

Appunti su Attacchi TLS: BEAST e CRIME

29 novembre 2025

Indice

1 L'attacco BEAST (Browser Exploit Against SSL/TLS)	2
1.1 Contesto: Modalità CBC e Initialization Vector (IV)	2
1.2 La Vulnerabilità: Chained IV in TLS 1.0	3
1.2.1 Formalizzazione Matematica	3
1.2.2 La Radice del Problema: Stream vs Record	3
1.3 La Meccanica dell'Attacco BEAST	4
1.4 Soluzione e Mitigazione	6
1.5 Rendere Pratico l'Attacco: Chosen Boundary Attack	7
1.5.1 Chosen Boundary Attack	7
1.5.2 Analisi della Complessità	8
2 L'attacco CRIME (Compression Ratio Info-leak Made Easy)	9
2.1 Analisi Formale: Violazione della IND-CPA	9
2.2 Dettagli Tecnici: DEFLATE e LZ77	10
2.3 Esecuzione Pratica: L'Attacco al Cookie	10

1 L'attacco BEAST (Browser Exploit Against SSL/TLS)

L'attacco BEAST (*Browser Exploit Against SSL/TLS*), presentato da T. Duong e J. Rizzo nel 2011, sfrutta una vulnerabilità nota (già teorizzata da Rogaway nel 1999 e Moller nel 2004) nel protocollo TLS v1.0. Nello specifico, l'attacco colpisce l'uso di un **Initialization Vector (IV)** prevedibile nella modalità di cifratura *Cipher Block Chaining* (CBC).

1.1 Contesto: Modalità CBC e Initialization Vector (IV)

La modalità CBC è uno standard per i cifrari a blocchi. Per crittografare un messaggio m , suddiviso nei blocchi m_1, m_2, \dots, m_n , la formula generica è:

$$C_i = E_K(m_i \oplus C_{i-1})$$

Dove E_K è la funzione di cifratura con chiave K e C_i è il blocco di testo cifrato attuale.

Per il primo blocco (m_1), non esiste un blocco cifrato precedente (C_0). Si utilizza quindi un **Initialization Vector (IV)**:

$$C_1 = E_K(m_1 \oplus IV)$$

Per garantire la sicurezza semantica (ovvero, impedire che un attaccante possa dedurre informazioni dai testi cifrati anche vedendo crittografato lo stesso messaggio più volte), l'IV deve soddisfare due proprietà:

1. **Unicità:** L'IV deve essere diverso per ogni messaggio crittografato con la stessa chiave. Altrimenti, due messaggi identici produrrebbero lo stesso testo cifrato.
2. **Imprevedibilità:** Un attaccante non deve poter prevedere quale sarà l'IV del prossimo messaggio.

