ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

VERIFICA DELL'INTEGRITÀ DI SISTEMI SOFTWARE

Elaborato in
PROGRAMMAZIONE DI RETI

Relatore
GABRIELE D'ANGELO

Presentata da
IVAN GORBENKO

Terza Sessione di Laurea
Anno Accademico 2015 – 2016

PAROLE CHIAVE

Sicurezza Informatica

Integrità software

Pacchetti software

Debian

Tails

Alla mia famiglia, ai miei compagni e a tutti quelli che mi hanno sempre sostenuto in questo percorso

Indice

Capitolo 1

Il probema dell'integrità

Questo capitolo ha lo scopo di definire il problema che verrà affrontato nell'elaborato, cercando di non andare nel dettaglio, ma esporre il problema in termini generali.

1.1 Un problema di sicurezza informatica

La sicurezza informatica si basa sulla definizione e applicazione di tre concetti e obbiettivi fondamentali conosciuti anche come triade CIA [1] (Confidentiality, Integrity e Availability):

- **Confidenzialità**: applicabile in due modi differenti:
  - **Confidenzialità dei dati**: la certezza che dati privati e, appunto confidenziali, non siano accessibili o vengano divulgati a individui non autorizzati.
  - **Privacy**: l'individuo ha il controllo completo su, o perlomeno è a conoscenza di, quali informazioni vengano raccolte e memorizzate su di esso, così come chi è autorizzato a raccoglierle e a chi possono essere divulgate.
- **Integrità**: applicabile in due modi differenti:
  - **Integrità dei dati**: assicura che la variazione delle informazioni e dei programmi avvenga solo in una maniera specificata e autorizzata.
  - **Integrità del sistema**: assicura che il sistema si comporti nel modo previsto e in una maniera inalterata, esente da manipolazioni non autorizzate, sia intenzionali che involontarie.

- **Disponibilità**: assicura che il sistema funzioni correttamente e il servizio non sia negato agli utenti autorizzati.

L'obbiettivo di questo elaborato sarà produrre uno strumento in grado di rilevare una perdita di integrità. Gli autori di [1] definiscono una perdita di integrità come una modifica o distruzione non autorizzata di informazioni. È interessante notare che le due sottodefinizioni di integrità possono essere strettamente connesse. L'integrità dei dati comprende anche l'integrità dei programmi, e, un programma alterato, può comportare la perdita dell'integrità del sistema. I due sottoconcetti sono dipendenti tra loro. Lo scopo dell'elaborato quindi sarà rilevare un'eventuale perdita di integrità dei programmi.

1.1.1 Conseguenze della perdita dell'integrità

Un attacco all'integrità di un sistema può comportare conseguenze disastrose che possono rimanere inavvertite dagli utenti. I programmi su cui ogni giorno facciamo affidamento, possono contenere errori di programmazione che lasciano spazio a vulnerabilità, potenzialmente sfruttabili per eseguire codice arbitrario e manipolare il sistema. Un potenziale attaccante potrebbe manipolare il sistema in modo tale da costruire un accesso nascosto che può essere usato per rubare dati degli utenti, sfruttare risorse di calcolo, rete e molto altro. In generale un attacco all'integrità può portare un sistema in uno stato in cui l'utente non è consapevole dell'attacco e l'attacco può estendersi a lungo nel tempo.

1.2 Il caso di studio

Come caso di studio verrà presa una distribuzione derivata da Debian GNU/Linux [3]: Tails [2]. Questa distribuzione ha come obbiettivo principale di mantenere l'anonimità dell'utente, sfruttando diverse tecniche, tra cui la più interessante è incorporata nel suo nome: The Amnestic Incognito Live System. Gli utenti rimangono al sicuro grazie al fatto che il sistema operativo viene utilizzato attraverso un CD/DVD oppure una chiavetta USB in modalità live, ovvero l'immagine del sistema operativo inizialmente viene copiata su un supporto removibile, poi Tails verrà avviato dal supporto removibile. Il sistema operativo viene caricato nella memoria centrale del computer sul quale viene avviato, ed è appositamente pensato per non avere memoria delle sessioni.

In pratica la distribuzione cerca di offrire sicurezza per default limitando ciò che l'utente può fare, cercando di avvertirlo in caso di eventuali azioni che possono essere un rischio per la sua sicurezza, come l'utilizzo di memoria di

massa del computer sul quale si lavora. Questo tipo di approccio che offre un insieme minimale e limitato di funzionalità, accuratamente preconfigurato, ha lo scopo di dare la possibilità agli utenti meno esperti di rimanere al sicuro senza il bisogno di complesse configurazioni e di evitare di compromettersi compiendo azioni inconsapevoli. Tuttavia il problema nasce dal fatto che se installato su una chiavetta USB, che è un supporto riscrivibile al contrario dei CD/DVD (non RW), è necessaria l'assoluta garanzia di sicurezza fisica, ovvero la certezza che i contenuti della chiavetta USB non siano stati manipolati o sovrascritti da un eventuale attaccante con lo scopo di compromettere l'utente. Perciò l'obbiettivo finale è dare all'utente un modo per verificare che i software, presenti in Tails che l'utente utilizzerà, siano originali e non modificati in alcun modo.

Capitolo 2

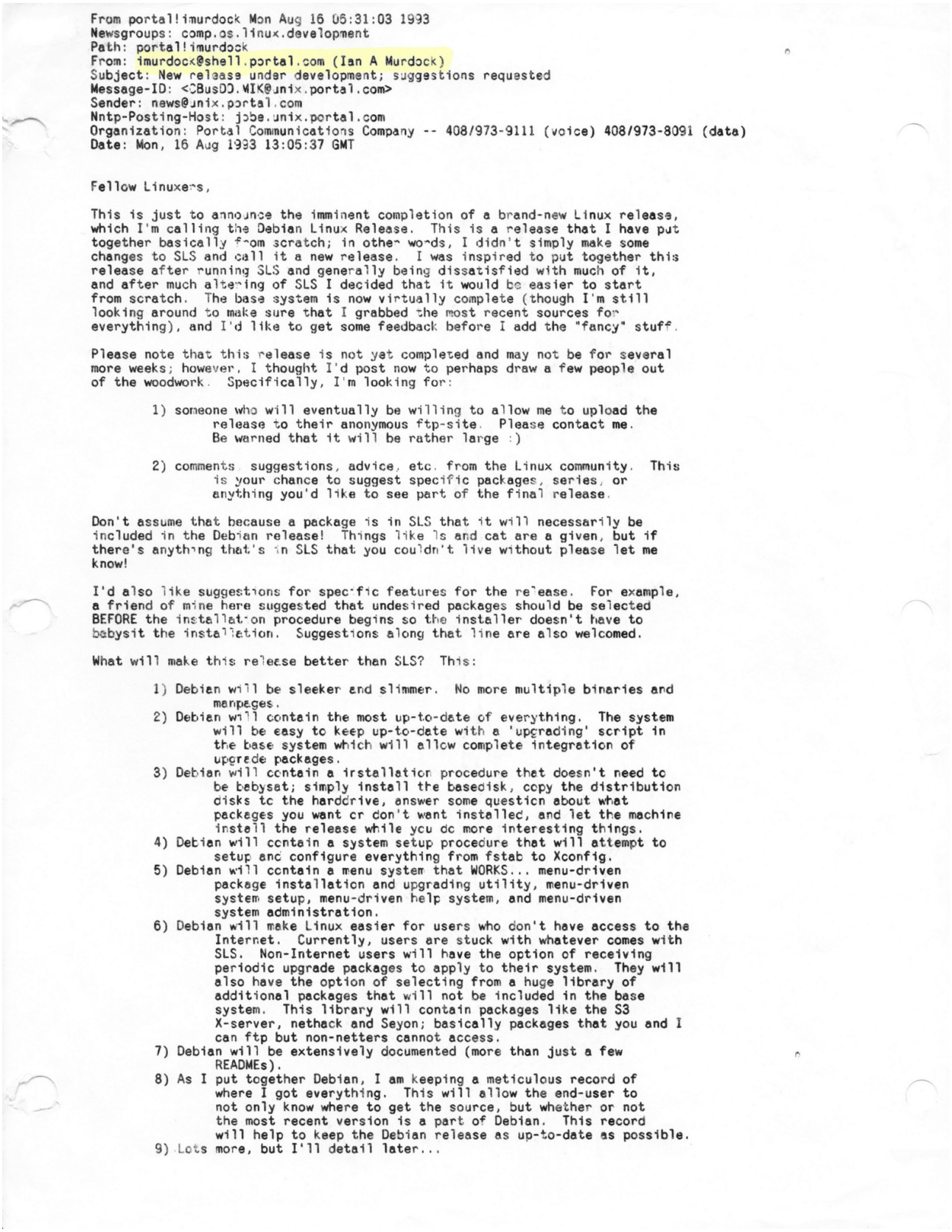
Sistema Operativo Debian

In questo capitolo verranno descritte alcune caratteristiche interessanti del sistema operativo Debian, in quanto Tails è derivato da Debian, quindi condivide la maggior parte delle caratteristiche con esso. Molte informazioni a seguire provengono da un libro [4] scritto da persone che hanno partecipato attivamente allo sviluppo di Debian, che quindi hanno deciso di condividere la propria esperienza documentando il progetto.

2.1 Nascita e filosofia di Debian

Debian nasce come progetto open source nel 1993 da Ian Murdock, ed ha come scopo principale offrire un sistema operativo libero. Murdock, spinto dall'insoddisfazione di come operava l'allora famosa distribuzione Softlanding Linux System (SLS), al posto di prendere come base SLS, decise di costruire un sistema operativo che utilizzasse il kernel Linux [6] e i software del progetto GNU [7], completamente da zero e pubblicò l'annuncio della nascita di Debian, visibile in Figura 2.1.

Il creatore del progetto decise che il sistema operativo doveva fornire ai propri utenti software libero e non commerciale ma che fosse di una qualità tale da competere con quelli commerciali, perciò decise di rendere aperto il processo di sviluppo di Debian. Grazie al fatto che ogni componente sviluppato è software libero, molti sviluppatori possono revisionare il lavoro fatto da altri e migliorarlo. Un'altra caratteristica fondamentale di Debian è che da un ruolo attivo ai suoi utenti, che segnalano i bug e fanno richieste di funzionalità nuove. Queste volontà alla base del progetto, furono espresse dal fondatore del progetto nel *Manifesto Debian* [5]. Ciò che contraddistingue particolarmente Debian è che lo sviluppo non è spinto dalle prospettive di un profitto, ma dalla volontà di fornire software di qualità, al contrario di sistemi operativi commerciali in competizione tra loro, costretti a date dettate dal mercato.



From portal!imurdock Mon Aug 16 06:31:03 1993
Newsgroups: comp.os.linux.development
Path: portal!imurdock
From: imurdock@shell.portal.com (Ian A Murdock)
Subject: New release under development; suggestions requested
Message-ID: <CBusDD.MIK@unix.portal.com>
Sender: news@unix.portal.com
Nntp-Posting-Host: jobe.unix.portal.com
Organization: Portal Communications Company -- 408/973-9111 (voice) 408/973-8091 (data)
Date: Mon, 16 Aug 1993 13:05:37 GMT

Fellow Linuxers,

This is just to announce the imminent completion of a brand-new Linux release, which I'm calling the Debian Linux Release. This is a release that I have put together basically from scratch; in other words, I didn't simply make some changes to SLS and call it a new release. I was inspired to put together this release after running SLS and generally being dissatisfied with much of it, and after much altering of SLS I decided that it would be easier to start from scratch. The base system is now virtually complete (though I'm still looking around to make sure that I grabbed the most recent sources for everything), and I'd like to get some feedback before I add the "fancy" stuff.

Please note that this release is not yet completed and may not be for several more weeks; however, I thought I'd post now to perhaps draw a few people out of the woodwork. Specifically, I'm looking for:

1) someone who will eventually be willing to allow me to upload the release to their anonymous ftp-site. Please contact me. Be warned that it will be rather large :)

2) comments, suggestions, advice, etc. from the Linux community. This is your chance to suggest specific packages, series, or anything you'd like to see part of the final release.

Don't assume that because a package is in SLS that it will necessarily be included in the Debian release! Things like ls and cat are a given, but if there's anything that's in SLS that you couldn't live without please let me know!

I'd also like suggestions for specific features for the release. For example, a friend of mine here suggested that undesired packages should be selected BEFORE the installation procedure begins so the installer doesn't have to babysit the installation. Suggestions along that line are also welcomed.

What will make this release better than SLS? This:

1) Debian will be sleeker and slimmer. No more multiple binaries and manpages.
2) Debian will contain the most up-to-date of everything. The system will be easy to keep up-to-date with a 'upgrading' script in the base system which will allow complete integration of upgrade packages.
3) Debian will contain a installation procedure that doesn't need to be babysat; simply install the basedisk, copy the distribution disks to the harddrive, answer some question about what packages you want or don't want installed, and let the machine install the release while you do more interesting things.
4) Debian will contain a system setup procedure that will attempt to setup and configure everything from fstab to Xconfig.
5) Debian will contain a menu system that WORKS... menu-driven package installation and upgrading utility, menu-driven system setup, menu-driven help system, and menu-driven system administration.
6) Debian will make Linux easier for users who don't have access to the Internet. Currently, users are stuck with whatever comes with SLS. Non-Internet users will have the option of receiving periodic upgrade packages to apply to their system. They will also have the option of selecting from a huge library of additional packages that will not be included in the base system. This library will contain packages like the S3 X-server, nethack and Seyon; basically packages that you and I can ftp but non-netters cannot access.
7) Debian will be extensively documented (more than just a few READMEs).
8) As I put together Debian, I am keeping a meticulous record of where I got everything. This will allow the end-user to not only know where to get the source, but whether or not the most recent version is a part of Debian. This record will help to keep the Debian release as up-to-date as possible.
9) Lots more, but I'll detail later...

