe introdurre uno strumento di analisi su larga scala di tali configurazioni. In un secondo momento, avendo a disposizione una grande quantità di dati, diventerà poi semplice stilare delle percentuali raffigurative dell'attuale livello di sicurezza dei server Web sparsi in tutto il mondo.

Ringraziamenti

Vorrei ringraziare il professor Gabriele D'Angelo che mi ha seguito, corretto ed indirizzato durante la stesura della tesi, e tutti coloro che mi hanno incoraggiato e stimolato in questo periodo, primi fra tutti la mia famiglia che mi ha dato la possibilità di arrivare sin qui.

Grazie davvero,

Carlo Alberto

Bibliografia

[1] Ivan Ristić. *Bulletproof SSL and TLS, Understanding and deploying SSL/-TLS and PKI to secure servers and web applications.* A cura di Feisty Duck. 2015.

[2] RFC 4492. *Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. URL: https://www.rfc-editor.org/rfc/pdfrfc/rfc4492.txt.pdf.

[3] acunetix. *CRIME SSL/TLS attack*. URL: https://www.acunetix.com/vulnerabilities/web/crime-ssl-tls-attack.

[4] Nadhem AlFardan e Kenny Paterson. *Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*. URL: http://www.isg.rhul.ac.uk/tls/Lucky13.html.

[5] Arstechnica. *Critics slam SSL authority for minting certificate for impersonating sites*. URL: https://arstechnica.com/information-technology/2012/02/critics-slam-ssl-authority-for-minting-cert-used-to-impersonate-sites/.

[6] Arstechnica. *HTTPS Certificate Revocation is broken, and it's time for some new tools*. 2017. URL: https://arstechnica.com/information-technology/2017/07/https-certificate-revocation-is-broken-and-its-time-for-some-new-tools/.

[7] arstechnica. *"Lucky Thirteen" attack snarfs cookies protected by SSL encryption*. URL: https://arstechnica.com/information-technology/2013/02/lucky-thirteen-attack-snarfs-cookies-protected-by-ssl-encryption/.

[8] Zineb Ait Bahajji e Gary Illyes. *HTTPS as a ranking signal*. 2014. URL: https://security.googleblog.com/2014/08/https-as-ranking-signal_6.html.

[9] Tal Be'ery. *A Perfect CRIME? A Perfect CRIME?Only TIME Will Only TIME Will Tell*. URL: https://media.blackhat.com/eu-13/briefings/Beery/bh-eu-13-a-perfect-crime-beery-wp.pdf.

[10] blog.cloudfare. *Introducing TLS 1.3*. URL: https://blog.cloudflare.com/introducing-tls-1-3/.

[11] blog.cloudfare. *Introducing Zero Round Trip Time Resumption (0-RTT)*. URL: https://blog.cloudflare.com/introducing-0-rtt/.

[12] blog.cloudflare. *Padding oracles and the decline of CBC-mode cipher suites*. URL: https://blog.cloudflare.com/padding-oracles-and-the-decline-of-cbc-mode-ciphersuites/.

[13] blog.qualys. *Is BEAST Still a Threat?* URL: https://blog.qualys.com/ssllabs/2013/09/10/is-beast-still-a-threat.

[14] blog.qualys. *Is HTTP Public Key Pinning Dead?* URL: https://blog.qualys.com/ssllabs/2016/09/06/is-http-public-key-pinning-dead.

[15] blog.qualys. *SSL and TLS Authentication Gap vulnerability discovered*. URL: https://blog.qualys.com/ssllabs/2009/11/05/ssl-and-tls-authentication-gap-vulnerability-discovered.

[16] Krzysztof Kotowicz Bodo Möller Thai Duong. *This POODLE Bites: Exploiting The SSL 3.0 Fallback*. URL: https://www.openssl.org/~bodo/ssl-poodle.pdf.

[17] bug.mozilla. *Investigate incident with CA that allegedly issued bogus cert for www.mozilla.com*. URL: https://bugzilla.mozilla.org/show_bug.cgi?id=470897.

[18] Censys. *The FREAK Attack*. URL: https://censys.io/blog/freak.

[19] Dev Chromium. *CRLSets*. URL: https://dev.chromium.org/Home/chromium-security/crlsets.

[20] csrc.nist.gov. *Suite B Cryptography Suite B Cryptography*. URL: https://csrc.nist.gov/CSRC/media/Events/ISPAB-MARCH-2006-MEETING/documents/E_Barker-March2006-ISPAB.pdf.

