Ingannare con la trasparenza

Mattia Monga
Dip. di Informatica — Università degli Studi di Milano
mattia.monga@unimi.it

Gli esperti di sicurezza sono abituati a diffidare della "*security through obscurity*", di quelle misure di protezione, cioè, che basano la difesa solo sulla non conoscenza dei dettagli da parte degli attaccanti. A volte si tratta di misure di comodo, spesso frutto di assunzioni sbagliate ("Li proteggo perché è richiesto per legge, ma tanto i miei dati non interessano a nessuno"), che finiscono per renderci vulnerabili anche ai criminali meno sofisticati, come quando lasciamo la chiave di casa sotto lo zerbino. Oppure si hanno casi di ignoranza arrogante: chi di noi non ha conosciuto fornitori che dichiarano inarrivabili livelli di sicurezza grazie a qualche loro peculiare procedimento crittografico tenuto, naturalmente, rigorosamente segreto? Eppure Auguste Kerckhoffs già nel 1883 aveva enunciato su una rivista di crittografia militare il principio che ora guida la ricerca in questo campo: bisogna tenere segrete le chiavi, non il meccanismo di cifra, che è invece bene assumere che possa essere conosciuto ed esaminato dal nemico. Anzi, come ricorda spesso Bruce Schneier: siccome "chiunque è in grado di progettare un algoritmo crittografico che supera le proprie capacità di crittoanalisi" è bene usare solo la crittografia uscita indenne da anni di scrutinio pubblico e universale.

L'*oscurità,* dunque, è spesso sintomo di analisi di sicurezza affrettate e superficiali: insomma, un ottimo indizio di debolezza per un attaccante in cerca di un bersaglio alla sua portata. La *trasparenza* è certamente un miglior biglietto da visita... Talvolta però perfino la 'trasparenza' può essere uno strumento di attacco ed è di questa possibilità che voglio discutere qui.

Ken Thompson (1943–), nella lezione magistrale ("*Reflections on trusting trust*") tenuta per la consegna del premio Turing 1983 (assegnato a lui e Dennis Ritchie per i loro contributi alla teoria dei sistemi operativi e l'implementazione di UNIX) descrisse un attacco insidiosissimo che ora è noto come 'attacco *trusting trust*' o *Thompson hack*, sebbene Thompson stesso lo facesse risalire a fonti precedenti. L'articolo rimane un gioiello: tre pagine di *humor* e intelligenza tecnica che consiglio caldamente a chi voglia approfondire[1]. Tenterò di riassumere il principio dell'attacco cui risultano essere intrinsecamente vulnerabili sostanzialmente tutti i sistemi digitali sufficientemente complessi. Il punto chiave, infatti, è che i sistemi informatici sono *descritti ed esaminati* a un livello di astrazione diverso da quello a cui vengono effettivamente *eseguiti,* il livello

[1] L'articolo è disponibile qui: http://dx.doi.org/10.1145/1283920.1283940

cioè in cui producono i loro effetti nel mondo, eventualmente, ahimé, arrecando danno a qualcuno, e l'insidia può nascondersi proprio nel passaggio da un livello d'astrazione all'altro.

Partiamo da un esempio. S'immagini di avere un programma che calcola la somma di due numeri. Per fissare le idee lo si può pensare scritto in C (il linguaggio di programmazione inventato da Dennis Ritchie...): probabilmente potrebbe contenere un'istruzione tipo `return x + y`. Chi volesse accertarsi del fatto che il programma fa effettivamente quello per cui lo vogliamo usare (fare le somme) potrebbe, conoscendo il C, basarsi sull'esame di questa istruzione (o, ancora meglio, dell'intero programma) per concludere che sí, il sistema in effetti fa le somme. Diciamo subito che questa sarebbe un'analisi di sicurezza certamente migliore rispetto a prendere il sistema come una *black box* e semplicemente provarlo per vedere se 'funziona' come ci si aspetta: infatti, se `x` e `y` fossero due interi a 64 bit, ci toccherebbe fare $(2^{64})^2 = 2^{128} \approx 3 \cdot 10^{38}$ prove per essere sicuri che il sistema, utile sommatore in tutti gli altri casi, non risulti un insidioso *ransomware* nel caso venga applicato alla coppia 'della morte' $666 + (-42)$.