Figura 2.1: L'annuncio originale di Ian Murdock sulla prima release di Debian. Immagine da [77]

2.1.1 Documenti fondanti e linee guida del software libero

Debian ad oggi è un progetto dalle dimensioni colossali ed offre decine di migliaia di pacchetti software pronti all'uso. Una crescita di questa proporzione è stata possibile grazie a delle rigide regole ed un'infrastruttura di supporto allo sviluppo che permettono di estendere il progetto con l'aggiunta di nuove funzionalità, mantenimento e correzione delle funzionalità già presenti, e tutto questo senza distruggere il lavoro già fatto.

Oltre all'originale manifesto Debian, un altro cruciale documento influenza in maniera decisiva il progetto: il Contratto Sociale Debian [8], sulla base del quale furono redatte le linee guida Debian per il software libero (DFSG). Il documento fu redatto al fine di sottoscrivere un impegno verso gli utenti e definire quale sarà il corso del progetto. Le stesse linee guida di Debian furono riutilizzate anche da Open Source Initiative per dare *la* definizione di software open source [9]. Il contratto sociale è definito attraverso 5 punti:

1. **Debian rimarrà libera al 100%**:

   Una promessa a mantenere il sistema e tutte le sue componenti liberi, secondo le DFSG. Supporto agli utenti di Debian che useranno sia opere libere che non, ed infine il sistema non diventerà mai dipendente da un componente non libero.

2. **Renderemo alla Comunità Free Software**:

   Una promessa a rilasciare i nuovi componenti, realizzati nel modo migliore possibile, sotto una licenza compatibile con le DFSG in modo tale da favorire la circolazione di opere libere. Le segnalazioni di bug, migliorie e richieste degli utenti saranno segnalate agli autori originali ("upstream") delle opere incluse nel sistema.

3. **Non nasconderemo i problemi**:

   L'intero sistema di segnalazione dei bug sarà aperto al pubblico.

4. **Le nostre priorità sono gli utenti ed il software libero**:

   Mette al centro i bisogni degli utenti e la comunità del software libero. Garantisce il supporto a diverse architetture. Non si vieterà o penalizzerà l'utilizzo o la creazione di opere non libere. Sarà possibile ad altri di creare distribuzioni che contengano Debian e altre opere. Per realizzare tutto ciò sarà fornito un sistema integrato di alta qualità senza restrizioni legali o limiti di utilizzo. Tutto questo senza richiedere alcun compenso.

5. **Opere che non rispettano i nostri standard free software**:

   Debian dedica risorse anche per i software che non sono conformi alla DFSG ed offre le aree "contrib" e "non-free" a queste opere in modo da permettere agli utenti di usarle. Chi intende redistribuire Debian sotto forma di CD, deve verificare che le rispettive licenze lo permettano.

2.1.2 Debian Policy

Al fine di mantenere le promesse fatte e riuscire a gestire l'immane quantità di lavoro richiesto per mantenere il progetto, sono indubbiamente necessarie delle regole. Da questo punto di vista il documento di riferimento è la *Debian Policy* [10], un documento che ha come principale scopo di descrivere l'interfaccia al sistema di gestione dei pacchetti che ogni pacchetto deve rispettare per poter entrare a far parte del progetto.

Il manuale documenta anche le specifiche di design di alcune parti del sistema operativo come la conformità a standard conosciuti come ad esempio il *Filesystem Hierarchy Standard* [11], gli utenti e i gruppi riservati ad uso del sistema, l'interfaccia per la gestione degli script di start/stop dei demoni [1], l'interfaccia per utilizzare cron [13], l'interfaccia per i menu grafici che le applicazioni possono usare per aggiungersi a tali menu e altre informazioni utili agli sviluppatori.

In generale il documento definisce una serie di regole che devono essere rispettate per mantenere una omogeneità non solo nella gestione dei pacchetti ma anche come questi pacchetti devono comportarsi e quali assunzioni possono o non possono fare o su quali servizi o funzionalità offerte dal sistema operativo possono contare. Senza un documento di questo tipo, la manutenzione dei pacchetti diventerebbe ingestibile, in quanto ogni sviluppatore potrebbe inventarsi il proprio modo di gestire una certa funzionalità al posto di utilizzarne una già implementata e standard.

2.1.3 Modello di sviluppo

La distribuzione del software disponibile per Debian avviene attraverso pacchetti. Ogni pacchetto prima di arrivare ad essere utilizzato da un utente finale, deve percorrere una serie di passi necessari per garantire la qualità del software. Le figure maggiormente responsabili di questi passi sono gli sviluppatori, o più correttamente *manutentori* Debian. Per garantire la qualità di ogni software, Debian mantiene contemporaneamente 3 o 4 versioni differenti

[1] Un demone è un processo che rimane in esecuzione a lungo, spesso dall'avvio del sistema fino al suo termine. I demoni vengono eseguiti in background e non hanno un terminale controllante [12]. Un esempio è *cron*.

di questo, ed ogni versione corrisponde a un particolare stadio di sviluppo [4] con un repository dedicato. Gli stadi più interessanti sono *Unstable*, *Testing* e *Stable*. Ogni rilascio ufficiale del sistema operativo è composto da pacchetti provenienti da stable che appunto dovrebbero essere stabili e privi di gravi bug.

Partendo dal principio, un manutentore ha il compito di prendere le versioni originali dei progetti, eventualmente contattare lo sviluppatore a monte del progetto in caso di non conformità alle DFSG e perlomeno tentare di convincerlo a conformarsi a tali regole, successivamente impacchettare i sorgenti del progetto in formato opportuno definito dalla Debian Policy, o semplicemente pacchetto sorgente. Una volta che i sorgenti sono nel formato corretto, a partire da questo il manutentore può compilare sul proprio computer il pacchetto in formato binario, quello necessario per installare ed utilizzare il software. Da qui il manutentore deve caricare sia il pacchetto sorgente che binario sul server Debian `ftp-master.debian.org`, nel repository unstable. Una volta che il pacchetto è reso disponibile in unstable, questo viene propagato a tutti i mirror Debian che si sincronizzano ogni 6 ore e rendono il pacchetto disponibile agli utenti. Visto che i manutentori possono compilare i pacchetti solo per la propria architettura, per risolvere questo problema e automatizzare una grande quantità di lavoro, Debian si avvale di una rete di autobuilder chiamata buildd [14]: un progetto nato con lo specifico compito di rendere disponibili i pacchetti per tutte le architetture supportate e velocizzarne la distribuzione automatizzando la compilazione a partire dai pacchetti sorgente. Quindi una volta che il manutentore carica il pacchetto in unstable, da il via alla compilazione automatica per tutte le architetture possibili, e nel frattempo gli utenti provano il software e segnalano i bug al manutentore del pacchetto, il quale dovrebbe risolvere il problema, lui stesso o possibilmente l'autore originale, e ricaricare una nuova versione in unstable. Si spera che dopo una serie di cicli all'interno di unstable, il pacchetto sia maturato e reso privo dei bug più gravi.

Testing è lo stadio successivo ad unstable e si suppone che i pacchetti instabili prima o poi migrino a questo stadio. La migrazione dei pacchetti avviene in automatico sotto una serie di condizioni di base come la compilazione con successo in tutte le architetture e l'assenza di modifiche recenti, questo perché si suppone che ogni modifica indipendentemente dal suo scopo introduca dei nuovi bug e ovviamente ad ogni modifica è necessario ricompilare il pacchetto per tutte le architetture. Altre condizioni più difficili da soddisfare sono necessarie per migrare in testing, tra cui l'assenza di bug critici o almeno di gravità inferiore alla versione attualmente in testing. È presente anche una condizione temporale di 10 giorni di permanenza in unstable: il tempo necessario a trovare e segnalare i bug. Si suppone che se in 10 giorni il pacchetto non subisce modifiche o segnalazioni, questo non contenga bug. L'ultima condizione da

soddisfare è la più complessa: è necessario che le dipendenze del pacchetto in unstable, pronto a migrare in testing, siano tutte soddisfatte in testing. Da qui sorge un problema significativo perché un pacchetto potrebbe essere pronto per l'uso e privo di bug significativi, ma non lo sono le sue dipendenze e quindi ne bloccano la migrazione se non è già disponibile una versione di queste in testing. Per risolvere questo problema, le migrazioni avvengono in gruppi perché ogni pacchetto deve sempre riuscire a soddisfare le sue dipendenze all'interno di testing. A questo punto il pacchetto in testing dovrebbe essere utilizzabile e privo dei bug più gravi, anche se questo non è sempre vero.

Stable è infine l'ultimo stadio del pacchetto. Ogni pacchetto una volta che raggiunge la prima volta testing, si suppone che sia più o meno maturo e quindi le successive modifiche, che passano sempre attraverso unstable, giungono più velocemente a testing perché i problemi più gravi si suppongono risolti e le sue dipendenze sono soddisfatte, quindi dopo una serie di cicli il pacchetto dovrebbe raggiungere lo stato dell'arte. Da testing il prossimo passo per il pacchetto è l'inclusione in una distribuzione stable il cui rilascio avviene ogni 2 anni circa. Ogni rilascio di una versione stable non è altro che una copia in dato momento di testing. Normalmente i rilasci avvengono quando testing è privo di bug critici conosciuti e quando il programma di installazione per la nuova versione del sistema operativo è pronto. Il Release Team [15] di Debian ha il compito di stabilire un calendario per il nuovo rilascio. A un certo punto, in prossimità di un nuovo rilascio, testing viene congelato e per aggiornare i pacchetti durante questo periodo è necessaria un'approvazione da parte dei *Release Manager*. Lo scopo del congelamento è evitare nuove versioni dei pacchetti che introducono nuovi bug e gli unici aggiornamenti possibili sono di correzione dei bug attuali. I manutentori dei pacchetti quindi devono sottostare al calendario stabilito dal release team e correggere prontamente i propri pacchetti se vogliono che questi siano inclusi nel prossimo rilascio, altrimenti verranno rimossi per evitare di fornire software difettoso agli utenti finali. Dal momento che viene rilasciata la nuova versione stabile, i pacchetti che la compongono si suppongono virtualmente perfetti, tuttavia alcuni problemi possono sempre sorgere e da qui è compito degli *Stable Release Manager* di gestire revisioni successive del rilascio, fornendo gli aggiornamenti di sicurezza e in caso di gravi problemi con i pacchetti, approvarne gli aggiornamenti.

È importante notare che può volerci più tempo per un nuovo rilascio di quanto stabilito dal calendario in base a dove i bug vengono trovati. Chiaramente se il pacchetto afflitto dal bug è un componente essenziale del sistema operativo ed è imperativo correggerlo, il rilascio viene ritardato. Nel caso di un pacchetto non importante, questo viene semplicemente escluso se non corretto in tempo.

Questo lungo processo e insieme di regole è ciò che rende Debian famoso

per la sua stabilità e lo rende utilizzabile in ambienti di produzione. Tuttavia per i lunghi tempi tra un rilascio stabile e un altro, alcuni utenti preferiscono utilizzare la versione testing perché desidera avere prima le funzionalità nuove dei pacchetti, e spesso è un buon compromesso tra novità e stabilità. Gli utenti più esperti e gli sviluppatori invece potrebbero preferire unstable perché magari preferiscono aggiornamenti più frequenti rispetto a testing e vogliono sempre l'ultima versione a monte, al prezzo di bug più significativi, che però in qualità di utenti esperti segnaleranno ai manutentori. Qualunque distribuzione venga scelta, l'utente ha sempre un ruolo attivo perché è grazie alle sue segnalazioni che è possibile fornire software di qualità e rendono il lavoro dei release manager più semplice in quanto questi sperano che al momento del congelamento di testing, almeno i problemi più banali siano già risolti. Lo schema in Figura 2.2 mostra i punti chiave del processo di sviluppo di un software in Debian.

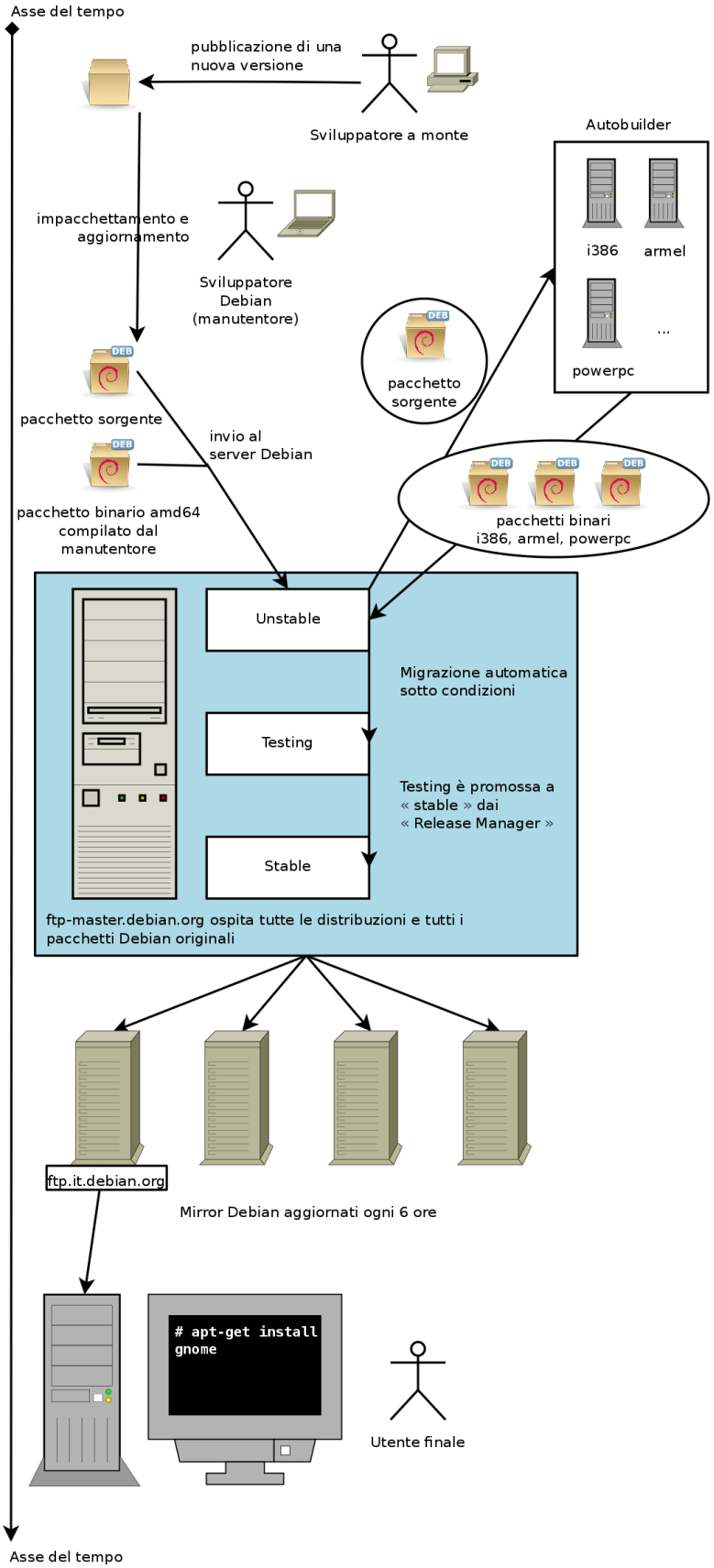


Figura 2.2: Il lungo percorso di un software attraverso il processo di sviluppo Debian. Immagine da [4]

Capitolo 3

Nozioni di base

Al fine di rendere chiara la trattazione degli argomenti nei capitoli successivi, è necessario definire alcuni concetti di base.