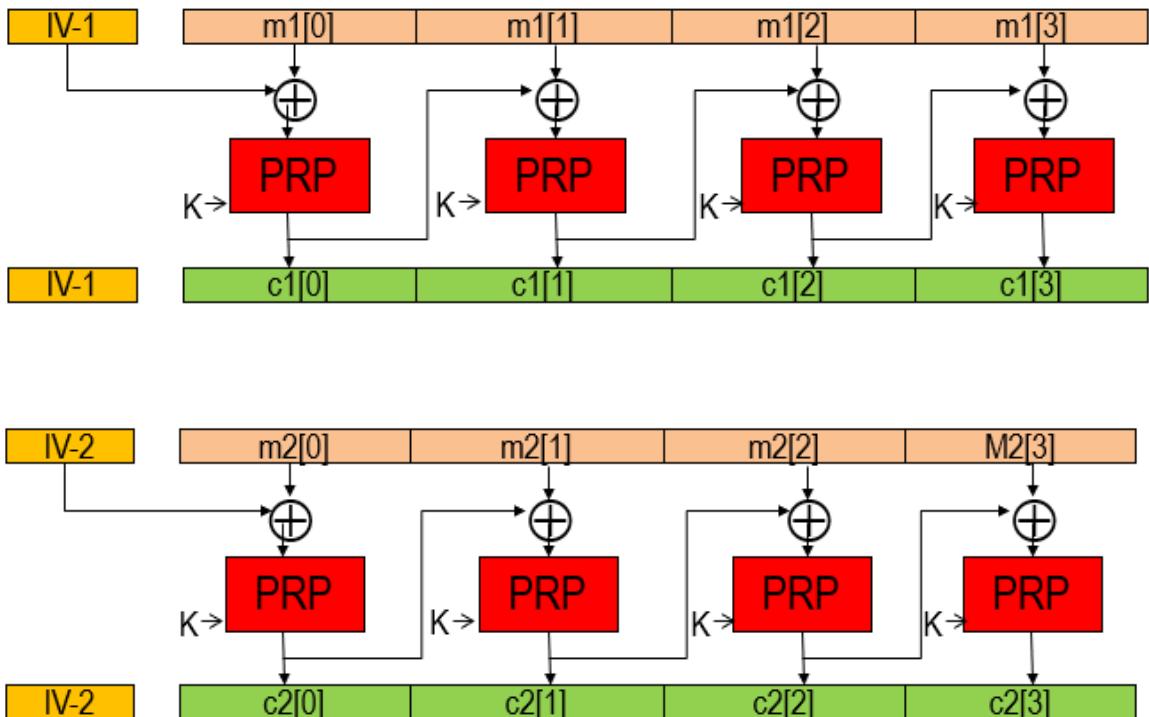


Figura 1: Schema della modalità CBC

1.2 La Vulnerabilità: Chained IV in TLS 1.0

Il protocollo TLS 1.0 presenta una criticità fondamentale nella gestione della modalità CBC: non garantisce la proprietà di imprevedibilità dell'Initialization Vector (IV). In questa versione, infatti, l'IV per un nuovo messaggio (o *record*) non viene generato casualmente, ma è definito come **l'ultimo blocco di testo cifrato del messaggio precedente**:

$$IV_{\text{nuovo_messaggio}} = C_{\text{ultimo_blocco_precedente}}$$

Questa tecnica è nota come **Chained IV**.

1.2.1 Formalizzazione Matematica

Consideriamo la trasmissione di dati suddivisa in due record distinti, M_1 e M_2 :

- Il primo record M_1 è composto dai blocchi $m_{1,1}, m_{1,2}, \dots, m_{1,n}$. La sua cifratura produce la sequenza di testi cifrati $C_{1,1}, \dots, C_{1,n}$.
- Il secondo record M_2 inizia con i blocchi $m_{2,1}, m_{2,2}, \dots, m_{2,n}$.

In TLS 1.0, l'Initialization Vector utilizzato per cifrare il primo blocco del secondo messaggio ($m_{2,1}$) è esattamente l'ultimo blocco cifrato del primo messaggio ($C_{1,n}$):

$$IV_{M_2} = C_{1,n}$$

Di conseguenza, la cifratura del primo blocco del nuovo messaggio diventa:

$$C_{2,1} = E_K(m_{2,1} \oplus C_{1,n})$$

Poiché $C_{1,n}$ è trasmesso in chiaro sulla rete ed è noto a tutti (incluso un attaccante Man-In-The-Middle) *prima* che $m_{2,1}$ venga generato, l'IV del nuovo messaggio è **perfettamente prevedibile**.

1.2.2 La Radice del Problema: Stream vs Record

La vulnerabilità non risiede nella matematica del CBC in sé, ma nel modo in cui TLS gestisce la trasmissione di messaggi multipli "spezzati" nel tempo. È fondamentale distinguere tra due scenari:

1. **Stream Continuo (Una riga):** Se cifrassimo tutti i dati in un unico flusso continuo $(m_1, \dots, m_n, m_{n+1}, \dots)$, la catena CBC sarebbe interna e atomica. L'attaccante vedrebbe solo il risultato finale e non avrebbe alcun momento temporale per "inserirsi" tra un blocco e l'altro.
2. **Record Separati (Due righe):** In TLS, i dati viaggiano in pacchetti distinti.
 - Il server cifra il **Record 1** (M_1) e invia i risultati $(C_{1,1}, \dots, C_{1,n})$.
 - Qui avviene una **interruzione critica**: il sistema si ferma e attende nuovi dati.
 - L'attaccante intercetta l'ultimo blocco cifrato $C_{1,n}$ sulla rete.