[21] Marcus Pinto Dafydd Stuttard. *The Web Application Hacker's Handbook, Finding and Exploiting Security Flaws, 2 edition*. A cura di Inc Wiley Publishing. 2011.

[22] Zakir Durumeric Pierrick Gaudry Matthew Green J. Alex Halderman Nadia Heninger Drew Springall Emmanuel Thomé Luke Valenta Benjamin VanderSloot Eric Wustrow Santiago Zanella-Béguelin David Adrian Karthikeyan Bhargavan e Paul Zimmermann. *Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice*. URL: https://weakdh.org/imperfect-forward-secrecy-ccs15.pdf.

[23] Zakir Durumeric Pierrick Gaudry Matthew Green J. Alex Halderman Nadia Heninger Drew Springall Emmanuel Thomé Luke Valenta Benjamin VanderSloot Eric Wustrow Santiago Zanella-Béguelin David Adrian Karthikeyan Bhargavan e Paul Zimmermann. *Weak Diffie-Hellman and the Logjam Attack*. URL: https://weakdh.org/.

[24] developer.mozilla. *HTTP Public Key Pinning (HPKP)*. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Public_Key_Pinning.

[25] Electronic Frontier Foundation. *The EFF SSL Observatory*. URL: https://www.eff.org/it/observatory.

[26] Mozilla Foundations. URL: https://developer.mozilla.org.

[27] Github. *Scapy-SSL/TLS*. URL: https://github.com/tintinweb/scapy-ssl_tls.

[28] Matthew Green. *Attack of the week: FREAK (or 'factoring the NSA for fun and profit')*. URL: https://blog.cryptographyengineering.com/2015/03/03/attack-of-week-freak-or-factoring-nsa/.

[29] Matthew Green. *Attack of the week: RC4 is kind of broken in TLS*. URL: https://blog.cryptographyengineering.com/2013/03/12/attack-of-week-rc4-is-kind-of-broken-in/.

[30] Kelly Jackson Higgins. *FREAK Out: Yet Another New SSL/TLS Bug Found*. URL: https://www.darkreading.com/attacks-breaches/freak-out-yet-another-new-ssl-tls-bug-found/d/d-id/1319320.

[31] ibm. *"A technical view of theOpenSSL 'Heartbleed' vulnerability A look at the memory leak in the OpenSSL Heartbeat*. URL: https://www.ibm.com/developerworks/community/files/form/anonymous/api/library/38218957-7195-4fe9-812a-10b7869e4a87/document/ab12b05b-9f07-4146-8514-18e22bd5408c/media.

[32] ietf. *Deprecating Secure Sockets Layer Version 3.0*. URL: https://tools.ietf.org/html/rfc7568.

[33] ietf. *RFC-6125 Representation and Verification of Domain-Based Application Service Identity within Internet Public Key Infrastructure Using X.509 (PKIX) Certificates in the Context of Transport Layer Security (TLS)*. URL: https://tools.ietf.org/html/rfc6125.

[34] ietf. *The TLS Protocol Version 1.0*. URL: https://tools.ietf.org/html/rfc2246.

[35] ietf. *The Transport Layer Security (TLS) Protocol Version 1.2*. URL: https://tools.ietf.org/html/rfc5246#section-6.2.

[36] ietf. *TLS Fallback Signaling Cipher Suite Value (SCSV) for Preventing Protocol Downgrade Attacks*. URL: https://tools.ietf.org/html/rfc7507.

[37] ietf. *Transport Layer Security (TLS) Renegotiation Indication Extension*. URL: https://tools.ietf.org/html/rfc5746.

[38] ietf. *Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension*. URL: https://tools.ietf.org/html/rfc7627.

[39] ietf. *Transport Layer Security (TLS) Session Resumption without Server-Side State*. URL: https://www.ietf.org/rfc/rfc5077.txt.

[40] Internet Engineering Task Force (IETF). *The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA*. URL: https://tools.ietf.org/html/rfc6698.

[41] Cedric Fournet Alfredo Pironti Karthikeyan Bhargavan Antoine Delignat-Lavaud e Pierre-Yves Strub. *Triple Handshakes Considered Harmful: Breaking and Fixing Authentication over TLS*. URL: https://www.mitls.org/pages/attacks/3SHAKE.

[42] learncryptography. *The Birthday Problem*. URL: https://learncryptography.com/mathematics/the-birthday-problem.