3.1 Crittografia

La crittografia è un insieme di tecniche utilizzate per trasformare dei dati in un formato intelleggibile in un altro formato non intelleggibile attraverso l'utilizzo di algoritmi crittografici. La crittografia può essere utilizzata per i seguenti scopi:

- **Confidenzialità**: solamente gli individui autorizzati possono accedere alle informazioni
- **Integrità**: in caso di alterazione dell'informazione, è possibile rilevarla
- **Autenticazione**: il destinatario di un messaggio può verificare l'identità del mittente
- **Non ripudio**: il mittente di un messaggio non può negare di essere il mittente di tale messaggio

3.1.1 Tipi di crittografia

Le tipologie fondamentali di crittografia sono:

- **crittografia simmetrica**: i dati vengono criptati e decriptati con la stessa chiave
- **crittografia asimmetrica**: si utilizza una coppia di chiavi legate tra loro

- **funzioni crittografiche di hash**: un tipo di crittografia unidirezionale che mappa dei dati di lunghezza variabile in una stringa di bit a lunghezza fissa

Il primo tipo utilizza la stessa chiave per criptare e decriptare i dati ed in questo caso se si vuole costruire un canale sicuro per scambiarsi messaggi, è necessario che le entità in comunicazione utilizzino inizialmente un altro canale sicuro per ottenere la chiave segreta usata per cifrare le comunicazioni. La crittografia asimmetrica invece utilizza una coppia di chiavi, di cui una si rende disponibile pubblicamente mentre l'altra mantenuta privata, quella pubblica viene utilizzata per cifrare mentre quella privata per decifrare. Infine le funzioni hash sono utilizzate per produrre un'impronta digitale dei dati in input, nel caso in cui anche un solo bit dell'input cambi, l'hash corrispondente sarà diverso.

3.1.2 Funzioni crittografiche di hash

Una funzione crittografica di hash è un algoritmo che mappa dei dati di lunghezza arbitraria in una stringa di bit a lunghezza fissa, ed è progettata per essere unidirezionale, ovvero non invertibile. Una funzione hash può essere utilizzata per produrre un'impronta digitale di un file, un messaggio o un qualunque tipo di dato. Una funzione di hash deve soddisfare i seguenti requisiti [1]:

- la funzione deve essere applicabile a un blocco di dati di qualunque lunghezza
- la funzione produce un output di lunghezza fissa
- per un qualunque output della funzione delle essere computazionalmente infattibile trovare l'input che ha generato tale output (unidirezionalità)
- per un qualunque input deve essere computazionalmente infattibile trovare un altro input tale che gli output di entrambi siano uguali (resistenza debole alle collisioni)
- deve essere computazionalmente infattibile trovare una qualunque coppia di input tale che l'output di entrambi siano uguali (resistenza forte alle collisioni)

Le funzioni hash attualmente esistenti sono molte, tuttavia con il continuo aumento della potenza computazionale dei moderni processori, alcune di queste non riescono più a soddisfare le condizioni sulle collisioni. In particolare le funzioni della famiglia *Message Digest* (MD) versione 4 [16] e 5 [17], sono

state provate non più resistenti alle collisioni. Dei ricercatori [18] sono stati in grado di trovare delle collisioni di MD5 in un tempo compreso tra i 15 minuti e un'ora, mentre con la versione 4 dell'algoritmo il tempo necessario è stato meno di una frazione di secondo. Visto le problematiche della famiglia di funzioni MD, queste non possono più essere usate per verificare l'integrità dei dati.

Dalla necessità di avere delle funzioni hash sicure che garantiscano la resistenza alle collisioni, nacquero le funzioni Secure Hash Algorithm (SHA). Esistono 5 diverse funzioni tra cui SHA-1, SHA-224, SHA-256, SHA-384 e SHA-512, la prima produce un output di 160 bit mentre le altre comunemente raggruppate con il nome di SHA-2 hanno un output di dimensione corrispondente a quello nel proprio nome. Attualmente non ci sono attacchi reali su SHA-1, tuttavia ci sono studi teorici [19] che dimostrano la possibilità di diminuire la complessità dell'algoritmo per trovare delle collisioni. Alla luce di questi studi, aziende tecnologiche produttrici di browser web con l'intento di anticipare la rottura di SHA-1, hanno annunciato che a partire dal 2017 i certificati SSL firmati con SHA-1 non verranno più considerati sicuri per timore di avversari capaci di grandi investimenti al fine trovare collisioni.

3.1.3 Crittografia a chiave pubblica

La crittografia a chiave pubblica risolve il problema dell'iniziale scambio di una chiave segreta. Ogni utente possiede una coppia di chiavi tra cui ne sceglie una da utilizzare come chiave pubblica che rende ben nota a coloro che desiderano comunicarci, mentre l'altra chiave diventa la sua chiave privata e questa va mantenuta al sicuro. Nel momento in cui un utente desidera inviare un messaggio confidenziale a un altro utente, il mittente recupera la chiave pubblica del destinatario del messaggio e la utilizza per cifrare il messaggio. A questo punto solamente la chiave compagna ovvero la chiave privata è in grado di decifrare il messaggio, garantendone la confidenzialità.

Con la crittografia a chiave pubblica è inoltre possibile ottenere anche l'autenticazione. Quando è necessario verificare l'identità del mittente di un messaggio, questo può utilizzare la propria chiave privata per criptare il messaggio, a questo punto il destinatario o in generale chiunque possieda la chiave pubblica del mittente, userà questa per decifrare il messaggio. Nel caso in cui il messaggio cifrato con la chiave privata del mittente venga decifrato con successo dalla corrispondente chiave pubblica, si ha la certezza che solamente il possessore della chiave privata può aver cifrato il messaggio, e quindi solamente lui può essere il mittente del messaggio. In Figura 3.1 è possibile osservare gli schemi appena descritti.

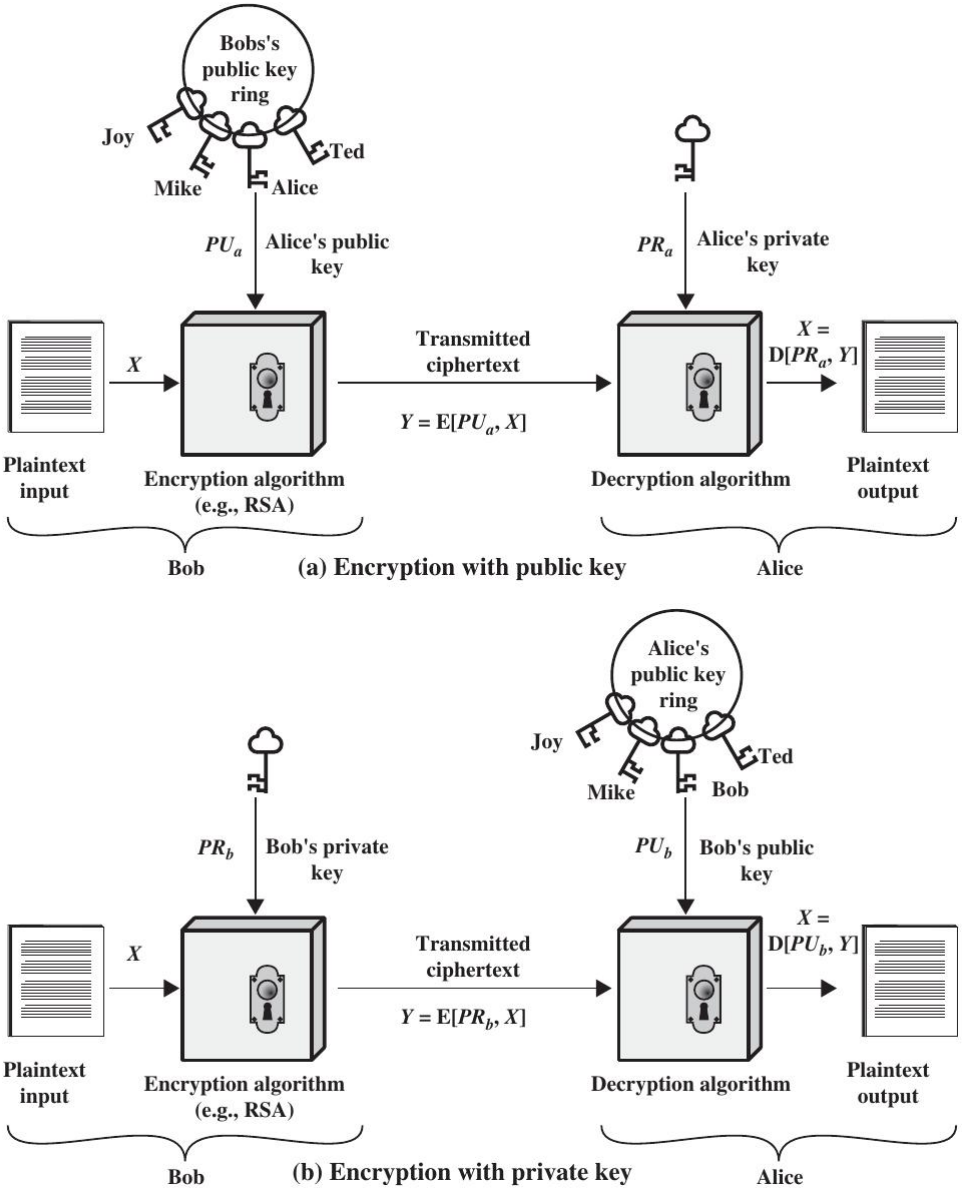


Figura 3.1: Schema crittografia a chiave pubblica [1]

3.1.4 OpenPGP

Combinando le diverse tipologie di crittografia, è stato possibile raggiungere gli scopi inizialmente elencati in maniera efficiente. Philip Zimmernann nel 1991 sviluppò un software chiamato Pretty Good Privacy (PGP) [20], il software che successivamente diede vita ad un protocollo standard specificato

dall'IETF come OpenPGP [21]. Lo standard definisce il formato dei messaggi, gli algoritmi utilizzati e le linee guida per implementare le varie combinazioni possibili di algoritmi crittografici esistenti in modo da rendere semplice l'implementazione di software per la crittografia che possano interoperare senza problemi.

OpenPGP per rendere efficiente la crittografia a chiave pubblica, dispendiosa in termini di computazionali, la combina con la più efficiente ma egualmente sicura crittografia simmetrica. Nella pratica gli schemi in Figura 3.1 non vengono utilizzati direttamente sui messaggi. Per inviare un messaggio in maniera confidenziale OpenPGP utilizza chiavi di sessione, ovvero genera una chiave causale che viene usata per criptare con la crittografia simmetrica il messaggio, mentre la chiave di sessione viene cifrata con la chiave pubblica del destinatario. Quindi il messaggio cifrato è composto dalla chiave di sessione cifrata con la chiave pubblica del destinatario e il messaggio cifrato con la chiave di sessione. Il destinatario del messaggio usa la propria chiave privata per decifrare la chiave di sessione e successivamente utilizza questa per decifrare il messaggio vero e proprio.

Per quanto riguarda l'autenticazione dei messaggi, OpenPGP descrive uno schema di firma digitale senza cifrare il messaggio stesso. Invece di cifrare il messaggio stesso, si utilizza una funziona crittografica di hash per calcolare il valore hash del messaggio, ed è il valore hash del messaggio che viene cifrato con la chiave privata del mittente. Quindi il messaggio digitalmente firmato è composto dal messaggio in chiaro e il valore hash del messaggio cifrato con la chiave privata. A questo punto per verificare l'autenticità del messaggio, si calcola l'hash del messaggio, si decifra l'hash del messaggio con la chiave pubblica del mittente e si verifica l'uguaglianza di questo con quello calcolato sul messaggio, in caso positivo la firma è autentica. La Figura 3.2 mostra lo schema di firma appena descritto.