È proprio questa pausa tra i record che espone la vulnerabilità: l'attaccante conosce l'IV futuro ($C_{1,n}$) prima che il prossimo messaggio venga creato. Questa prevedibilità è il cuore della vulnerabilità sfruttata dall'attacco ****BEAST****.

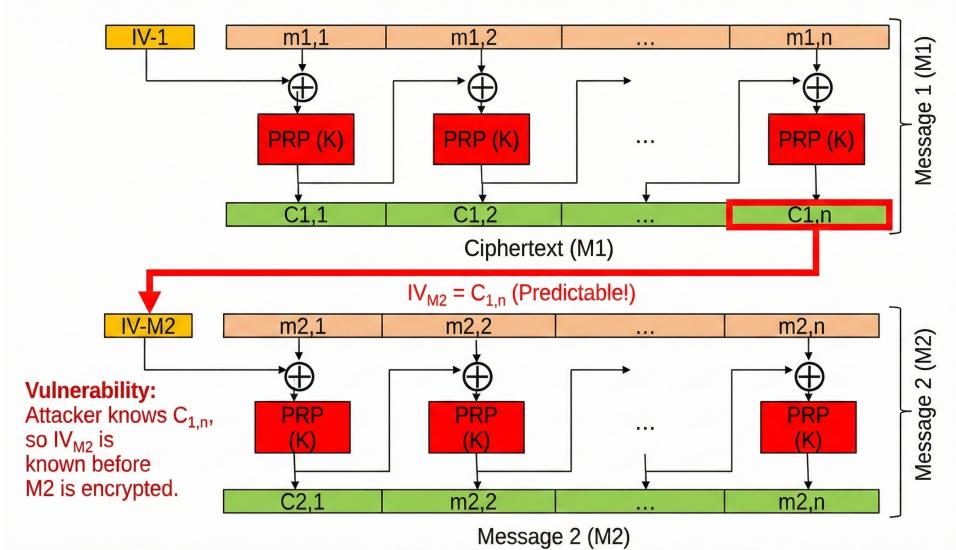


Figura 2: Rappresentazione della vulnerabilità Chained IV in TLS 1.0

Nota Storica: Le versioni successive (TLS 1.1 e 1.2) hanno corretto questo problema implementando un IV esplicito e generato casualmente per ogni singolo record, disaccoppiando così la dipendenza tra messaggi successivi.

1.3 La Meccanica dell'Attacco BEAST

L'obiettivo dell'attaccante è indovinare il contenuto di un blocco di testo in chiaro P_{target} (ad esempio, un cookie di sessione o una password) che è stato precedentemente cifrato nel blocco C_{target} .

Sappiamo che la cifratura del blocco target è avvenuta secondo la formula:

$$C_{target} = E_K(P_{target} \oplus C_{target-1})$$

L'attaccante vede C_{target} e conosce $C_{target-1}$ (il blocco cifrato precedente), ma non conosce P_{target} .

Sfruttando l'IV prevedibile, l'attaccante può montare un attacco a dizionario attivo (Chosen Plaintext Attack):

1. **Previsione dell'IV:** L'attaccante attende di poter iniettare un nuovo messaggio (o record). Sa che l'IV utilizzato per questo nuovo messaggio (IV_{attack}) sarà esattamente l'ultimo blocco cifrato che ha appena sniffato sulla rete.
2. **Formulazione dell'Ipotesi:** L'attaccante fa un'ipotesi sul contenuto del testo in chiaro che vuole scoprire. Chiamiamo questa ipotesi *GUESS* (ad esempio: "la password è UGO?").
3. **Costruzione del Chosen Plaintext:** L'attaccante induce il browser della vittima a crittografare un blocco appositamente creato (P_{chosen}). La formula per scegliere questo testo è il cuore dell'attacco:

