**EUGENIO LIOTINE - Python**

---------------------------------------------------------------------------------------------------------------------------

**Symmetric encryption**

[] **Attacchi**

**[1] Meet in the middle**

target: algoritmi simmetrici con doppia cifratura (es 2DES)

conosciamo: plaintext e ciphertext (known plaintext attack)

finalità: trovare k1 e k2 usate per cifrare il plaintext

**Funzionamento doppia cifratura**:

plaintext + k1 -> intermediate artifact +k2 -> ciphertext

Funzionamento attacco: ricorda che : -> **doppia cifratura** offre solo **doppia sicurezza a doppio costo computazionale** (non sicurezza esponenzialmente migliore). Come se fossero due cifrature separate.

**Procedimento dell’attacco:**

da plaintext si bruteforca l’encrypt (k1) e da ciphertext si bruteforca il decrypt (k2) finchè non otteniamo un artefatto intermedio comune.

Dal bruteforce di k1 si ottiene un dictionary con tutte le chiavi possibili di cifratura del plaintext e facendo il bruteforce di k2 ad ogni iterazione si verifca di avere quel risultato nel dictionary (possibili falsi positivi, Basile cicla tutto)

**[2] Bit Flipping**

**Target**: stream algorithms molto vulnerabili (chacha20),salsa20, block ciphers: CBC parzialmente (**altera il blocco precedente a quello da modificare**). ECB impossibile con bit flipping.

**Conosciamo:**

-struttura plaintext (o plaintext completo)

**Finalità:**

-modifiche puntuali al ciphertext -> forzando così un nuovo valore voluto dall’attaccante nel plaintext (es bit admin a 1). **Manipolazione di plaintext.**

\*\*Modifiche all’esame: vs stream: cambiare posizione del bit da cambiare, new\_byte (valore che vogliamo (2)), old\_byte (valore attuale es 1 che deve diventare 2)\*\*

**Bit Flipping in CBC mode**: modificare blocco precedente per fare modifiche al successivo (caso cookie: modifico la parte dell’user per ottenere admin=1) al blocco precedente a quello che vogliamo modificare facciamo **XOR** con “admin=0” per annullare questo valore nel plaintext e poi **XOR** con “admin=1” per metterci questo valore

**\*\*NOTA\*\***: vecchio valore **XOR** vecchio valore = **RIPULIRE**

Valore ripulito **XOR** nuovo valore = **SCRIVERE**

**[3] Server Detection Mode (ECB vs CBC)**

**Target:** AES CBC o ECB con **input** almeno 2 blocchi interi

**Conosciamo:**

**-**Ciphertext

**Finalità:**

Scoprire la modalità di encryption di AES.

Si mandano al Server 2 blocchi uguali e si vede il ciphertext che ritorna, se i 2 blocchi cifrati sono ancora uguali si lavora in ECB viceversa CBC.

funzionamento attacco: partiamo dall’avere un input modificabile, facciamo in modo di vedere le modifiche a due blocchi interi (add AAAAA all’inizio per saturare il blocco iniziale)

inviare questi due blocchi uguali (uguali CBC altrimenti ECB)

**[4] Copy&Paste**

**Target:** server ECB

**Finalità:**

cambiare parzialmente o totalmente un blocco.

**Sviluppo dell’attacco:**

metto AAAA(email lunga) per isolare il blocco da modificare (es |aaaaaaarole=|userPPPPPPP| ->creare un blocco ad hoc (adminPPPPPPP) -> l’oracolo lo cifrerà -> sostituire il blocco adminPPPPPPP al blocco userPPPPPPP e lo invierà al Server di accesso. (**NB: ci sono due server**, uno forgia i cookie e l’altro testa il cookie (se sei admin))

**Crei 3 blocchi da inviare**, i primi 2 blocchi sono quelli con email e UID, **il 3 blocco è invece quello fatto cifrare appositamente con “admin”**

**[5] ECB ACPA**

**Target:** server ECB

**Finalità:**

decrypt del ciphertext

**Sviluppo dell’attacco:**

servono due blocchi liberi per blocco da cifrare (eventuali possibili ottimizzazioni). Di base bisogna confrontare un blocco totalmente noto tranne 1 byte che subirà bruteforce con il blocco nostro che itera cambiando sempre quel byte

| is the secret:0| ← 0: elemento da trovare

| is the secret:B| ←- B: bruteforce byte per byte (B itera fino ad arrivare a 0)