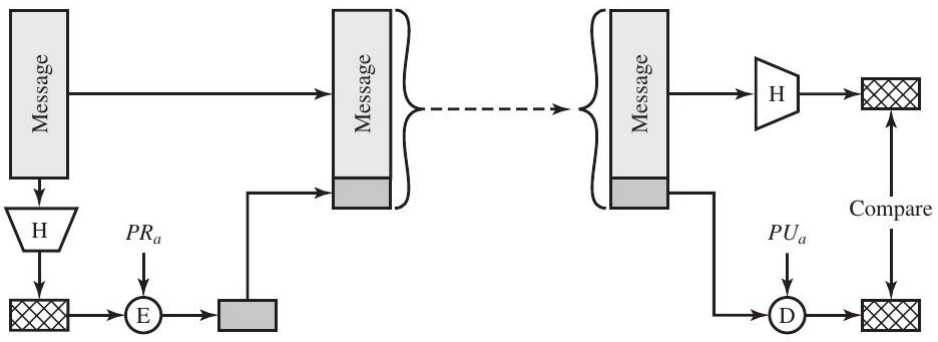


Figura 3.2: Schema di firma e verifica di un messaggio firmato digitalmente [1]

3.1.5 File binario eseguibile

Per file binario eseguibile si intende un file in un formato tale da poter essere eseguito su un computer. Un file eseguibile contiene istruzioni in linguaggio macchina che possono essere eseguite direttamente da un processore. A partire da un insieme di file sorgente scritti in un linguaggio di programmazione di alto livello come il C, attraverso l'utilizzo di un compilatore, questi possono essere tradotti in file binari eseguibili composti da istruzioni macchina. I file eseguibili non sono facilmente comprensibili dagli esseri umani in quanto non sono composti da testo ma codici numerici corrispondenti alle istruzioni macchina ed i dati utilizzati all'interno del programma, necessari per l'esecuzione del file su un computer. I programmi e le librerie condivise vengono compilate in un formato standard chiamato Executable and Linkable Format (ELF), compatibile con i sistemi operativi basati su Linux. Normalmente questo tipo di file rimane invariato per tutto il tempo di permanenza sul sistema, ad esempio dal momento dell'installazione della corrispondente applicazione che lo fornisce fino al momento della sua disinstallazione.

3.1.6 Altri tipi di file eseguibile

Oltre ai tradizionali linguaggi di alto livello che hanno bisogno di essere compilati in linguaggio macchina per permettere l'esecuzione del programma, esistono i cosiddetti linguaggi di scripting. I linguaggi di scripting permettono di scrivere programmi, comunemente chiamati *script*, sotto forma di testo ed invece di compilarli in linguaggio macchina, eseguirli attraverso un programma *interprete* che, a tempo di esecuzione tradurrà le istruzioni contenute nello script nel corrispondente codice macchina eseguibile. Nei sistemi operativi di tipo Unix, gli script sono identificati dalla prima riga che inizia con i caratteri `#!` dopo i quali segue il percorso al programma interprete che dovrà eseguire lo script, ad esempio uno script che deve essere interpretato da una shell Bash [22] conterrà nella prima riga del file `#!/bin/bash` ad indicare che l'interprete da utilizzare è posizionato al percorso `/bin/bash`.

Capitolo 4

Gestione dei pacchetti software

Questo capitolo tratterà di come vengono gestiti i pacchetti software in Debian. Verranno discussi i software che manipolano i pacchetti, il formato dei pacchetti software e come questi sono distribuiti da Debian all'utente finale. Infine si discuterà di quali meccanismi di sicurezza i tool di gestione software implementano per garantire l'autenticità dei pacchetti che manipolano.

4.1 Struttura di un pacchetto Debian

Il sistema operativo Debian per far fronte alla necessità di distribuire in maniera semplice e rendere automatizzabile l'installazione, rimozione ed altre operazioni comuni di gestione delle applicazioni software, definisce un proprio formato che rappresenta la singola unità di applicazione software: il pacchetto binario Debian, degli archivi con un preciso formato e contenuto terminanti con l'estensione .deb. I pacchetti Debian contengono l'insieme di file che compongono il software di per sé e una serie di altri file di controllo che sono necessari agli strumenti di gestione dei pacchetti Debian. In particolare un pacchetto Debian è un archivio di tipo ar [23] contenente 3 elementi con un nome specifico e nel seguente ordine [24]:

1. **debian-binary** è un semplice file di testo contenente la versione del formato del pacchetto, attualmente 2.0

2. **control.tar.gz** è un archivio tar compresso con gzip che contiene i metadati relativi al pacchetto

3. **data.tar** è un archivio tar, opzionalmente compresso con gzip, bzip2 o lzma, contenente tutti i file che il pacchetto installa nel sistema, organizzati nella gerarchia di cartelle che rispecchia i percorsi in cui devono essere installati.

4.1.1 Archivio Control

Come suggerisce il nome, l'archivio control contiene una serie di file di controllo, utilizzati dai software di gestione dei pacchetti. I file di controllo hanno anche loro dei nomi e formati specifici definiti dalla Debian Policy [10]:

control È l'unico file di controllo obbligatorio e contiene le informazioni relative al pacchetto come il nome, la versione, l'architettura del processore per cui il pacchetto è stato compilato e altre informazioni descrittive, inoltre è possibile definire relazioni tra pacchetti, ad esempio un pacchetto per funzionare può dipendere da altri così come essere in conflitto con altri. Le informazioni definite in questo file sono utili ai tool di gestione pacchetti, che come vedremo in seguito sono in grado di risolvere le dipendenze e aiutare l'amministratore di sistema a trovare problemi più facilmente.

conffiles Questo file elenca i file installati dal pacchetto che devono essere considerati come file di configurazione. In quanto opzionale non tutti i pacchetti lo utilizzano, ma la sua utilità è dovuta al fatto che la policy di Debian mette al primo posto la conservazione della configurazione fatta dall'amministratore, in particolare nel momento in cui un pacchetto deve essere aggiornato, il sistema di gestione dei pacchetti preserverà la configurazione fatta, se questa non è quella di default.

{pre,post}{inst,rm} I manutentori dei pacchetti possono aver bisogno di eseguire delle operazioni durante specifici stati del ciclo di vita di un pacchetto. Questi file, nominati in base al momento di invocazione, sono definiti come *script del manutentore* e vengono invocati dal tool di gestione pacchetti. È interessante notare che i tool di gestione dei pacchetti, a meno che non vengano utilizzati solo per visualizzare informazioni senza alterare lo stato del sistema, sono invocati con i permessi di amministratore che a loro volta sono ereditati dagli script del manutentore. Un manutentore deve essere certo di non causare danni con i propri script, infatti la Debian Policy [10] sancisce che questi devono essere idempotenti, ovvero a fronte di più esecuzioni, magari a causa di un errore o interruzione, questi devono verificare che le azioni necessarie siano state eseguite o eventualmente eseguirle in caso negativo.

md5sums È un file di testo contenente gli hash MD5 dei file installati dal pacchetto. Questo file può essere usato per la verifica dell'integrità dei file installati dal package.

shlibs Un file normalmente presente in pacchetti che contengono librerie condivise. Il suo scopo è mappare i nomi e le versioni delle librerie condi-

vise nei corrispondenti nomi di pacchetto della libreria [25]. È utilizzato dai tool di compilazione dei pacchetti binari Debian per automatizzare la costruzione delle dipendenze da librerie condivise nel file control del pacchetto.

symbols Anche questo normalmente fornito da pacchetti che contengono librerie condivise il cui scopo è praticamente lo stesso di shlibs ma descrive i singoli simboli forniti dalla libreria e la corrispondente versione minima contenente il simbolo [26].

config e templates Nel caso in cui un pacchetto debba necessariamente fare domande all'utente per poter essere configurato correttamente, il file config è lo script delegato a farlo. Le domande sono definite attraverso una sintassi specifica e sono descritte nel file templates. La configurazione avviene attraverso un protocollo [27] che permette di semplificare la gestione della configurazione ed evitare che ogni pacchetto implementi il proprio metodo. Debconf [28] è il tool che implementa il protocollo, lo script config e gli script del manutentore interagiscono con questo per fare domande e ottenere le risposte. Le risposte dell'utente vengono salvate in un database mantenuto da debconf per permetterne il riutilizzo in futuro.

4.2 Strumenti di gestione dei pacchetti

Ciò che rende veramente famoso Debian sono i suoi tool di gestione dei pacchetti che permettono di installare, rimuovere e aggiornare applicazioni in maniera facile e intuitiva, cercando di evitare di causare problemi e mantenere il sistema in uno stato sempre consistente. Il tipico utente è abituato a gestire le applicazioni come un'entità unica e non come un aggregato di file che devono essere posizionati nei percorsi giusti, allo stesso modo l'utente finale da per scontato che una volta installata l'applicazione, tutti i file esterni e le librerie condivise di cui ha bisogno siano già presenti. Tutto ciò è stato possibile grazie alla combinazione di diversi strumenti che interagiscono tra loro e automatizzano le operazioni descritte.

4.2.1 Dpkg

Il gestore di pacchetti dpkg [29] è il programma capace di installare applicazioni fornite come pacchetti Debian. È un programma definito di basso livello perché non è in grado di acquisire pacchetti, così come non è in grado di soddisfare le dipendenze dei pacchetti, infatti nel caso in cui le dipendenze

richieste non siano già presenti, questo si rifiuterà di installare il pacchetto in quanto porterebbe il sistema in uno stato instabile.

Dpkg quindi gestisce solamente l'installazione e la rimozione di singoli pacchetti Debian. Per riuscire ad eseguire le sue operazioni senza causare problemi, il programma tiene traccia dei pacchetti installati e perfino di quelli rimossi mantenendo un database sotto forma di file di testo nella cartella */var/lib/dpkg*, in particolare il file *status* contiene lo stato attuale dei pacchetti presenti nel sistema. Quando un pacchetto viene installato, il contenuto del suo file control viene inserito nel file status con l'aggiunta di un campo "*Status*" che rappresenta appunto il corrispondente stato del pacchetto. È da notare che dpkg quando disinstalla un pacchetto mantiene i suoi file di configurazione, assegnando lo stato di "file di configurazione" al pacchetto, in questo modo se il pacchetto viene reinstallato in seguito, è possibile riutilizzarne la configurazione precedente. Per quanto riguarda i file di controllo del pacchetto, dpkg salva tutto il contenuto dell'archivio control nella cartella */var/lib/dpkg/info* nominandoli con il nome del pacchetto e aggiungendogli come estensione il corrispondente nome del file di controllo. Per poter eliminare i file installati da un pacchetto quando questo deve essere disinstallato, a tempo di installazione dpkg genera un file *nome-pacchetto.list* che mantiene insieme ai file di controllo, in cui salva i percorsi in cui i file installati vengono estratti, mappando il pacchetto ai file che lo compongono.

I processi di installazione, configurazione e rimozione seguono delle procedure ben definite che portano i pacchetti a degli altrettanto ben definiti stati. Dpkg con l'utilizzo del formato .deb generalizza questi processi, astraendo dal tipo di applicazione e offrendo un'interfaccia per gestirne il ciclo di vita sul sistema attraverso l'utilizzo degli script del manutentore. Tuttavia, alcune applicazioni possono avere molte interazioni complesse con altre applicazioni e possono esserci molte cause di errore che dpkg deve gestire. Per chiarire come sono legati tra loro gli stati dei pacchetti agli script del manutentore, un manutentore di pacchetti Debian ha descritto i vari processi con dei diagrammi UML [30], visto che effettivamente come notato dall'autore, la descrizione fornita dalla Debian Policy non è molto chiara a riguardo.

4.2.2 Advanced Package Tool

Per offrire un'interfaccia utente di semplice utilizzo e una serie di funzionalità di più alto livello rispetto a dpkg, sono stati sviluppati i tool della famiglia *APT*. Il più famoso tool della famiglia è sicuramente *apt-get* [31] anche se ne esistono diverse alternative visto che le funzionalità principali sono implementate in librerie condivise. Il tool offre la possibilità di installare, aggiornare e rimuovere pacchetti, utilizzando il corrispondente nome. APT è definito di più

alto livello perché è capace di acquisire i pacchetti da repository remoti e locali utilizzando vari protocolli tra cui HTTP, FTP, SSH e altri, ma la funzionalità più interessante e complessa che implementa è la risoluzione delle dipendenze. Nel momento in cui l'utente chiede di installare o aggiornare un pacchetto, APT utilizzerà le informazioni di controllo del pacchetto per costruire l'albero delle dipendenze di tale pacchetto, ripetendo l'operazione ricorsivamente per ogni pacchetto da cui dipende, fino a trovare l'elenco dei pacchetti con le corrispondenti versioni richieste come dipendenze. Una volta calcolate le dipendenze, APT si occuperà di acquisire i corrispondenti file .deb dei pacchetti da installare. A questo punto il compito di installare i file passa a dpkg, ma come descritto in precedenza dpkg non permetterà l'installazione di pacchetti le cui dipendenze rimarrebbero insoddisfatte, quindi anche qui APT prenderà le giuste decisioni per evitare i conflitti e calcolerà l'ordine corretto di installazione dei pacchetti per fare in modo che ogni pacchetto che dpkg dovrà installare abbia le corrispondenti dipendenze già installate. APT spesso installerà pacchetti che sono richiesti solamente come dipendenze da pacchetti che l'utente esplicitamente ha richiesto di installare, e nel momento in cui il pacchetto richiesto dall'utente verrà disinstallato, i pacchetti da cui dipende potrebbero diventare "inutili", per questo APT terrà traccia dei pacchetti installati automaticamente nel file */var/lib/apt/extended_states* e proporrà all'utente di disinstallarli quando il pacchetto che li ha richiesti viene disinstallato.

4.3 Distribuzione dei pacchetti Debian

Debian utilizza un'architettura a repository per distribuire i propri pacchetti agli utenti. I tool APT richiedono che l'utente specifichi delle fonti da cui acquisire pacchetti, ovvero i repository, nel file di configurazione */etc/apt/sources.list* il quale ha uno specifico formato [32] per descrivere le fonti, dove ogni riga rappresenta una fonte ed ogni fonte ha una forma del tipo:
`deb uri distribuzione componenti`.
I tool APT supportano l'acquisizione anche di pacchetti sorgente dai repository e perciò è necessario specificare per ogni fonte se è una fonte di pacchetti binari (deb) o sorgenti (deb-src). L'elemento successivo definisce il corrispondente URI da utilizzare che come detto prima può essere HTTP, FTP ma allo stesso modo anche un percorso locale. La distribuzione invece rappresenta il rilascio di Debian da utilizzare, qui è possibile usare *stable*, *testing* e *unstable* che faranno riferimento alle corrispondenti distribuzioni in quel dato momento, ma normalmente si utilizzano i nomi in codice (attualmente stable=jessie) perché nel momento in cui una nuova distribuzione stabile viene rilasciata, si rischia di avere un sistema che mescola due versioni stabili che può essere fonte

di problemi. L'ultima parte definisce i componenti che si vogliono utilizzare che corrispondono alle aree dell'archivio Debian: *main*, *contrib* e *non-free*. La Debian Policy differenzia le aree tra loro in base alle licenze, le dipendenze e alla conformità con la policy stessa. Main è l'area destinata ai pacchetti che soddisfano le *DFSG* ed hanno dipendenze soddisfatte solamente all'interno di main, mentre i pacchetti in contrib devono comunque soddisfare le DFSG ma hanno dipendenze con pacchetti nello stessa contrib oppure non-free, quindi contrib contiene comunque software libero ma che dipende da software non libero, infine non-free contiene software che non è conforme alle DFSG, ha delle restrizioni sulla distribuzione e non fornisce il codice sorgente, il tipico caso dei firmware proprietari.