$$P_{chosen} = IV_{attack} \oplus C_{target-1} \oplus GUESS$$

4. **Verifica (The XOR Trick):** Il sistema cifra questo blocco utilizzando l'IV prevedibile (IV_{attack}). Il risultato sarà un nuovo cifrato C_{probe} :

$$C_{probe} = E_K(P_{chosen} \oplus IV_{attack})$$

Sostituendo il valore di P_{chosen} calcolato al punto 3:

$$C_{probe} = E_K((IV_{attack} \oplus C_{target-1} \oplus GUESS) \oplus IV_{attack})$$

Per le proprietà dello XOR ($A \oplus A = 0$), i due IV_{attack} si annullano a vicenda all'interno della funzione di cifratura:

$$C_{probe} = E_K(C_{target-1} \oplus GUESS)$$

5. **Confronto:** Ora l'attaccante confronta il nuovo cifrato C_{probe} con il cifrato originale target C_{target} .

- Se $C_{probe} == C_{target}$, allora $E_K(C_{target-1} \oplus GUESS) == E_K(C_{target-1} \oplus P_{target})$.
- Di conseguenza, $GUESS == P_{target}$. L'ipotesi è corretta.

In pratica, poiché l'attaccante conosce esattamente l'IV che il server (o il client) userà per cifrare il nuovo messaggio, può calcolare un testo P_{chosen} tale che, una volta applicato l'XOR iniziale della modalità CBC, l'IV venga "neutralizzato".

Ciò che rimane all'interno della funzione di cifratura è puramente $C_{target-1} \oplus GUESS$. Questo permette all'attaccante di verificare se la sua ipotesi corrisponde al testo originale, trasformando la cifratura in un oracolo di verifica per un attacco a forza bruta (tipicamente condotto byte per byte).

Analizziamo l'esempio nella Figura 3.

1. Sia C_i il blocco contenente la password (vera) cifrata usando come IV il blocco precedente C_{i-1} (che funge da IV in CBC), quindi

$$C_i = E_k(C_{i-1} \oplus P_{target})$$

2. L'attaccante vuole verificare se la password è, ad esempio, "UGO". Conosce C_{i-1} (dal traffico precedente) e conosce X (l'IV che verrà usato per la prossima cifratura, predibile per costruzione). L'attaccante crea un messaggio scelto (Chosen Plaintext) P_{chosen} costruito così:

$$P_{chosen} = X \oplus C_{i-1} \oplus P_{guess}$$

Dove P_{guess} è la stringa che l'attaccante sta provando a indovinare.

3. Quando il sistema cifra questo nuovo messaggio, calcolerà:

$$C_{new} = E_k(X \oplus P_{chosen})$$

Sostituendo il valore di P_{chosen} calcolato sopra:

$$C_{new} = E_k(X \oplus (X \oplus C_{i-1} \oplus P_{guess}))$$

I due X si annullano ($X \oplus X = 0$), lasciando:

$$C_{new} = E_k(C_{i-1} \oplus P_{guess})$$

4. Se $C_{new} = C_i$, allora l'input della funzione di cifratura era identico. Dato che C_{i-1} è lo stesso in entrambi i casi, implica necessariamente che $P_{guess} = P_{target}$. L'attaccante ha indovinato la password.