Sfortunatamente, però, nemmeno l'analisi del sorgente C ci può levare tutti i sospetti. Infatti, ciò che viene effettivamente eseguito non è il codice C, ma una sua *traduzione* ottenuta tramite un *compilatore* che produce il codice macchina da dare in pasto (probabilmente tramite un sistema operativo, che qui trascuriamo) al processore. Tutto sommato, non è troppo difficile immaginare che anche il compilatore è potenzialmente sovvertibile da un attaccante, e che quindi potrebbe tradurre malamente la *back-door* $666 + (-42)$. Se fosse solo questo, in fondo, basterebbe analizzare il programma *e il compilatore*[2].

Ma Thompson ci ha chiarito una volta per tutte che non basta nemmeno considerare la coppia (sorgente + compilatore). Il compilatore, infatti, è frutto esso stesso di una traduzione. Ricapitoliamo: la semantica di un programma P scritto in un linguaggio di programmazione è definita dal suo traduttore K. Se K definisce la 'somma' in una maniera diversa da quella dei *Principia Mathematica* per servire gli scopi di un attaccante digitale, non basta analizzare il programma P per accorgersene, bisogna studiarsi il sorgente di K e P. Ma anche K è un programma ottenuto con una traduzione di un compilatore che chiamiamo $K^\dagger$. Ed ecco il problema: K e P potrebbero a questo punto contenere solo una operazione di somma cui diventa difficile attribuire comportamenti dannosi; però in realtà la somma in P (apparentemente innocua) verrebbe tradotta nella somma in K (apparentemente innocua) che potrebbe essere stata sovvertita in $K^\dagger$ (dannosa)! Cosí chi avesse a disposizione solo K e P e i rispettivi codici sorgente non ha un modo semplice di accorgersene[3].

L'attacco *trusting trust* si presta molto bene per creare un ecosistema di piattaforme vulnerabili, per esempio con obiettivi di sorveglianza di massa. Si immagini un attaccante con fini di questo tipo: chiamiamolo NSA per *No Such*

[2] O il codice macchina, accettando però in generale una complessità del sistema maggiore di vari ordini di grandezza.

[3] Di nuovo, escludendo la possibilità di analizzare il codice macchina di K e P.

Attacker, visto che stiamo parlando in termini del tutto ipotetici[4]. NSA non dovrebbe far altro che identificare un compilatore *open source* di grande diffusione, per esempio il GCC[5]. La disponibilità dei sorgenti di GCC è una garanzia per tutti... O no? O potrebbe diventare proprio il cavallo di Troia che ci porta gli Achei in casa? Qualora NSA riuscisse a distribuire un GCC† in un buon numero di sistemi, grazie magari a qualche distributore compiacente o inconsapevolmente collaborativo, saremmo in una situazione assai spiacevole. Analizziamo la situazione di un utente (salutarmente paranoico) di un sistema qualsiasi in cui sia possibile utilizzare GCC: diciamo per esempio una piattaforma Debian GNU/Linux[6]. Potrebbe usare il GCC fornito dalla distribuzione (chiamiamolo GCC*), ma dovrebbe fidarsi di Debian e del fatto che il distributore non abbia — magari in perfetta buona fede — ottenuto il GCC che distribuisce compilandolo con un GCC† (o un suo 'discendente' altrettanto compromesso). Oppure potrebbe analizzare il codice sorgente di GCC e magari compilarselo in proprio, ma dovrebbe essere sicuro di non usare un GCC†. E il problema si ripropone per tutti i programmi che volesse compilare con il GCC* di cui dispone.