Un tipico utente dell'attuale versione stabile, Debian jessie, utilizzerà i seguenti repository nel file sources.list:

aggiornamenti di sicurezza
: `deb http://security.debian.org/ jessie/updates main contrib non-free`
questo repository contiene gli aggiornamenti di sicurezza per i pacchetti in cui vengono scoperte vulnerabilità ed hanno urgenza di essere aggiornati

distribuzione stabile di base
: `deb http://mirror.debian.org/debian/ jessie main contrib non-free`
contiene i pacchetti base della distribuzione, la maggior parte dei pacchetti provengono da questo repository

aggiornamenti importanti alla versione stabile
: `deb http://mirror.debian.org/debian/ jessie-updates main contrib non-free` normalmente i pacchetti della distribuzione stabile vengono aggiornati raramente ma potrebbero venire scoperti dei bug di una gravità sufficiente da aggiornare il pacchetto prima del nuovo rilascio

backport alla distribuzione stabile
: `deb http://mirror.debian.org/debian jessie-backports main contrib non-free` questo repository contiene i pacchetti della prossima distribuzione stable, adattati per funzionare sulla versione stabile attuale per dare la possibilità di usare le nuove funzionalità, senza usare esplicitamente i repository testing e unstable, tuttavia la compatibilità non è garantita

Le fonti elencate dal file sources.list inoltre verranno considerate in base all'ordine di inserimento, ovvero se due fonti offrono lo stesso pacchetto con la stessa versione, verrà usata la fonte elencata per prima, per questo tipicamente le fonti si ordinano dalla più veloce alla più lenta.

Alcuni utenti potrebbero preferire utilizzare solo il componente main che è completamente libero, ma tipicamente sono necessari pacchetti sia da contrib che non-free per avere tutti i dispositivi del proprio computer funzionanti, come nel mio caso i firmware della scheda grafica provengono da contrib mentre quelli della scheda audio da non-free.

4.3.1 Acquisizione e autenticazione dei pacchetti Debian

I client APT utilizzano le fonti specificate nel file sources.list per acquisire una serie di file di metadati dai repository, per i quali è stato definito uno specifico formato [33]. Per verificare l'autenticità dei pacchetti che vengono acquisiti è stata implementata una forma di autenticazione denominata *apt-secure* [34] [35].

Ogni repository descrive il proprio contenuto mediante il file *Release*, il cui percorso è determinato concatenando l'URI della fonte, una cartella implicita `dists`, e la distribuzione. Nel file Release sono contenute diverse informazioni relative alla distribuzione, in particolare a quale versione, suite e nome in codice corrisponde la distribuzione, quali architetture e componenti sono disponibili. Dopo le informazioni descrittive nel file Release, sono elencati gli hash MD5, SHA1 e SHA256 di tutti i file indice presenti sotto quella specifica distribuzione. Sotto ogni distribuzione seguono le cartelle corrispondenti ai componenti della distribuzione, da qui, nelle cartelle di ogni componente, è presente una cartella *binary-<architettura>* per differenziare le varie architetture, e sotto ogni cartella binary si trova un file chiamato *Packages*, solitamente compresso. Il file Packages quindi contiene l'elenco dei pacchetti disponibili di una architettura, di uno specifico componente in una specifica distribuzione. Il contenuto del file Packages è strutturato per paragrafi suddivisi da righe vuote e ogni paragrafo corrisponde alla descrizione di un pacchetto, descritto con il contenuto del suo file *control*. Per ogni paragrafo nel file Packages, che corrisponde al file control di un pacchetto, vengono aggiunti i seguenti campi:

Filename è il percorso relativo rispetto alla radice del repository (la radice corrisponde all'URI della fonte APT), al corrispondente file .deb del pacchetto

Size è la dimensione in byte dell'archivio .deb del pacchetto

MD5Sum, SHA1, SHA256 sono gli hash del pacchetto al percorso specificato da Filename

Per permettere l'autenticazione dei pacchetti, il file Release di ogni distribuzione è accompagnato da un file *Release.gpg*: una firma digitale del file Release

fatta con GNU Privacy Guard o gpg [36], l'implementazione GNU di OpenPGP. Un client APT per poter autenticare i pacchetti di un repository deve avere a disposizione le chiavi pubbliche utilizzate per firmare i file Release, nel caso di Debian queste sono distribuite con il pacchetto *debian-archive-keyring*. La procedura per verificare l'autenticità di un pacchetto è la seguente:

1. **Acquisizione dei file Release, Release.gpg e gli indici Packages**
   con il comando `apt-get update` il client scarica i file Release e Release.gpg e verifica la firma digitale, dopodiché scarica i file Packages per le architetture da lui supportate dei componenti in uso, poi calcola l'hash dei file Packages ottenuti che devono corrispondere a quelli nel file Release

2. **Acquisizione pacchetto binario Debian**
   con il comando `apt-get instal nome-pacchetto` il client cerca nei file Packages, scaricati e autenticati, il corrispondente pacchetto da cui prende il campo Filename e lo concatena con l'URI della fonte per scaricare il pacchetto

3. **Autenticazione pacchetto acquisito**
   una volta scaricato il pacchetto, viene calcolato il suo hash e confrontato con quello presente nel file Packages, in caso positivo il pacchetto è integro e autentico.

In pratica l'autenticazione dei pacchetti è realizzata con una catena di hash dove l'integrità di ogni pacchetto è verificato con i file Packages, mentre l'integrità dei file Packages a loro volta è verificata con gli hash nel file Release, il quale è firmato digitalmente e certifica l'autenticità e l'integrità dell'intera catena. È interessante notare che nei file Release e Packages vengono specificati i valori hash di MD5, SHA1 e SHA256, questo perché al momento della nascita di apt-secure, MD5 si riteneva una funzione hash sicura, ma negli anni a seguire venne aggiunta SHA1 e attualmente quella in uso è la SHA256, anche se le altre sono tuttora presenti probabilmente per motivi di retrocompatibilità con i client APT più vecchi.

L'attuale schema di autenticazione si affida alla sicurezza delle chiavi utilizzate per firmare l'archivio. Tuttavia non è sufficiente a garantire che un pacchetto non sia malevolo, in quanto i manutentori Debian hanno a loro volta delle chiavi che permettono di caricare pacchetti nell'archivio. Il software che gestisce l'archivio, al momento della ricezione di un pacchetto, controlla l'autenticità dei pacchetti caricati verificando la firma digitale del corrispondente pacchetto sorgente. Quindi il manutentore dell'archivio a sua volta ripone la propria fiducia nei manutentori dei pacchetti Debian, un numero piuttosto elevato, che mantengono al sicuro la propria chiave di firma dei pacchetti. In caso di compromissione della chiave di un manutentore, pacchetti contenenti

codice malevolo potrebbero finire nell'archivio e questi saranno semplicemente considerati affidabili in quanto firmati da una chiave considerata valida. Le chiavi sono ritenute valide se sono presenti nel portachiavi gpg distribuito con il pacchetto *debian-keyring*, contenente le chiavi dei manutentori e sviluppatori ufficiali Debian. Ovviamente nel caso di introduzione di codice malevolo in un pacchetto sorgente, data la quantità di persone che partecipano al progetto, sicuramente verrebbe segnalato prima o poi.

Un altro potenziale problema di sicurezza è dato dal fatto che i manutentori caricano nell'archivio sia il pacchetto sorgente che il pacchetto binario compilato da loro. Attualmente non c'è modo di stabilire se un pacchetto binario sia stato compilato da un certo pacchetto sorgente e nulla vieta al manutentore di caricare un pacchetto .deb arbitrario. Debian cerca di risolvere questo problema muovendo passi in due direzioni: da un lato accettare nell'archivio solamente i pacchetti sorgente [37] e delegare *buildd* a compilare tutti corrispondenti binari, dall'altro rendere la compilazione dei pacchetti binari riproducibile [38] in modo che un pacchetto sorgente produca sempre lo stesso identico pacchetto binario, per stabilire una corrispondenza univoca tra i sorgenti e i binari. In realtà, come vedremo, la compilazione riproducibile è un grande sforzo messo in atto da diversi progetti, Tails incluso, che vogliono stabilire un collegamento esplicito tra il codice sorgente e il risultato della sua compilazione che normalmente viene distribuito.

4.3.2 Pacchetti binari firmati

Lo schema implementato da secure APT permette di verificare che i pacchetti scaricati provengano da un certo repository e associarli a una certa distribuzione. Di fatto non sono i pacchetti binari ad essere firmati, ma i metadati generati dal repository. Quindi un pacchetto binario ottenuto con un metodo diverso da APT, non rende possibile verificarne l'autenticità visto che sono assenti i metadati del repository.

Per rendere possibile una verifica indipendente ed a livello di singolo pacchetto binario, sono stati implementati due metodi simili ma incompatibili tra loro. Entrambe le implementazioni sfruttano il fatto che dpkg ignora i membri dell'archivio ar che iniziano con il carattere _(underscore), fattore che permette di aggiungere membri all'archivio senza compromettere la compatibilità con dpkg. I due tool, `dpkg-sig` [39] e `debsigs` [40], permettono entrambi di associare un ruolo ad una firma e di apporre firme multiple sui pacchetti. I ruoli dovrebbero essere associati ad entità che rappresentano l'origine del pacchetto, il suo manutentore e l'archivio che lo distribuisce, rendendo possibile la tracciabilità del percorso del pacchetto fino all'utente finale il quale potrà verificare che durante il passaggio da un'entità all'altra, questo non sia stato alterato.

Ciò che differenzia i due tool innanzitutto è che dpkg-sig può sia firmare che verificare i pacchetti, mentre debsigs può solo firmare ed è necessario un altro tool, `debsig-verify` [41], per verificare le firme. Un'altra importante differenza è che per verificare le firme con debsigs-verify è necessario definire delle policy in XML e dei corrispondenti portachiavi gpg. Le policy definiscono le regole di verifica da applicare in base all'ID della chiave usata per firmare il pacchetto, permettendo di associare i ruoli alle chiavi. Un punto a favore di debsigs-verify è la sua integrazione in dpkg, che supporta la verifica dei .deb prima della loro installazione, anche se di default è disabilitato nel file di configurazione */etc/dpkg/dpkg.cfg* dalla linea "*no-debsig*" perché di fatto i pacchetti provenienti dai repository ufficiali di Debian non sono firmati, ritenendo sufficiente il meccanismo di secure APT.

4.3.3 Compilazioni riproducibili

L'obbiettivo della compilazione *riproducibile* o deterministica, è quello di dare la garanzia che ad un certo sorgente corrisponda esattamente un certo binario [63]. L'unico modo di dare questa garanzia è permettere a chiunque di compilare i sorgenti e ottenere sempre lo stesso identico risultato bit per bit. La motivazione dietro a questo obbietto è stata in parte già anticipata nella sezione 4.3.1, ma oltre alla possibilità di uno sviluppatore malevolo che pubblica un trojan (sezione 4.4.1), gli sviluppatori stessi possono essere bersaglio di attacchi mirati. Il caso di XCodeGhost [64] mise in evidenza il problema: delle versioni trojan dell'ambiente di sviluppo Apple furono distribuite in Cina e gli sviluppatori di applicazioni iOS colpiti, includevano inconsapevolmente un malware nelle loro app, compilandole con l'IDE modificato. La novità di questo attacco è che l'integrità del software viene compromessa indirettamente, attraverso la compromissione dei tool usati nella costruzione del software. Il processo di compilazione trasforma l'unica cosa umanamente comprensibile e verificabile (il codice sorgente) in un qualcosa che lo sviluppatore considera affidabile a prescindere e che finirà per essere firmato e rilasciato agli utenti che a loro volta lo considereranno affidabile. Se consideriamo il caso di Debian, i bersagli di tale attacco possono essere non solo i manutentori dei pacchetti, ma anche la rete dei builder automatici *buildd* [65], che di fatto compilano la maggior parte dei pacchetti binari presenti nell'archivio.

La compilazione deterministica permetterebbe di rilevare tale tipo di attacchi, rimuovendo la necessità di fiducia nei singoli sistemi di build usati. Gli sviluppatori a questo punto possono usare un'infrastruttura di build distribuita, come quella di Debian ad esempio, per compilare i propri software e verificare che tutti i nodi producano lo stesso risultato. A questo punto l'attaccante dovrebbe compromettere tutti i nodi di build contemporaneamente,

perché la compromissione di uno solo verrebbe identificata dal fallimento nel riprodurre il risultato degli altri nodi. Un altro vantaggio del determinismo è che si riduce la necessità di fiducia anche nelle chiavi usate dagli sviluppatori per firmare le release, perché se il binario pubblicato è riproducibile, la fiducia è riposta nello specifico valore hash del binario, che può essere certificato da entità anche esterne allo sviluppatore.

La riproducibilità del software è sicuramente una caratteristica desiderabile considerati gli innumerevoli vantaggi dal punto di vista della sicurezza, ma ci sono diversi fattori che influenzano la compilazione, fino al punto che è possibile ottenere diversi risultati ogni volta perfino sullo stesso sistema. In [63] sono documentate le numerose fonti di non determinismo che influenzano il prodotto finale, tra cui informazioni locali al computer come ad esempio il nome utente dell'utente che compila, il path da dove si compila, così come informazioni varianti nel tempo come i timestamp del momento di compilazione o file esterni ottenuti dalla rete che potrebbero variare. Ma possono esserci cause ancora più subdole come l'ordine di input dei file al compilatore, che deve essere sempre lo stesso e perciò è necessario un esplicito ordinamento prima, usando esplicitamente la stessa localizzazione perché influenza il modo in cui i file vengono ordinati.