What if predictable IV? - Chosen Plaintext Attack Demonstration

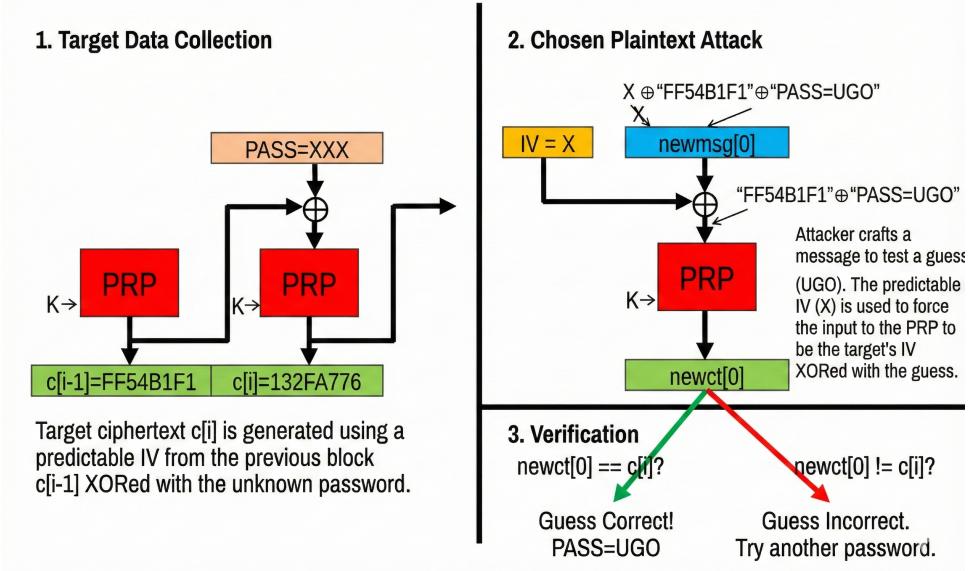


Figura 3: Esempio di attacco

1.4 Soluzione e Mitigazione

Come evidenziato dall'analisi dell'attacco, la vulnerabilità risiede strutturalmente nella prevedibilità dell'IV nel concatenamento dei blocchi (CBC). Le soluzioni adottate dall'industria sono state di tre tipi principali:

- **Soluzione Architetturale (TLS 1.1 e 1.2):**
Questi protocolli hanno risolto il problema alla radice implementando un **IV esplicito** per ogni singolo record. L'IV viene generato casualmente e trasmesso all'inizio del pacchetto cifrato, rendendolo indipendente dal ciphertext precedente ($IV \neq C_{last}$) e quindi imprevedibile per l'attaccante.
- **Mitigazione "Record Splitting" ($1/(n - 1)$ split):**
Per mettere in sicurezza le connessioni TLS 1.0 (dove l'IV è implicito), i browser hanno adottato una tecnica di frammentazione. Il messaggio viene inviato in due parti:
 1. Un primo frammento contenente solo 1 byte (o 0 byte). Questo frammento "consuma" l'IV predibibile noto all'attaccante.
 2. Il resto del messaggio viene inviato subito dopo. L'IV per questo secondo frammento sarà l'ultimo blocco cifrato del primo frammento. Essendo generato al momento, l'attaccante non può prevederlo in tempo per calcolare il suo P_{chosen} .
- **Workaround (Deprecato):**
Prima della migrazione a TLS 1.1+, una mitigazione comune lato server era forzare l'uso di cifrari a flusso come **RC4**, che non utilizzano la modalità CBC e non sono quindi soggetti a questo attacco. Tuttavia, RC4 è stato successivamente abbandonato (RFC 7465) a causa di gravi bias statistici nel keystream.

In conclusione, questa vulnerabilità compromette la sicurezza semantica del sistema, permettendo di recuperare il testo in chiaro tramite un attacco di enumerazione. Ciò costituisce una violazione formale della proprietà IND-CPA (Indistinguishability under Chosen Plaintext Attack), dato che l'attaccante è in grado di distinguere i crittogrammi sfruttando la predicitività dell'IV.

1.5 Rendere Pratico l'Attacco: Chosen Boundary Attack

"La domanda fondamentale è: un attacco di questo tipo è realmente fattibile?

Teoricamente, l'attaccante deve essere in grado di forzare il browser della vittima a cifrare messaggi scelti (Chosen Plaintext). Un'obiezione comune era che, se un attaccante riuscisse a eseguire codice nel browser della vittima, sarebbe già 'Game Over'. Tuttavia, questo non è vero: le moderne sandbox e flag come `HttpOnly` impediscono agli script di leggere direttamente i cookie sensibili dalla memoria.