[43] *Let's Encrypt*. 2014. URL: https://letsencrypt.org/.

[44] David A. McGrew e John Viega. *The Security and Performance of the Galois/Counter Mode (GCM) of Operation*. URL: https://eprint.iacr.org/2004/193.pdf.

[45] Technet Microsoft. *Securing PKI: Technical Controls for Securing PKI*. URL: https://technet.microsoft.com/en-us/library/dn786426(v=ws.11).aspx.

[46] msdn.microsoft. *Cipher Suites in TLS/SSL (Schannel SSP)*. URL: https://msdn.microsoft.com/it-it/library/windows/desktop/aa374757(v=vs.85).aspx.

[47] Kenny Paterson Bertram Poettering Nadhem AlFardan Dan Bernstein e Jacob Schuldt. *On the Security of RC4 in TLS and WPA*. URL: http://www.isg.rhul.ac.uk/tls/.

[48] NIST. *DES MODES OF OPERATION*. URL: https://csrc.nist.gov/CSRC/media/Publications/fips/81/archive/1980-12-02/documents/fips81.pdf.

[49] OWASP. URL: https://www.owasp.org.

[50] Public-Key Infrastructure (X.509) (pkix). URL: https://datatracker.ietf.org/wg/pkix/charter/.

[51] Qualys. *Qualys SSL Labs*. URL: https://www.ssllabs.com/index.html.

[52] Qualys. *Qualys SSL Pulse Scan*. URL: https://www.ssllabs.com/ssl-pulse/.

[53] Rijksoverheid. *Black Tulip Update*. URL: https://www.rijksoverheid.nl/documenten/rapporten/2012/08/13/black-tulip-update.

[54] Ivan Ristić. *Internet SSL Survey 2010 is here!* URL: https://blog.ivanristic.com/2010/07/internet-ssl-survey-2010-is-here.html.

[55] Sumita Gupta Rounak Sinha Hemant Kumar Srivastava. *Performance Based Comparison Study of RSA and Elliptic Curve Cryptography*. URL: https://www.ijser.org/researchpaper/Performance-Based-Comparison-Study-of-RSA-and-Elliptic-Curve-Cryptography.pdf.

[56] security.googleblog. *Announcing the first SHA1 collision*. URL: https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html.

[57] Simon Singh. *Codici & Segreti, La storia affascinante dei messaggi cifrati dall'antico Egitto a Internet*. A cura di BUR Rizzoli Saggi. 2011.

[58] Dave Aitel Susan Young. *The Hacker's Handbook, The Strategy behind Breaking into and Defending Networks*. A cura di CRC Press LLC. 2004.

[59] Chrome Team. *Certificate Transparency in Chrome - Change to Enforcement Date*. URL: https://security.googleblog.com/2016/10/distrusting-wosign-and-startcom.html.

[60] techopedia. *Attack Vector*. URL: https://www.techopedia.com/definition/15793/attack-vector.

[61] Thawte. *How certificate chains work*. URL: https://search.thawte.com/support/ssl-digital-certificates/index?page=content&actp=CROSSLINK&id=SO16297.

[62] US-CERT. *Alert (TA14-290A) SSL 3.0 Protocol Vulnerability and POODLE Attack*. URL: https://www.us-cert.gov/ncas/alerts/TA14-290A.

[63] Mathy Vanhoef e Frank Piessens. *RC4 NOMORE Numerous Occurrence MOnitoring & Recovery Exploit*. URL: https://www.rc4nomore.com/.

[64] WhiteHatSecurity. *WEB APPLICATIONS SECURITY STATISTICS REPORT*. 2015. URL: https://info.whitehatsec.com/rs/675-YBI-674/images/WH-2016-Stats-Report-FINAL.pdf.

[65] Wikipedia. *Entropy as a measure of password strength*. URL: https://en.wikipedia.org/wiki/Password_strength#Entropy_as_a_measure_of_password_strength.

[66] Wikipedia. *Wikipedia*. URL: https://www.wikipedia.org/.

[67] wiki.python.org. *GlobalInterpreterLock*. URL: https://wiki.python.org/moin/GlobalInterpreterLock.

[68] zdnet. *SSL broken! Hackers create rogue CA certificate using MD5 collisions*. URL: http://www.zdnet.com/article/ssl-broken-hackers-create-rogue-ca-certificate-using-md5-collisions/.