La strada migliore per riuscire a rendere il processo di compilazione deterministico, è stabilire uno specifico ambiente di compilazione e renderne possibile la riproduzione esatta, a partire dal sistema operativo fino agli strumenti usati per compilare il software. Da questo punto di vista vengono in aiuto le macchine virtuali che rendono semplice la distribuzione e ricostruzione di un ambiente identico su computer differenti. Tra i progetti che hanno raggiunto la riproducibilità ci sono Bitcoin Core [66] e Tor Browser [61], ed entrambi usano Gitian [67] un tool che semplifica il setup dell'ambiente di compilazione con le macchine virtuali, attraverso l'uso di un file che descrive l'ambiente da riprodurre.

Debian sta cercando di realizzare la riproducibilità seguendo i principi appena descritti. In particolare il team coinvolto in [38], sta progettando un design [68] che possa integrarsi nell'attuale infrastruttura dell'archivio di Debian. L'idea è quella di includere un nuovo file di controllo *.buildinfo* in cui viene registrato l'ambiente di compilazione usato per compilare un certo pacchetto sorgente e ottenere un certo pacchetto binario o insieme di pacchetti binari se il sorgente ne produce più di uno. Il nuovo file di controllo registra informazioni specifiche al computer su cui viene fatta la compilazione, in particolare l'architettura, il path in cui il pacchetto è stato compilato, l'elenco dei pacchetti e versioni, che formano l'ambiente di compilazione insieme alle loro dipendenze. Quindi il nuovo file di controllo vuole mettere in relazione l'ambiente, il pacchetto sorgente e i suoi prodotti binari, il tutto accompa-

gnato da una firma crittografica dell'entità che ha eseguito la compilazione. L'integrazione del file di controllo dovrebbe avvenire in questo modo [68]:

1. i manutentori caricano normalmente nell'archivio i pacchetti sorgente e i pacchetti binari compilati da loro e in più caricano il file .buildinfo, generato durante la compilazione, in un archivio apposito contenente solo i file .buildinfo

2. i rebuilder, che possono essere gli autobuilder *buildd*, scaricano dall'archivio i pacchetti sorgente e i corrispondenti file .buildinfo dall'archivio dedicato, riproducono l'ambiente di compilazione definito e ricompilano i pacchetti sorgente controllando di ottenere lo stesso risultato certificato dal file .buildinfo, in caso positivo creano anche loro un .buildinfo firmato e lo inviano all'archivio dedicato

3. gli utenti finali scaricano solamente i pacchetti binari e i corrispondenti file .buildinfo firmati disponibili negli archivi dedicati, verificano che il loro pacchetto binario corrisponda esattamente a quello compilato da tutte le altre entità, senza il bisogno di ricompilarlo loro stessi.

Le statistiche [69] sullo stato di riproducibilità dei pacchetti Debian sono piuttosto promettenti e si spera che la maggior parte dei pacchetti della prossima distribuzione stabile, Debian 9 stretch, siano riproducibili. Tuttavia, come si può vedere nella sezione "Categorized issues" del sito [69], i problemi sono ancora molti ed è necessario uno sforzo collettivo da parte di tutti i manutentori in modo che siano in grado di riconoscere e rimuovere le fonti di non determinismo dai loro pacchetti.

4.4 Attacchi all'integrità dei software

In questa sezione verranno discusse le principali minacce alle componenti dei software installati, analizzando come queste tipicamente agiscono e quali sono i danni derivanti.

4.4.1 Trojan

Uno dei più classici attacchi è quello di indurre l'utente a credere che il programma che sta scaricando o utilizzando è un programma ben conosciuto oppure un qualche "magico" software dalle incredibili funzionalità, quando in realtà al suo interno contiene un malware che infetterà il sistema non appena l'utente lo installa o lo manda in esecuzione. Sono moltissimi gli esempi di trojan trovabili in rete, così come sono moltissimi gli scopi per cui vengono

utilizzati [42]. In particolare in ambiente Debian, possono capitare casi di progetti che vengono sviluppati, impacchettati e distribuiti dagli sviluppatori stessi attraverso un repository APT gestito da loro, come ad esempio il browser Google Chrome [43]. Una volta che l'utente aggiunge una fonte non ufficiale e la corrispettiva chiave di firma del repository con il comando `apt-key add` *`chiave.gpg`*, considera il repository affidabile. Lo scenario più comune è che un repository fornisca direttamente il trojan con il pacchetto pubblicizzato, ma sono possibili scenari più subdoli. Un altro attacco potrebbe essere che il repository malevolo fornisca veramente il software pubblicizzato e che questo non sia malevolo, ma oltre a questo potrebbe fornire anche un finto aggiornamento di un altro pacchetto qualsiasi, semplicemente fornendolo con lo stesso nome e un numero di versione superiore, per sostituire tale pacchetto nel momento in cui l'utente eseguirà `apt-get upgrade` per aggiornare i propri pacchetti. È importante inoltre considerare che i pacchetti dei repository non ufficiali non sono tenuti a seguire la Debian Policy che pone delle limitazioni a ciò che un pacchetto può fare. Considerando l'esempio di Google Chrome, questo inizialmente viene fornito come pacchetto .deb a se stante, e l'utente lo installa direttamente usando dpkg, ma esaminando lo script di post-installazione, si può vedere che questo al suo interno contiene la fonte APT e la chiave di firma dell'archivio di Google, lo script del manutentore durante la sua esecuzione aggiunge la fonte e la chiave alla configurazione di APT, per abilitare gli aggiornamenti dal repository. Chiaramente Google procede in questo modo solo per non chiedere agli utenti di farlo a mano, ma a prescindere gli script del manutentore potrebbero fare qualunque cosa visto che sono eseguiti con i privilegi di amministratore. L'utente, quindi, deve stare attento a quali fonti APT aggiunge e quali pacchetti .deb installa, verificandone l'attendibilità, visto che è sufficiente un solo repository o pacchetto binario malevolo per compromettere il sistema.

I gestori dei pacchetti software come APT e dpkg sono strumenti relativamente recenti che sono stati sviluppati per rendere semplice la gestione dei software, tuttavia ancora può capitare di trovare applicazioni che vengono distribuite precompilate sotto forma di archivi tar compressi o altre forme. La pratica comune utilizzata dagli sviluppatori è di distribuire insieme agli archivi dei file dell'applicazione, anche i corrispondenti hash, ad esempio pubblicandoli sulla pagine di download in modo da permettere a chi li scarica di verificare l'integrità dei file. Il problema è che solo gli hash non sono sufficienti perché in caso di compromissione del sito di distribuzione, l'attaccante potrebbe sostituire non solo l'applicazione con un trojan ma anche i corrispondenti hash che permettono di verificare l'integrità ma non l'autenticità. Quindi per garantire l'autenticità e l'integrità ciò che normalmente viene fatto è analogo al metodo usato da secure APT: si distribuiscono i binari accompagnati dai cor-

rispondenti hash e una firma digitale degli hash. Nonostante questa pratica sia standard di fatto, gli utenti non sempre verificano ciò che scaricano, probabilmente perché non hanno le nozioni di funzione hash e firma digitale e non sanno come utilizzare i dati forniti, ma anche in questo caso gli sviluppatori pubblicano anche istruzioni o riferimenti a guide e tutorial sulla procedura di verifica.

Ken Thompson, uno dei creatori di Unix, espose per la prima volta l'idea dei trojan già nel 1983 durante il suo discorso [44] per la vittoria del Turing award: egli ipotizzò che modificando il compilatore C per riconoscere che l'input da compilare è il comando login [45], sarebbe stato possibile generare un binario alterato in modo tale che il comando oltre ad accettare la password dell'utente legittimo, accettasse anche una password "backdoor" che permettesse di accedere al sistema anche all'attaccante, ed inoltre essendo scritto in C il compilatore stesso, questo avrebbe potuto riconoscere la compilazione di una nuova versione del compilatore stesso, e quindi manipolare anche questo per mantenere l'accesso al sistema. L'idea di Ken Thompson fece da precursore ad una tipologia di malware nata all'inizio degli anni 90: i rootkit.

4.4.1.1 Attacco a Linux Mint

Linux Mint [46], una conosciuta distribuzione derivata da Ubuntu [47] che a sua volta è basata su Debian, ha subìto un attacco al sito ufficiale usato per la distribuzione delle ISO del sistema operativo e al relativo forum dedicato agli utenti. Uno degli amministratori del progetto ha informato i propri utenti che il link per il download dell'ISO, relativo alla versione 17.3 con ambiente desktop Cinnamon, è stato modificato per puntare ad una versione trojan dell'ISO contenente una backdoor. Non appena gli amministratori si accorsero dell'attacco, rimossero il link alterato e dissero che gli utenti colpiti erano limitati a coloro che nella giornata del 20 febbraio 2016 scaricarono quella specifica versione, ma il giorno dopo un utente segnalò nuovamente che il link di download sembrava sospetto: il sito è stato violato nuovamente e quindi la vulnerabilità usata per farlo era ancora disponibile. A seguito del secondo attacco l'amministratore fu costretto a mettere offline il sito per investigare sulla natura dell'attacco.

Prima dell'attacco, il progetto pubblicava solamente gli hash MD5 delle ISO pubblicate e considerato che l'attaccante ha modificato il link per il download, poteva anche modificare gli MD5 stessi, senza contare il fatto che MD5 è obsoleto. A seguito di molte critiche sul non utilizzo di una funzione hash sicura e l'assenza di release firmate, il progetto si adattò agli standard e attualmente pubblica l'elenco degli hash SHA256 delle ISO firmato con gpg.

Questo specifico caso di attacco mise in evidenza la negligenza dei gestori del progetto che non usavano le pratiche comuni adottate da praticamente tutte le maggiori distribuzioni Linux. Dall'altra parte l'attaccante non modificò gli MD5 pubblicati sul sito, dimostrando anche la non curanza degli utenti che a prescindere dal metodo di verifica offerto, non lo utilizzano.

4.4.2 Rootkit a livello utente

Sono una categoria di malware che modificano il sistema per nascondere la propria presenza e le tracce dell'intrusione. Esistono diversi tipi di rootkit [53] e sono categorizzati in base al contesto di esecuzione, tra cui a livello di firmware, kernel o livello utente. I più interessanti per il nostro caso di studio sono i rootkit a livello utente. Una volta ottenuti i privilegi di amministratore, cercano di nascondere i file che li compongono e le eventuali connessioni di rete che usano [48]. Normalmente questi rootkit sostituiscono comandi tipicamente presenti sulla maggioranza dei sistemi Unix-like con versioni trojan. I programmi più bersagliati sono quelli usati per fare login, come l'omonimo comando, o anche *sshd* [49], il demone in ascolto di connessioni remote attraverso ssh, tutto questo per rendere semplice l'accesso da parte dell'attaccante. Per nascondere le tracce del proprio operato, i rootkit sostituiscono anche comandi usati per ottenere informazioni sui processi in esecuzione come *ps* [50], *netstat* [51] per le connessioni di rete ed *ls* [52] per elencare i file, tutto ciò per nascondere i propri processi, connessioni e file.

Un altro approccio usato dai rootkit a livello utente è sostituire le librerie condivise installate sul sistema. È una pratica comune delle applicazioni ma anche dei sistemi operativi stessi, in ambiente Unix-like, implementare le funzionalità chiave con delle librerie condivise per renderle disponibili anche ad altre applicazioni, per evitare la duplicazione del codice. In questo caso le librerie trojan installate dal rootkit forniranno risultati errati alle applicazioni che le usano, nascondendo allo stesso modo le tracce della propria presenza, ma senza alterare direttamente i programmi elencati in precedenza.

4.5 Tecniche e strumenti per la verifica dell'integrità post-installazione dei software

Finora abbiamo discusso del percorso che fa un'applicazione a partire dal suo sviluppatore fino all'utente finale. È stata evidenziata l'importanza di un'infrastruttura sicura e delle catene di fiducia che si formano per l'autenticazione dei pacchetti software che vengono distribuiti in Debian. È sicuramente un requisito fondamentale per mantenere il proprio sistema sicuro, installare

software proveniente da fonti fidate, tuttavia lo stesso software fidato può avere dei difetti che possono essere sfruttati per alterare quelli già installati. I pacchetti software installati su un sistema non sono altro che una collezione di file sul file system e come tali sono vulnerabili ad alterazione o completa sostituzione. Questa sezione di conseguenza tratterà di quali siano i metodi e i tool che possono essere usati per verificare l'integrità dei pacchetti software installati.

4.5.1 Dpkg

Un primo modo di verificare l'integrità dei pacchetti installati è integrato nel sistema di gestione dei pacchetti. Tra i file di controllo opzionali forniti dai pacchetti binari, può esserci un file *md5sums* contente gli hash MD5 dei file che verranno installati dal pacchetto. Il gestore di pacchetti può eseguire la verifica dei file installati dai pacchetti con il comando `dpkg --verify` utilizzando i corrispondenti file *<nome-pacchetto>.md5sums*, se presenti. Il comando, se invocato senza argomenti, verifica tutti i pacchetti installati, altrimenti è possibile specificare quali pacchetti verificare.

Questo metodo di verifica è principalmente pensato per rilevare le alterazioni di natura accidentale, come ad esempio malfunzionamenti dell'hard disk. Se per ipotesi un'attaccante può modificare i file dei pacchetti installati, per cui sono necessari i permessi di amministratore, oltre al fatto che potrebbe forgiare un trojan che collide con l'MD5 del corrispondente file sostituito, allo stesso modo può direttamente sostituire il corrispondente valore hash nel file md5sums con il valore del suo file malevolo. L'attaccante potrebbe anche sostituire il binario del programma `md5sum` (fornito dal pacchetto *coreutils* [78]), che viene usato per il calcolo dei valori MD5 dei file. La versione trojan di `md5sum`, sarà modificata in modo da mascherare gli altri trojan presenti, restituendo i valori hash dei file originali invece che quelli dei trojan. Quindi questo metodo non è molto utile visto che non c'è modo di verificare l'integrità e l'autenticità dei file md5sums.