È qui che Duong e Rizzo, con l'attacco **BEAST**, hanno cambiato le regole del gioco. Hanno dimostrato come un **Active Agent** (un semplice script in una pagina web) possa aggirare le protezioni risolvendo due problemi apparentemente insormontabili:

- **Il problema Software:** Come generare traffico massivo e manipolare i confini dei blocchi usando tecnologie web standard (WebSockets/Java) senza avere il controllo totale della macchina. Ad oggi è possibile implementare tecniche simili, nel quale il browser della vittima esegue script con tool come <https://beefproject.com/>, anche detti attacchi **MITB - Man In The Browser**.
- **Il problema Computazionale:** In teoria, l'attacco a IV predicibile sembrerebbe impraticabile su cifrari moderni. Se si usa AES, la dimensione del blocco è 128 bit (16 byte). Un attacco a forza bruta richiederebbe di indovinare l'intero blocco contemporaneamente, risultando in 2^{128} tentativi: computazionalmente impossibile. Tuttavia, Duong e Rizzo hanno introdotto il concetto di **Chosen Boundary Attack**, rendendolo la complessità lineare.

1.5.1 Chosen Boundary Attack

L'idea fondamentale è manipolare la posizione dei dati all'interno del flusso cifrato per "spostare il confine" del blocco AES. L'attaccante non cerca di decifrare l'intero blocco, ma lo allinea in modo che contenga quasi tutti dati noti e solo un byte ignoto.

Per comprendere la gravità dell'attacco BEAST, analizziamo la sua applicazione pratica nel furto di un cookie di sessione HTTP (**Session Hijacking**).

- **Vittima:** Un utente autenticato su un sito sicuro (es. banca). Il browser possiede un cookie segreto, ad esempio: `Cookie: ID=X29A...`
- **Attaccante:** Controlla parzialmente il browser della vittima tramite un *Active Agent* (JavaScript malevolo caricato da un banner o un sito visitato dalla vittima).
- **Obiettivo:** Estrarre il valore del cookie byte per byte.

L'attaccante induce il browser a inviare richieste verso il server della banca. Il browser compone la richiesta concatenando dati controllati dall'attaccante (es. l'URL) con dati segreti (gli Header HTTP automatici).

Una richiesta tipica appare così:

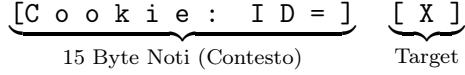
```
POST /<PADDING_ATTACCANTE> HTTP/1.1
Host: bank.com
... [Header Vari] ...
Cookie: ID=X29A...
```

L'attaccante manipola la lunghezza del `<PADDING_ATTACCANTE>` per allineare i blocchi AES (16 byte). L'obiettivo è far sì che il "confine" (boundary) tra un blocco e l'altro cada esattamente dopo il primo carattere sconosciuto del cookie.

Supponiamo che l'attaccante sappia che il prefisso del cookie è "Cookie: ID=". Egli calcola il padding affinché il blocco di interesse sia composto da:

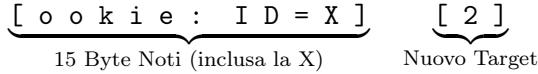
1. **15 Byte Noti:** La stringa "Cookie: ID=" (o parte del padding precedente + header).
2. **1 Byte Ignoto:** Il primo carattere del segreto ('X').

Visualizzazione del Blocco AES:



Una volta allineato il blocco:

1. **Sniffing:** L'attaccante registra il blocco cifrato C_{target} generato dalla richiesta della vittima.
2. **Guessing:** L'attaccante usa JavaScript per inviare nuove richieste, provando a indovinare il byte mancante.
 - Tentativo 1: Invia un blocco che termina con 'A'. Ottiene C_{guess} .
 - Confronto: Verifica se $C_{guess} == C_{target}$ (sfruttando l'IV predicibile per normalizzare il confronto).
 - ...
 - Tentativo 'X': I blocchi coincidono. Il byte è trovato.
3. **Shift (Sliding Window):** L'attaccante riduce il padding di 1 byte. Il blocco AES "scivola" a destra e ora include due caratteri del segreto:



Il processo si ripete con complessità lineare $256 \times N$.