Una volta trovato un byte i blocchi shiftano:

|is the secret:01| ← 1: elemento da trovare

|is the secret:0B| ←- B: bruteforce byte per byte (B itera fino ad arrivare a 0)

**[6] CBC Padding Oracle Attack**

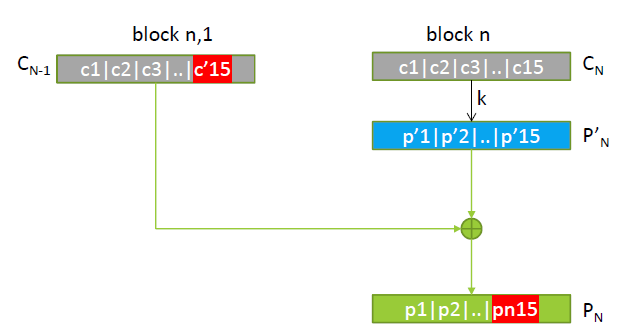
**Target:**

server CBC che ritorna un messaggio sul padding 🡪 **padding (OK/NO)**

**Finalità:**

decrypt di un intero ciphertext (blocco per blocco alterando temporaneamente il

blocco precedente a quello che vogliamo decifrare)

**APPROCCIO USATO CBC BIT FLIPPING sul blocco precedente**

**Sviluppo dell’attacco:**

partiamo con il ciphertext

il server ci torna OK solo quando p15 = x01

c'15 **XOR** p'15 = x01 OK

sappiamo il c15 indovinato

sappiamo che ha fatto 1

possiamo quindi calcolare p'15 = x01 **XOR** c15

abbiamo c15 (versione cifrata originale)

lato attaccante: c15ORIGINALE **XOR** p'15 -> plaintext

**\*NOTA: il server deve dirci che il padding va bene per implementare questo attacco**

**[7] Key Stream nonce reuse attack**

**Target:** Algoritmo Stream che riusa sempre lo stesso **Nonce** (e stessa chiave)

**Finalità:**

Scoprire la chiave usata per decifrare un testo (e ottenere tutti i plaintext).

**Sviluppo dell’attacco:**

costruire la chiave del key stream a partire da una tabella di 256 valori (2^8).

Si fa un **for** **loop** che va dal **byte 00000000** al **byte 11111111** facendo lo **XOR** con la prima lettera del ciphertext vedendo se è printabile.

Se lo è, incrementa la frequenza della lettera printabile nella posizione del byte candidato.

**key[ ]**

**c[ ] = ciphertext;**

**index = 0;**

**FOR index in range(16)**

**{**

**i=0;**

**counter\_frequenza[ ] = 0;**

**FOR i in range(256)**

**{**

**(byte 00000000)🡪 result = (byte0) \_base10 XOR c[index]**

**IF (result🡪printable)**

**🡪 ++counter\_frequenza[byte0]**

**}**

**counter\_frequenza.sort(desc)**

**key[index] = counter\_frequenza[0]**

**}**

**Key =** la chiave si costruisce volta per volta prendendo il byte con più alta frequenza dalla tabella.

**I valori della tabella vengono ordinati in maniera decrescente** in modo tale che **il valore con maggior frequenza** occorra come primo valore all'interno della tabella, cioè si trovi in **posizione 0** nella tabella.

Una volta **trovato il primo byte della chiave**, **per scoprire il secondo byte della chiave si passa al secondo carattere del ciphertext** e così via fino a trovare tutta la chiave.

In pratica **si utilizzano due for loop**:

-uno più esterno che si occupa di incrementare l'index del ciphertext.

-l'altro più interno che si occupa di controllare ed incrementare il byte index dal byte numero 0 al byte numero 255.

Ad ogni incremento dell'index del ciphertext il **byte counter i** e le frequenze della tabella vengono azzerati in modo tale da calcolare nuove frequenze di occorrenza per lo **XOR** tra l'attuale byte e l'attuale index del ciphertext.