4.5.2 Debsums

È un tool [54] che cerca di colmare le lacune di `dpkg --verify`. Esegue la stessa procedura del gestore dei pacchetti per verificare i file installati, ma in più offre la possibilità di generare i file md5sums mancanti. Inoltre per diminuire il numero di falsi positivi, normalmente esclude i file di configurazione dei pacchetti dal controllo, visto è normale che questi cambino dal momento di installazione.

Il tool ovviamente soffre degli stessi problemi del gestore dei pacchetti.

4.5.3 Monitoraggio dell'integrità dei file (FIM)

È un metodo generico per monitorare file arbitrari nel filesystem, basandosi sempre sul confronto di valori hash ed eventualmente altre proprietà dei file come i permessi, le date di creazione, accesso e modifica. Sono disponibili diverse soluzioni sia commerciali che open source per il monitoraggio dei file. In generale i FIM inizialmente vengono configurati selezionando quali parti del file system monitorare, dopodiché il software esegue una scansione dei file da monitorare calcolandone gli hash e salvandoli insieme ad eventuali altre proprietà del file in un database. Il database generato rappresenta lo stato attuale dei file nel sistema, che si assume per integro. In un secondo momento il software FIM utilizzerà questo stesso database per verificare l'integrità dei file ricalcolando i valori hash e confrontandoli con quelli precedentemente salvati, rilevando così le modifiche avvenute dal momento della creazione del database.

Il più famoso FIM open source, incluso anche in Debian, è Open Source Tripwire [55]. Il software si configura con la definizione di una policy che descrive le parti del file system da monitorare, e la caratteristica più interessante è l'utilizzo della crittografia per tenere al sicuro la policy e il database con gli hash, proteggendoli da modifiche non autorizzate.

I FIM sono sicuramente utilizzabili per verificare l'integrità delle applicazioni, ma il problema deriva dalla granularità dei loro controlli a livello di file. Un FIM non è in grado di raggruppare insiemi di file e considerarli come applicazione, per poter segnalare all'utente quale pacchetto software è stato alterato, ma saprebbe indicare solamente che qualche file è cambiato. Un altro inconveniente è la necessità di definire la policy, ed un utente dovrebbe sapere quali parti del file system monitorare, e quali evitare perché contengono file che cambiano durante il corso di esecuzione del sistema operativo o delle applicazioni stesse. È importante anche considerare che gli aggiornamenti dei pacchetti fatti tramite APT, nonostante siano cambiamenti attesi, verrebbero comunque visti come alterazioni.

4.5.4 File eseguibili digitalmente firmati

Un altro approccio possibile per verificare i software installati è utilizzare file eseguibili digitalmente firmati. Le proposte di questo tipo, [56] e [57] sfruttano la stessa idea di base. In ambiente Linux, ma anche in molti altri sistemi Unix-like, il formato utilizzato per gli eseguibili binari è l'Executable and Linkable Format (ELF). Il formato ELF suddivide i componenti di un programma, il codice, i dati statici e variabili usati nel programma, e altre informazioni necessarie al caricamento ed esecuzione del programma, in parti chiamate sezioni. Il vantaggio del formato ELF è che è possibile includere nei

file dei dati arbitrari, senza renderli incompatibili per l'esecuzione e senza il bisogno di ricompilare il programma, ma semplicemente definendo delle nuove sezioni. Le proposte inizialmente elencate infatti sfruttano questo metodo per aggiungere i propri dati utilizzati per la verifica del file eseguibile in sezioni da loro definite.

Le soluzioni descritte in [56] e [57] propongono sostanzialmente di firmare digitalmente le sezioni utilizzate per l'esecuzione e il caricamento del programma in memoria, e salvare tale firma in una sezione apposita. Con l'uso della crittografia a chiave pubblica, tutti i file ELF vengono firmati, e prima dell'esecuzione di questi, è il kernel a fare la verifica della firma del file. Nello specifico, un kernel modificato, al momento della richiesta di eseguire un programma con la chiamata di sistema `execve`, prima di mandare effettivamente in esecuzione il programma, verifica la firma presente nella sezione aggiunta, e in caso di assenza o invalidità di questa, l'esecuzione del file viene rifiutata.

In Debian è disponibile un tool [59] che permette di aggiungere ai file ELF con la tecnica appena descritta, l'hash SHA1 del file o una firma digitale fatta con gpg. Il solo hash permette di verificare l'integrità, mentre la firma garantisce anche l'autenticità. Se un attaccante sa che tutti gli eseguibili sono stati modificati da `bsign` per includere i loro valori hash, l'attaccante stesso potrà usare il tool sul proprio trojan e per aggiungergli il suo valore hash, così anche questo potrà essere verificato e considerato integro, rimanendo comunque un trojan. Utilizzando una firma digitale invece, oltre all'integrità del file, viene garantita anche la sua autenticità se l'utente che ha eseguito la firma è l'unico che controlla la chiave privata. Anche se il tool è ancora disponibile nell'attuale versione stabile di Debian, non sarà incluso nella prossima, visto che non è più in sviluppo attivo ed è compatibile solamente con il formato ELF a 32 bit.

Il progetto DigSig [58] è l'unica implementazione, che segue le idee di [56] e [57], ancora pubblicamente disponibile ma di fatto abbandonata e non più aggiornata. DigSig ha optato per l'utilizzo di bsign per la firma dei binari, quindi inizialmente l'utente usa il tool per firmare i propri file eseguibili nel sistema. Una volta che le firme sono state apposte, si utilizza il modulo kernel di DigSig per verificare le firme e prevenire l'esecuzione degli eseguibili non firmati.

Il metodo di verifica appena descritto è sicuramente interessante e utile, tuttavia non ha ricevuto il supporto sperato dalla comunità e di fatto le implementazioni sono state abbandonate. La più grande limitazione di questo approccio è che sfrutta una caratteristica dei file ELF, e quindi non è applicabile a tutti gli altri tipi di eseguibili come ad esempio gli script. Attualmente in Debian sono numerosi i pacchetti che forniscono software implementato esclusivamente con linguaggi di scripting perché li rende altamente portabili e indipendenti dall'architettura del processore. In [56] si propone di aggiungere

ai file di testo degli script, le firme come commenti dei rispettivi linguaggi di scripting, ma data la quantità e varietà dei linguaggi non sarebbe molto pratico. Inoltre il problema sorge nel momento in cui si vuole aggiornare un pacchetto perché è necessario ripetere il processo di firma per il nuovo binario installato.

4.5.5 Confronto dei tool analizzati

Attualmente la maggiore attenzione viene posta sulla distribuzione e autenticazione sicura dei pacchetti software nel loro complesso (l'intero pacchetto .deb) e non sui singoli componenti in esso contenuti. Debian offre un modo basilare di verificare i file installati dai pacchetti con `dpkg` e `debsums`, tuttavia abbiamo visto che non sono sufficientemente sicuri. I FIM invece richiedono un'attenta configurazione e comunque non hanno modo di distinguere tra i cambiamenti attesi e quelli malevoli. Gli eseguibili firmati, la più promettente delle soluzioni, è in grado di risolvere il problema solo in parte dato che sfrutta una caratteristica specifica al formato ELF.

Capitolo 5

Tails

Nel primo capitolo è stato definito il problema da affrontare: verificare l'integrità dei software installati nella distribuzione Tails. L'analisi della gestione dei pacchetti software in Debian ci sarà utile perché Tails è una versione live di Debian con alcune modifiche e personalizzazioni mirate a proteggere l'utente medio che non ha conoscenze tecniche approfondite. In questo capitolo verranno descritte le caratteristiche principali di Tails, documentate in [60], che influenzeranno lo sviluppo del sistema utilizzato per verificarne i software installati.

5.1 Un sistema operativo live

Come anticipato, Tails, acronimo di The Amnestic Incognito Live System [2], è un sistema operativo live. Tails si prepone come obbiettivo di proteggere l'anonimità e la privacy del proprio utente con un sistema operativo che è pensato per poter essere usato indipendentemente dal sistema operativo installato sull'hard disk di un particolare computer e come si evince dal nome, non lasciare tracce della propria esecuzione.

Un tradizionale sistema operativo è installato sull'hard disk e durante il suo utilizzo anche i file e i dati prodotti durante una sessione, vengono salvati sull'hard disk. I sistemi operativi normalmente salvano molti dati utili a tracciare le attività svolte nelle sessioni di utilizzo, e nel caso in cui l'utente voglia proteggere ad esempio dati particolarmente sensibili, la sola cancellazione dei file dall'hard disk non è sufficiente perché i dati rimangono nell'hard disk fino al momento in cui le aree del disco in cui erano salvate non vengono sovrascritte. Inoltre nel caso di sistemi operativi commerciali come Windows, non è possibile sapere quali dati vengano raccolti, per quanto tempo e se è possibile eliminarli. Tra gli altri pericoli è l'utilizzo di computer pubblici ad esempio in un internet café o una biblioteca, che sono esposti ad alterazioni da parte di

chi lo utilizza. Quindi in generale se si vuole essere certi di non lasciare tracce della propria attività, l'utente non può fare affidamento sul sistema operativo installato perché non ha modo di sapere quali dati vengano salvati e come eliminarli.

Attualmente molte distribuzioni Linux, Debian incluso, distribuiscono le immagini del sistema operativo sotto forma di ISO, un tipo di file system pensato per essere scritto sui CD. La tipica procedura per installare la distribuzione consiste nel copiare il file ISO su un CD, ma è possibile utilizzare anche altri tipi di supporti come chiavette USB, e al posto di avviare il computer dall'hard disk e far partire il sistema operativo già installato, lo si avvia dal supporto contenente la distribuzione. L'immagine del sistema operativo nel supporto viene caricata in RAM e da qui è possibile provare la distribuzione prima di installarla, senza alterare il sistema operativo già presente ma utilizzare la distribuzione in modalità live appunto. Solo nel momento in cui si decide di installare la distribuzione in modo permanente, entra in gioco la persistenza, perché a questo punto l'installazione sull'hard disk consiste nel copiare i contenuti dal supporto rimovibile all'hard disk. Se l'installazione non viene eseguita ed il computer viene spento, tutti i dati della distribuzione vengono persi perché contenuti solamente in RAM, di fatto lasciando inalterato l'hard disk.

Tails è una distribuzione pensata per essere usata solamente in modalità live. Gli utenti quindi creano un proprio supporto rimovibile a partire da un'immagine di Tails in formato ISO, ed avviano Tails da tale supporto. Comunque, in modalità live è possibile utilizzare l'hard disk del computer su cui Tails è avviato, ma per farlo l'utente deve esplicitamente configurare una password di amministratore al momento dell'avvio di Tails e comunque verrà avvertito che tale azione lascerà tracce visibili sul computer.

Tails quindi essendo un sistema operativo completamente live, non ha memoria delle sessioni e tutti i file creati, modificati o scaricati dalla rete, vengono persi se non esplicitamente salvati su un supporto persistente. Nel caso in cui Tails sia stato installato su una chiavetta USB, c'è la possibilità di utilizzare lo spazio rimanente a disposizione per creare una partizione persistente criptata con una password. Tails all'inizio di ogni sessione chiede all'utente alcuni parametri di configurazione, tra cui l'abilitazione o meno dell'utente root impostandone la password, e nel caso in cui sia presente una partizione criptata sul supporto da cui è avviato, si chiede all'utente se vuole utilizzarla o meno chiedendo la password per decriptarla.

La documentazione descrive che anche se i contenuti della RAM vengono persi quando il computer viene spento, esiste un attacco [76] che permette di recuperare i contenuti della RAM al momento dello spegnimento se si agisce in tempi brevi dallo spegnimento. Per proteggere ulteriormente l'utente, al

termine di ogni sessione, Tails sovrascrive esplicitamente i contenuti della RAM per renderne impossibile il recupero.

5.2 Accesso a Internet attraverso Tor

L'altra caratteristica fondamentale di Tails è che dirige tutto il traffico Internet attraverso Tor [61] per proteggere l'anonimità online dell'utente. Più nello specifico, Tails ha un firewall configurato per bloccare tutte le connessioni eccetto quelle alla rete Tor. Di conseguenza solamente le applicazioni che utilizzano TCP possono usare la rete, in quanto è l'unico protocollo di trasporto supportato da Tor. Considerando che il protocollo UDP è bloccato di default perché non supportato da Tor, le richieste DNS che usano appunto UDP come trasporto, vengono identificate dalla porta di destinazione standard 53, e redirezionate al servizio di risoluzione DNS di Tor che incapsulerà la richiesta in TCP e si prenderà carico della risoluzione, mentre tutti gli altri datagrammi UDP sono bloccati.

Dal punto di vista delle applicazioni, Tor non è altro che un server proxy a livello TCP. Un'applicazione che vuole utilizzare Tor per il suo traffico Internet, deve essere in grado di usare almeno una tra le versioni del protocollo SOCKS. Attualmente esistono le versioni 4, 4a e 5, di cui solo l'ultima è specificata con un RFC [62]. Il funzionamento generale del protocollo è semplice: il client chiede al server di connetterlo a un certo host su una certa porta, il server stabilisce la connessione con tale host sulla porta richiesta e lo notifica al client, a questo punto il client invia i dati al proxy e questo si occuperà di inviarli all'host e porta inizialmente specificati. La differenza principale tra le versioni, è che la 4 permette di specificare l'host solamente tramite il suo indirizzo IP, mentre dalla versione 4a in poi è possibile specificare il nome di dominio dell'host e di fatto delegare il proxy alla risoluzione DNS. La versione 5 ha introdotto un'iniziale handshake per negoziare un metodo di autenticazione, e al contrario delle versioni precedenti supporta anche UDP, ma nel caso di Tor l'autenticazione non è richiesta e il protocollo UDP viene comunque rifiutato.

Il grande vantaggio del protocollo SOCKS è che le applicazioni possono usare la rete Tor in modo trasparente. Dopo la fase iniziale di connessione al proxy, e una volta che questo ha stabilito la connessione con l'host remoto, dal punto di vista dell'applicazione client è come se fosse direttamente connesso con l'host remoto. Quindi qualsiasi applicazione che usa TCP, a prescindere dal suo protocollo applicativo, può usare la rete Tor senza il bisogno di introdurre modifiche specifiche.