L'attacco BEAST ha dimostrato che la **Same Origin Policy** (che impedisce la lettura diretta dei cookie via script) può essere aggirata forzando il browser a cifrare i dati e analizzandoli sulla rete, sfruttando la vulnerabilità CBC di TLS 1.0.

1.5.2 Analisi della Complessità

Questa tecnica abbatte drasticamente la sicurezza:

- **Approccio Classico:** Per un segreto di N byte, la complessità sarebbe 256^N (esponenziale).
 - Esempio (8 byte): $256^8 = 2^{64} \approx 1.8 \times 10^{19}$ tentativi (impraticabile).
- **Chosen Boundary Attack:** La complessità diventa lineare, $256 \times N$.
 - Esempio (8 byte): $256 \times 8 = 2^{11} = 2048$ tentativi (banale).

2 L'attacco CRIME (Compression Ratio Info-leak Made Easy)

La vulnerabilità CRIME nasce da una scelta di design di TLS apparentemente obbligata, dettata dai principi della teoria dell'informazione:

1. **Il vincolo dell'Entropia:** La cifratura moderna è progettata per produrre output indistinguibili dal rumore casuale (massima entropia). Di conseguenza, tentare di comprimere un messaggio *già cifrato* è inutile: gli algoritmi di compressione non trovano pattern ripetuti nel caos randomico e non guadagnano spazio (anzi, spesso aggiungono overhead).

$$\text{Compress}(\text{Encrypt}(M)) \approx \text{Size}(\text{Encrypt}(M))$$

2. **La Decisione Critica:** Per ottenere benefici in termini di performance e larghezza di banda, i progettisti di TLS hanno dovuto necessariamente posizionare la compressione **prima** della cifratura.

$$C = \text{Encrypt}(\text{Compress}(M))$$

3. **Il Fallimento di Sicurezza:** Questa decisione, sebbene corretta per l'efficienza, ha creato il canale laterale (side-channel) sfruttato da Duong e Rizzo nel 2012. Poiché la cifratura preserva (approssimativamente) la lunghezza del suo input, la lunghezza finale di C rivela quanto efficacemente M è stato compresso.

Dato che la compressione dipende dal contenuto (ridondanza), la lunghezza di C diventa un oracolo che rivela informazioni sul contenuto di M (il segreto).

A differenza di BEAST, che dipendeva dalla modalità CBC, CRIME funziona **indipendentemente dal cifrario utilizzato** (AES, RC4, ecc.). La vulnerabilità non risiede nell'algoritmo di cifratura, ma appunto nel fatto che la compressione avviene *prima* della cifratura, creando un canale laterale (side-channel) basato sulla dimensione.

2.1 Analisi Formale: Violazione della IND-CPA

L'attacco CRIME rappresenta una violazione da manuale della proprietà **IND-CPA** (Indistinguishability under Chosen Plaintext Attack).

Per definizione, uno schema di cifratura è IND-CPA sicuro se un attaccante non è in grado di distinguere i ciphertext corrispondenti a due messaggi in chiaro scelti da lui, m_0 e m_1 , aventi la **stessa lunghezza**. Tuttavia, l'introduzione della compressione crea un canale laterale (side-channel) sulla lunghezza che rompe questa proprietà.

Supponiamo che il sistema cifri concatenando un segreto $S = \text{"SHARON"}$.

