Servizio Crypto_Signatures CTF 2019

– Jelly Hinge Team –

1 Introduzione

Crypto_Signatures è uno dei servizi proposti alla uniCTF2019[1]. Dato un set di firme ed un algoritmo di verifica veniva richiesto di produrre una firma valida diversa da quelle fornite.

2 L'algoritmo di verifica

L'algoritmo di verifica prende in input un array `input` contenente non meno di 514 stringhe in numero pari. Ciascuna di queste viene quindi distribuita in quattro gruppi:

`id` contiene la sola stringa `input[0]` che identifica il mittente

`msg` contiene la sola stringa `input[1]` in cui è riportato in chiaro il messaggio

`sign` contiene le stringhe `input[2:514]` contenenti la codifica di interi esadecimali di 64 cifre

`others` contiene le stringhe `input[514:]` tali che quelle di indice pari codifichino 0 o 1 mentre quelle di posto dispari un intero a 64 cifre esadecimale

2.1 msg_internal_validity

l'algoritmo inizia verificando che il messaggio sia ben formato controllando che sia delle forma

```
sender_id + " sent " + money + " zuccoins to " + receiver_id
```

[1] I sorgenti, ed in particolare il programma `verifier.py` a cui si fa riferimento nel seguito, sono reperibili al link
https://github.com/unictf/unictf-2019/blob/master/services/Crypto_Signatures

dove `money` deve essere un float strettamente minore di 500, `sender_id` deve uguagliare `id`, e `receiver_id` deve contenere 64 caratteri e nessuno spazio[2] Tutte le fasi successive sono volte a verificare l'identità dell'autore

2.2 msg_to_hashes

Posta H una funzione di hashing a 256 bit (nel caso specifico `SHA256`) ed $\texttt{h\_msg} = H(\texttt{msg})$, l'algoritmo calcola la rappresentazione binaria di quest'ultimo intero. Detto dunque b_k il k-esimo bit di `h_msg` l'algoritmo pone

$$\begin{cases} \texttt{sign[2k]} = H(\texttt{sign[2k]}) & \text{Se } b_k = 0 \\ \texttt{sign[2k+1]} = H(\texttt{sign[2k+1]}) & \text{Se } b_k = 1 \end{cases}$$

Il risultato viene dunque passato ad altre due funzioni:

`make_top_hash_from_leaves` `make_top_hash_from_others` (1)

2.3 Il resto

Non introduciamo nel dettaglio le due funzioni sopracitate in quanto calcolano deterministicamente una stringa a partire dagli array `sign` ed `others`. L'algoritmo si conclude verificando che questa stringa coincida con `id` ed eventualmente accetta il messaggio

3 L'attacco

L'attacco mostrato di seguito produce un messaggio firmato correttamente del quale non è però possibile determinare a priori l'identità del mittente.

3.1 Generazione dell'identità

Detta a una stringa contenente la codifica di un intero di 64 cifre esadecimali, costruiamo `h_sign` in modo che

$$\texttt{h\_sign[k]} = H(a) \qquad \forall k \in \{0, \dots, 255\}$$

ed `others` in modo che sia ben formattato. Ad esempio, per $k \in \{0, \dots, 9\}$

$$\begin{cases} \texttt{others}[k] = \text{"0"} & \text{Se } k \text{ pari} \\ \texttt{others}[k] = \text{"0"} * 64 & \text{Se } k \text{ dispari} \end{cases}$$

[2] l'idea sarebbe verificare che receiver_id sia un effettivo id - ovvero un intero di 256 bit rappresentato in forma esadecimale - tuttavia passano il controllo stringhe contenenti qualsiasi carattere alfanumerico e speciale diversi dallo spazio

Con questi due array eseguiamo le funzioni (1) ottenendo una stringa `fake_id`

3.2 Creazione del messaggio

Per fare in modo che il messaggio passi il test di validità fissiamo una qualunque stringa `receiver_id` di 64 caratteri non contenenti spazi, una stringa `money` che codifica un float minore di 500. Il messaggio risultante sarà

```
msg = fake_id + " sent " + money + " zuccoins to " + receiver_id
```

3.3 Falsificazione della firma

Detto `h_msg` = H(`msg`) e calcolata la sua rappresentazione binaria poniamo b_k il k-esimo bit. Costruiamo l'array `sign` come segue

$$\texttt{sign}[2k+1-b_k] = H(a) \qquad \texttt{sign}[2k+b_k] = a$$

e resituiamo in output l'array ottenuto concatenando `fake_id`, `msg`, `sign`, `others`.

4 Correttezza

Brevemente osserviamo che la procedura sopra mostrata è corretta dal momento che, raggruppando gli elementi di `sign` a due a due osserviamo che, la k esima coppia sarà

$$\begin{cases} (a, H(a)) & \text{Se } b_k = 0 \\ (H(a), a) & \text{Se } b_k = 1 \end{cases}$$

dove il primo elemento della coppia ha indice di posto pari (per l'esattezza $2k$). Applicando `msg_to_hashes` otteniamo in entrambi i casi una stringa in cui tutti gli elementi coincidono con $H(a)$. Il determinismo delle funzioni (1) assicura che l'algoritmo si fermerà per costruzione con la stringa `fake_id` che ovviamente coincide con l'identificativo fornito.
Il controllo di verifica interno invece viene banalmente superato per costruzione.