**Alla fine di ogni ciclo** del loop più esterno, ovvero quello che incrementa l'index del ciphertext **il carattere con massima frequenza** appena calcolato, **viene aggiunto alla chiave.**

(versione base è solo contare i caratteri printabili trovati con una certa chiave, quelle chiavi che permettono di rendere printabile tutta la colonna sono potenziali chiavi (ma troppe))

**\*\*NOTA\*\*: Piu ciphertext abbiamo piu l’attacco avrà successo.**

**Asymmetric encryption**

**[] Attacchi**

**[1] Factorize the modulus**

**Target:** RSA with small modulus n (**n<150 bit**)

**Conosciamo:**

-e (public exponent)

-n (modulo)

**Finalità:** trovare p e q (→ phi(n) -> troviamo d (private exponent) -> ROMPIAMO RSA)

[] **Implementazione**

f = FactorDB(modulo)

f.connect()

print(f.get\_factor\_list())

**[2] Fermat’s factorization**

**Target:** RSA with p e q close numbers

**Conosciamo:** n (modulo)

**Finalità:** trovare p e q (→ phi(n) -> troviamo d (private exponent) -> ROMPIAMO RSA)

[] **Implementazione**

n = pq = (a+b)(a – b) = a^2 – b^2 → b^2=a^2-n

con p=a+b q=a-b

**[3] Common prime**

**Target:** RSA con un cattivo generatore di numeri casuali🡪**RSA with bad RNG**(es no init di /dev/random) .Bassa entropia nei moduli

**Conosciamo:** n1, n2🡪2 moduli diversi ma con un **numero primo in comune p1** (si pensi a 2 certificati diversi generati dallo stesso utente)

**Finalità:** sappiamo che **n1=p1\*p2** e che **n2=p1\*p3**, con un **cattivo generatore** avremo un **MCD diverso da 1** in quanto a causa del cattivo generatore **si ha p1 come fattore comune.**

Possiamo trovare p1 (**MCD dei moduli n1 ed n2**) e **da esso trovare p2 e p3**:

-**p2 = n1/p1,**

**-p3 = n2/p1**

ROMPIAMO RSA

**[4] Common modulus**

**Target:** RSA **due chiavi pubbliche e1, e2** con **modulo n in comune**

**Conosciamo:**

-e1, e2 chiavi pubbliche

-modulo n

-ciphertext delle chiavi private d1 e d2 che **è uguale**

**Finalità:** decifrare il messaggio🡪 **decrypt del ciphertext**

[] **Implementazione**

res = egcd(e1,e2)

u = res[1]

v = res[2]

# c1^u mod n

decrypted = pow(c1,u,n) \* pow(c2,v,n) % n

**[5] Low public exponent**

**Target:**

-RSA con **esponente pubblico piccolo (3,17)**

-**messaggio troppo corto rispetto alla lunghezza della chiave**

**Facilmente riconoscibile perché non esco dallo spazio riservato al modulo n**

|----------------------------| ← messggioCorto^esponentePiccolo **non esco dal modulo**

m m^3

--\*\*\*cifro con **c = m^e** **mod n** se ho il public exponent **e** e non supero il modulo **root^e(c)** e **trovo il messaggio\*\*\***--

**cifrare:** con **c = m^e** **mod n**

**decriptare il messaggio: root^e(c)**

**Conosciamo:** esponente pubblico

**Finalità:** decifrare messaggio

**[6] Hastad’s broadcast**

**Target:**

-RSA che cifra uno stesso messaggio e lo invia a e-destinatari.

**-Ogni destinatario ha una sua chiave pubblica🡪(e, n)**

**-Ogni destinatario HA LO STESSO ESPONENTE PUBBLICO PICCOLO** es (3,17)

**\*NOTA: chiavi pubbliche diverse perché pur avendo l’esponente pubblico uguale possono avere moduli diversi**

**\*\*NOTA\*\*: e = esponente pubblico (e, n) = chiave pubblica**

**Conosciamo:**

**-e-diversi messaggi cifrati**

-**e-chiavi pubbliche dei destinatari**.

**Sviluppo attacco:**

facciamo un **CRT** per espandere il modulo e poter fare come il low public exponent, radice dentro questo spazio matematico esteso

**Finalità:** decifrare il messaggio ma in un modulo più grande

[] **Implementazione**

e=3

key1 = RSA.generate(n\_length, e1=e)

key2 = RSA.generate(n\_length, e2=e)

key3 = RSA.generate(n\_length, e3=e)

n1 = key1.n

n2 = key2.n

n3 = key3.n

N = n1 \* n2 \* n3

N1 = n2 \* n3

N2 = n1 \* n3

N3 = n1 \* n2

g, u1, v1 = egcd(N1, n1)

g, u2, v2 = egcd(N2, n2)

g, u3, v3 = egcd(N3, n3)

c = c1 \* u1 \* N1 + c2 \* u2 \* N2 + c3 \* u3 \* N3 # testo cifrato nello spazio mat grosso

dec\_int = iroot(3,c) # decifro in quello spazio matematico

**[7] LSB\_Oracle (Least Significat Bit)**

**Target:** Oracle that leaks the last decrypted bit

**Conosciamo:**

-e (pub exponent),

-n (modulo)