Possiamo difenderci dagli attacchi *trusting trust?* Sí, ma come spesso accade la protezione ha costi e cautele organizzative non indifferenti. La prima soluzione (diversa dallo scriversi i propri compilatori in linguaggio macchina) è stata suggerita nel 1998 da Henry Spencer e verificata formalmente da David A. Wheeler nella sua tesi di dottorato del 2009[7]. La tecnica prende il nome di *Diverse Double-Compiling* e necessita di due compilatori, che chiameremo K_1 e K_2. Non è strettamente necessario che K_1 e K_2 siano perfettamente fidati, è sufficiente che almeno non siano stati compromessi nello stesso modo... A questo punto si prende il codice sorgente di un (terzo) compilatore, potrebbe essere il sorgente del GCC che abbiamo analizzato e trovato convincentemente privo di indizi di sabotaggio, chiamiamolo $\mathcal{S}$, e si segue lo schema di Figura 1, compilandolo *quattro* volte: seguendo il diagramma con calma dovrebbe risultare evidente che K^* e K^{**} sono il prodotto dello stesso compilatore partito dallo stesso codice sorgente. In altre parole, se per caso fossero diversi, le differenze devono essere state introdotte da K' e K'' che però dovrebbero essere lo stesso programma, a meno che non abbiano 'ereditato' qualche compromissione.

Chi vuole dare garanzie che la trasparenza che sbandiera (la disponibilità del codice sorgente) non è un cavallo di Troia dovrebbe perciò considerare tecniche di *Diverse Double-Compiling*. E in effetti progetti particolarmente sensibili a questo tipo di problemi come Tor[8] o Bitcoin[9] si stanno da tempo muovendo in questo senso. Da un paio d'anni anche i distributori di *software* libero generico hanno iniziato ad affrontare il problema: Debian, per esempio, sta sperimen-

[4]Pare che — probabilmente in maniera altrettanto ipotetica — negli Stati Uniti abbiano considerato la possibilità di infiltrare gli strumenti di sviluppo Apple: `https://edwardsnowden.com/2015/03/10/strawhorse-attacking-the-macos-and-ios-software-development-kit/`

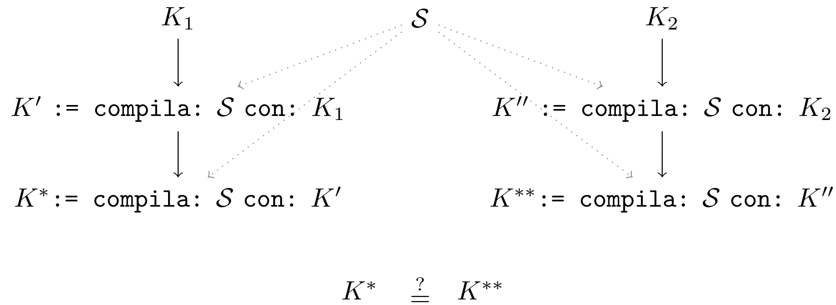
[5]`https://gcc.gnu.org/`

[6]`https://www.debian.org/`

[7]`http://www.dwheeler.com/trusting-trust/`

[8]`https://www.torproject.org/`

[9]`https://bitcoincore.org/`



$$K^* \stackrel{?}{=} K^{**}$$

Figura 1: *Diverse Double-Compiling*

tando[10] la distribuzione di pacchetti binari *riproducibili:* l'utente può provare a ricompilarne i sorgenti su un sistema diverso e verificarne l'integrità bit a bit. È un'impresa tutt'altro che banale, soprattutto perché in molti casi i programmi non sono scritti con l'obiettivo esplicito di essere compilati in maniera *deterministica* cioè producendo sistematicamente, a parità di compilatore (e ambiente di compilazione), sempre la stessa sequenza di bit: si pensi per esempio al caso in cui il programma contenga una stringa con la data di compilazione ('*timestamp*') o l'influenza che possono avere dettagli apparentemente irrilevanti come le date di creazione dei *file*. Debian dichiara[11] di riuscire a riprodurre deterministicamente piú dell'80% dei pacchetti che distribuisce: un risultato eccezionale, per quanto preliminare... Ancora una volta il *software* libero dimostra una cura per le problematiche di sicurezza che il mondo proprietario trascura: del resto è proprio la disponibilità del codice sorgente che contemporaneamente abilita e provoca queste verifiche incrociate.

Insomma una nuova conferma del fatto che non si può mai dormire tranquilli: una volta imparato a diffidare della *dezinformatsiya* che ci circonda, ricordiamoci di stare attenti pure alla *glasnost*!

[10] https://wiki.debian.org/ReproducibleBuilds/About
[11] https://tests.reproducible-builds.org/reproducible.html