1. **Fase di Sfida:** L'attaccante sceglie due messaggi di input di eguale lunghezza:
 - $m_0 = \text{"AAA"}$ (Nessuna correlazione con il segreto).
 - $m_1 = \text{"SSS"}$ (Correlazione con l'inizio del segreto).
2. **L'Oracolo (Encryption):** Il sistema comprime e poi cifra. Poiché la cifratura preserva la lunghezza dell'input compresso:

$$\text{Len}(C_0) \approx \text{Len}(\text{Compress}(\text{"AAA"} || \text{"SHARON"}))$$

$$\text{Len}(C_1) \approx \text{Len}(\text{Compress}(\text{"SSS"} || \text{"SHARON"}))$$

3. **La Distinzione:** Nel caso di m_1 , l'algoritmo di compressione (es. DEFLATE) rileva la ripetizione del carattere 'S' tra l'input dell'attaccante e il segreto, riducendo la dimensione totale.

$$\text{Len}(C_1) < \text{Len}(C_0)$$

4. **Conclusione:** L'attaccante riceve i ciphertext C_0 e C_1 . Semplicemente osservandone la lunghezza, può determinare quale dei due è stato compresso e su quale carattere (S), riuscendo così a trovare la prima lettera del segreto. L'incapacità del sistema di nascondere quale dei due messaggi è stato cifrato (distinguibilità) decreta il fallimento della sicurezza IND-CPA.

2.2 Dettagli Tecnici: DEFLATE e LZ77

L'attacco reale si basa sul funzionamento dell'algoritmo **DEFLATE** (usato in TLS e HTTP), che combina due sotto-algoritmi:

1. **LZ77 (Il cuore dell'attacco):** Sostituisce le stringhe ripetute con puntatori nella forma (offset, length).

- Se il testo contiene

GIUSEPPE BIANCHI AND MARCO BIANCHINI

la seconda occorrenza di BIANCHI viene sostituita da un puntatore all'indietro,

GIUSEPPE BIANCHI AND MARCO (-18, 7)NI

2. **Codifica di Huffman:** Comprime ulteriormente i simboli in base alla frequenza.

2.3 Esecuzione Pratica: L'Attacco al Cookie

L'attacco si svolge in uno scenario in cui l'attaccante, tramite un agente attivo nel browser (es. JavaScript), costringe la vittima a generare richieste HTTP. Queste richieste fondono inevitabilmente due componenti nello stesso flusso di dati, che verranno **compressi e cifrati insieme**:

- Il **Cookie segreto**, allegato automaticamente dal browser (es. `Cookie: twid=flavia`).
- Il **Testo iniettato** arbitrariamente dall'attaccante (es. `...q=twid=a...`).

Il recupero del segreto avviene monitorando le variazioni di dimensione del pacchetto cifrato:

1. **Il Tentativo Fallito:**

Inizialmente, l'attaccante inietta una supposizione errata, ad esempio provando la lettera "a" (`twid=a`). L'algoritmo di compressione (LZ77) analizza il flusso combinato:

`...Cookie: twid=flavia... GET /?q=twid=a...`

Poiché la sequenza `twid=a` non appare nel cookie reale (che contiene "f"), non c'è ridondanza significativa da sfruttare. Il risultato è un pacchetto cifrato di lunghezza base L .

2. **Il Tentativo Riuscito:**

La situazione cambia drasticamente quando l'attaccante indovina il carattere corretto, iniettando `twid=f`.

`...Cookie: twid=flavia... GET /?q=twid=f...`

L'algoritmo LZ77 rileva che la sequenza `twid=f` è un duplicato esatto dell'inizio del cookie e la sostituisce con un **puntatore** (back-reference) molto più breve dei caratteri originali.

3. Conferma e Iterazione:

Questa compressione efficace riduce la dimensione totale del ciphertext a $L - \Delta$. Questa differenza di lunghezza funge da oracolo: conferma all'attaccante che la lettera "f" è corretta. L'attacco procede quindi linearmente (`twid=fa, twid=fb...`), "mangiando" il cookie un carattere alla volta.

La complessità è lineare rispetto alla lunghezza del segreto ($O(N)$), rendendo l'attacco molto veloce.

- **La Soluzione:** L'unica mitigazione reale è stata **disabilitare la compressione TLS**. Chrome l'ha disabilitata definitivamente dopo settembre 2012.
- **Curiosità Storica:** Come mostrato nella slide "R.I.P. TLS Compression", questa vulnerabilità non era nuova. Era nota teoricamente dal 2002 (paper di Kelsey), ma ignorata per 10 anni perché ritenuta impraticabile.