**Quindi conosciamo la chiave pubblica (e, n)**

**Sviluppo attacco:**

inviamo un ciphertext cifrato con **il public exponent e** fornito

**Finalità:** decipher the original message

**[8] Low Private exponent attacks [NO IMPLEMENTAZIONE]**

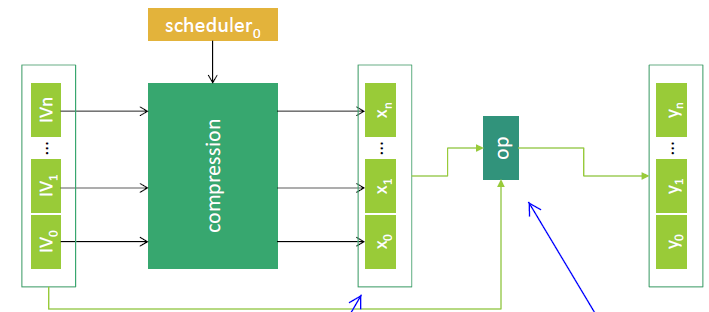
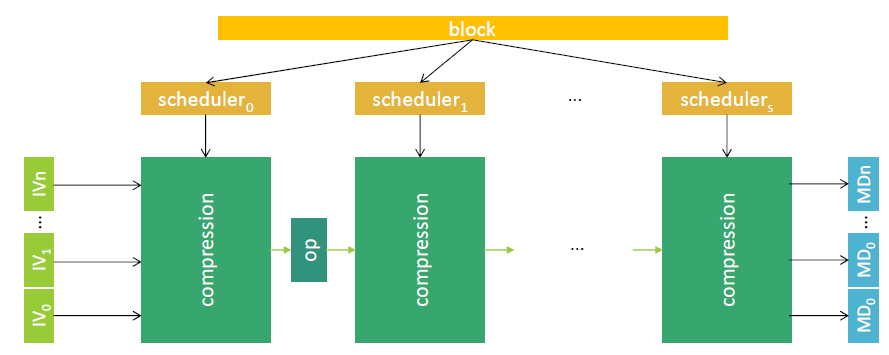
Se esponente privato piccolo -> RSA Rotto, ottengo la chiave

**n^1/4** è considerato un **private exponent piccolo**, es se n = 1024, radice quarta è tipo 5.qualcosa cioè **6 bit**

**MD4 Merkle-Damgard architecture**

**Typical block size:** 512/1024 bit

**num. scheduler in sha256:** 64



**[1] Length extension attack**

**Target:** md4/md5 usato per keyed digest (key e poi messaggio) con ultimo blocco non pieno (con padding)

**Conosciamo:**

-il digest

-la lunghezza del messaggio hashato (caso mai tramite sniffing)

**Finalità:** aggiungere dati alla fine del messaggio originale che NON abbiamo. Otterremo il nuovo digest del messaggio.

Questo può essere fatto senza il bisogno di conoscere il **messaggio m** o la **key k**

Il **Length extension attack** è molto efficace contro i MAC digestspoiché **può creare messaggi con un valido MAC.**

L’HMAC viene utilizzato per prevenire certi attacchi in quanto più sicuro.

**Risultato:**

**messaggio | paddingVecchio | nuovo messaggio | nuovo Padding**

Praticamente **aggiungiamo un blocco intero** al messaggio precedente

**Rompi l’autenticità:** aggiungi qualcosa alla fine di un messaggio come se fosse scritto dal mittente

What good can it do to the attacker ? Well, consider the following (flawed) construction for a [Message Authentication Code](http://en.wikipedia.org/wiki/Message_authentication_code): for a given key *k* and data to protect *d*, compute SHA-1(*k||d*) as the MAC value. This scheme breaks in the presence of the length-extension attack: if I, as an attacker, sees a MAC for a message *d*, then I can compute the MAC for a message *d'* which extends *d* -- and I can do that without knowing *k||d* (so, in particular, without knowing the key). This allows me to **forge** messages with a valid MAC.

The length-extension attack is the reason why, when building a MAC out of a hash function, we need something a bit more convoluted, namely [HMAC](http://en.wikipedia.org/wiki/HMAC) (which is safe against it).

**SRP: Secure Remote Password**

L’SRP è sostanzialmente un Diffie Hellman, (cioè la condivisione di una chiave di sessione tra le due parti).

SRP produce anche un **proof of session per verificare di aver creato la stessa chiave.**