5.3 Software installato

Tails è basato sull'attuale release stabile di Debian ed ha a disposizione dpkg e APT per la gestione dei pacchetti software. Il progetto ha scelto uno specifico insieme limitato di pacchetti da includere [70]. Il criterio di scelta del software è stato guidato dalle attività tipiche di un utente medio, in modo da accomodare qualsiasi tipo di utente e cercando di offrire più software possibile con un'interfaccia grafica, facilitandone l'uso per gli utenti meno esperti. Gli utenti tipici vorranno sicuramente navigare su Internet, inviare email, chattare online, fare editing audio, video e di immagini, ma allo stesso modo vorranno anche la classica suite office, insomma deve essere possibile fare tutto o almeno la maggior parte di ciò che si può fare con un sistema operativo normale, ma usando solo software libero. Oltre ai software classici usati dagli utenti, sono disponibili strumenti per la crittografia come *gpg* e un gestore di password, molto utile se si configura una partizione persistente dove potranno essere salvate in maniera sicura. Inoltre per poter accontentare gli utenti che vogliono usare dei pacchetti che non sono inclusi, all'inizio della sessione è possibile configurare la password di amministratore per poter usare APT oppure la variante con interfaccia grafica Synaptic [71]. I pacchetti installati durante una sessione saranno chiaramente persi al momento dello spegnimento, dato che il supporto da cui Tails è caricato in RAM è montato in modalità read-only.

Tutti i pacchetti software che sono installati in Tails, provengono da 3 fonti: archivi ufficiali Debian per la maggior parte del software installato, archivi ufficiali Tor Project per i pacchetti relativi a Tor, ed infine gli archivi con software specifico a Tails sviluppato dal progetto stesso. Il progetto cerca di limitare il più possibile le differenze con Debian perché offre delle buone garanzie sulla qualità, quindi gli sviluppatori di Tails spesso contribuiscono direttamente a Debian.

5.4 ISO riproducibili

Nel tentativo di riuscire ad avere delle ISO riproducibili, Tails costruisce le ISO in una macchina virtuale. In particolare usa un tool chiamato Vagrant [72], che semplifica il setup della macchina virtuale e la ricostruzione di un ambiente di compilazione al fine di ottenere una compilazione deterministica. All'interno della macchina virtuale, viene utilizzato un'insieme di tool chiamato `live-build` [73], che permette la costruzione di immagini live basate su Debian, offrendo un'alta possibilità di configurazione del prodotto finale attraverso l'uso di una gerarchia di file di configurazione, applicati durante diversi stadi. Ad alto livello, i tool di live-build, costruiscono l'immagine creando inizialmente il file system di base contenente il minimo necessario per un siste-

ma operativo Debian per funzionare, in seguito il sistema viene personalizzato applicando i file di configurazione, i quali permettono di specificare l'elenco dei pacchetti da installare nell'immagine e da quali fonti APT ottenerli. Una volta terminata la personalizzazione dei contenuti del file system, questo viene trasformato in un tipo di file system compresso e read-only chiamato *SquashFS* [75], il quale diventerà appunto il file system che sarà caricato durante le sessioni live.

Come spiegato in 4.3.3, per ottenere una compilazione deterministica è necessario che ogni input alla compilazione venga preservato e rimanga identico. Considerando che i tool live-build, scaricano e installano con APT i pacchetti installati nelle ISO rilasciate da Tails, è necessario salvare l'insieme dei pacchetti binari usato durante la build per poterli riusare in futuro e ricostruire una certa ISO. Tails infatti, per ogni versione rilasciata, crea un corrispondente repository APT in cui salva tutti i pacchetti usati nella build dell'ISO della specifica versione [74]. In particolare i pacchetti provenienti da Debian e Tor Project vengono salvati nei cosiddetti repository *tagged* (`https://tagged.snapshots.deb.tails.boum.org/`), mentre i pacchetti sviluppati dal progetto stesso vengono salvati in `https://deb.tails.boum.org/`. Quindi usando questo insieme di repository, è possibile ottenere i pacchetti installati in ogni versione rilasciata.

Capitolo 6

Progettazione e implementazione del sistema di verifica

Questo capitolo descrive il sistema di verifica sviluppato per verificare i software installati in Tails. Dopo un'analisi del problema, verranno descritte le soluzioni costruite, motivando le scelte progettuali e implementative.

6.1 Analisi del problema

L'obbiettivo inizialmente preposto è verificare l'integrità dei pacchetti Debian disponibili in Tails. In particolare le parti di maggiore interesse sono i file eseguibili che i pacchetti installano, dato che questi sono i principali bersagli di attacchi. Tuttavia non è detto che un pacchetto contenga del software perché effettivamente i pacchetti Debian sono un metodo di distribuzione di file in generale. Alcuni esempi di pacchetti non contenti software sono i pacchetti di documentazione, pacchetti di soli file di configurazione per altri pacchetti, pacchetti di localizzazione, e pacchetti di dati in generale (icone, font, ecc.). È necessario quindi distinguere pacchetti contenenti software da quelli contenenti solo dati, perché solo quelli che installano file eseguibili sono di interesse per la verifica. Si vorrà anche restringere la verifica dei pacchetti solamente ai file eseguibili che i pacchetti installano, evitando di verificare file di configurazione, documentazione e licenza in quanto sono di secondaria importanza.

La verifica di integrità avverrà calcolando il valore hash dei file e confrontandolo con un valore di riferimento, visto che rimane sempre il metodo più sicuro per rilevare qualsiasi modifica ai file. I valori di riferimento dovranno provenire da una fonte esterna e calcolati con una funzione hash sicura come SHA256, perché gli unici valori di riferimento disponibili sono quelli dei file

locali *md5sums* che sono inaffidabili per i motivi descritti nella sezione 4.5.1. Sarà necessaria quindi una fonte di dati accessibile tramite Internet che fornisca dei valori hash di riferimento per i file da verificare. Il team di Tails cerca di rilasciare una nuova versione ogni 6 settimane, durante le quali potrebbero esserci release di emergenza a causa di vulnerabilità particolarmente gravi. Ogni nuova versione aggiorna dei pacchetti e a volte ne installa di nuovi, quindi la fonte di dati di riferimento dovrà essere aggiornata di conseguenza. In ogni momento la fonte dovrà avere i dati per la verifica di almeno le ultime 2 versioni rilasciate, dato che gli utenti di solito impiegano del tempo per aggiornare.

6.2 Panoramica dell'implementazione

Le parti che costituiscono il sistema di verifica sono il servizio di back-end che dovrà fornire i valori hash di riferimento e il client che li usa come riferimento per la verifica. Il servizio di back-end sarà costituto da un server web, quindi il client acquisirà i dati di verifica con il protocollo HTTP o HTTPS nel caso in cui il server lo supporta. È preferibile che il client utilizzi HTTPS perché oltre a fornire riservatezza delle comunicazioni, rende possibile autenticare il server e garantisce che i dati scambiati tra client e server non siano stati alterati durante il trasporto. Comunque supponendo l'utilizzo della versione non cifrata del protocollo, un attacco mirato a uno specifico utente Tails rimane estremamente difficile grazie al fatto che il traffico di rete transita attraverso la rete Tor, rendendo impossibile associare le richieste dei dati a degli specifici utenti.

Il primo tool sviluppato, a partire da un'insieme di pacchetti Debian, costruirà l'insieme di dati di riferimento che poi verranno resi disponibili attraverso il server web. Il secondo tool invece è il client che esegue la verifica utilizzando i dati forniti dal server web. Per lo sviluppo dei tool è stato scelto il linguaggio C++ perché rende possibile l'utilizzo delle interfacce al sistema operativo implementate in C, ma supporta anche il paradigma a oggetti ed offre una libreria standard molto ricca. Per rendere più agevole lo sviluppo, è stato utilizzato l'IDE Eclipse [79].

Inizialmente si descriverà la realizzazione del servizio di back-end per poi passare alla descrizione del client.

6.3 Costruzione dati di verifica

Come detto nella sezione 5.4, Tails rende disponibili i pacchetti Debian installati in ogni release attraverso dei repository APT, quindi i dati di ve-

rifica per ogni release saranno prodotti a partire dal corrispondente insieme dei pacchetti installati. La prima parte della costruzione del dataset consiste nell'acquisizione dei pacchetti di una determinata release con l'uso di `apt-get`. Una volta ottenuti i pacchetti della release, il primo tool, chiamato `pkghash`, costruirà il dataset per la release calcolando i valori hash di riferimento dei file installati dai pacchetti.

6.3.1 Esportazione elenco pacchetti installati in una release

Il primo passo per l'acquisizione dei pacchetti di una determinata release è ottenere l'elenco dei pacchetti installati. L'elenco dei pacchetti deve essere esportato in un formato tale da poter essere dato in input ad `apt-get` per eseguirne il download. Per esportare l'elenco dei pacchetti si è optato per l'uso di `dpkg-query` [80] che permette di interrogare il database di dpkg e specificare come deve essere formattato l'output. Il comando deve essere eseguito in un'istanza della release in questione, usando una macchina virtuale avviata con l'ISO della release. Una volta preparata ed avviata la macchina virtuale con Tails, per esportare l'elenco in un file, si esegue il seguente comando in un terminale:

```
$ dpkg-query -W -f='${Package}:${Architecture}=${Version}\n' > elenco
```

Il file conterrà un pacchetto per ogni riga ed ogni riga è del tipo *nome-pacchetto:architettura=versione*, questo formato è necessario per poter scaricare esattamente lo stesso pacchetto con APT. Dato che Tails attualmente supporta solo l'architettura *i386*, è necessario specificare esplicitamente l'architettura per i pacchetti, perché se si esegue il download su un computer *amd64* (come nel mio caso), APT per default scaricherà i pacchetti per amd64. Per essere certi inoltre di ottenere esattamente le stesse versioni dei pacchetti, si aggiunge esplicitamente quale versione si vuole del pacchetto. Quindi il file così ottenuto sarà utilizzato per scaricare i pacchetti in questo modo:

```
$ apt-get download $(cat elenco)
```

Prima di poter utilizzare l'elenco esportato, è necessario rimuovere alcuni pacchetti dall'elenco che non sono presenti nei repository APT. In particolare ci sono dei pacchetti "placeholder" che in realtà non installano nulla ma sono necessari solo per soddisfare delle dipendenze di altri pacchetti. I pacchetti placeholder sono identificabili dalla parola "fake" nella versione del pacchetto, quindi per rimuovere tali pacchetti dall'elenco è sufficiente usare `sed` [81] in questo modo:

```
$ sed -i '/^.*=.*fake.*/d' elenco
```

C'è anche il caso particolare del pacchetto *libdvdcss2* che non è un placeholder ma è comunque assente nei repository perché viene compilato dai sorgenti a tempo di costruzione dell'ISO, e quindi è necessario rimuovere questo pacchetto dall'elenco eseguendo il comando:

```
$ sed -i '/^libdvdcss2.*/d' elenco
```

Per comodità i comandi possono essere concatenati per produrre l'elenco direttamente filtrato prima di salvarlo nel file in questo modo:

```
$ dpkg-query -W -f='${Package}:${Architecture}=${Version}\n' | sed
   '/^.*=.*fake.*/d' | sed '/^libdvdcss2.*/d' > elenco
```

6.3.2 Configurazione APT e download dei pacchetti

Prima di utilizzare il comando per il download descritto nella sezione precedente, è necessario fare alcune configurazioni. Innanzitutto serve un file di fonti APT che elenca tutte le fonti necessarie per acquisire i pacchetti dai repository Tails. Per avere la certezza di scaricare i pacchetti solamente dai repository Tails ed ottenere esattamente quelli che sono installati nella release, è necessario che APT utilizzi solamente le fonti di Tails. Il primo step quindi è spostare temporaneamente i file .list delle fonti APT attualmente in uso, in particolare il file */etc/apt/sources.list* e gli eventuali file aggiuntivi nella cartella */etc/apt/sources.list.d/*, in una cartella di backup in modo da ripristinare la propria configurazione precedente al termine del download dei pacchetti.

L'ultima release di Tails al momento della scrittura è la 2.10, osservando i repository *tagged* di questa versione (`https://tagged.snapshots.deb.tails.boum.org/2.10/`), il file sources.list sarà il seguente il seguente:

```
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/debian/ jessie main contrib non-free
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/debian/ jessie-updates main contrib non-free
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/debian/ jessie-proposed-updates main contrib non-free
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/debian/ jessie-backports main contrib non-free
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/debian/ stretch non-free contrib main
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/debian/ sid main contrib non-free
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/debian-security/ jessie/updates main
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/torproject/ jessie main
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/torproject/ obfs4proxy main
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/torproject/ stretch main
deb  http://tagged.snapshots.deb.tails.boum.org/2.10/torproject/ sid main
deb  http://deb.tails.boum.org/ 2.10 main contrib non-free
```

Dopo aver configurato le fonti APT in modo da usare solo i repository Tails, è necessario importare la chiave di firma dei repository per poter autenticare i

pacchetti con secure APT. La chiave di firma è disponibile all'URL `https://deb.tails.boum.org/key.asc`, e per importare la chiave in APT è sufficiente il seguente comando da eseguire come utente root:

```
# curl https://deb.tails.boum.org/key.asc | apt-key add -
```

Ora è possibile eseguire il download dei pacchetti della release 2.10. Dato che il comando `apt-get download` scarica i pacchetti nella cartella corrente, è meglio creare una cartella apposita in cui verranno scaricati i pacchetti. A questo punto si utilizza il file con l'elenco dei pacchetti precedentemente esportato dalla macchina virtuale con la release 2.10, quindi i comandi per scaricare i pacchetti della release sono i seguenti:

```
$ mkdir 2.10
$ cd 2.10
$ apt-get update
$ apt-get download $(cat <path all'elenco dei pacchetti>)
```

Nel caso della versione 2.10, i pacchetti da scaricare sono 1881 per un totale di 1,1 GB, quindi ci vorrà del tempo per completare il download di tutti i pacchetti.

Il procedimento appena descritto permette di ottenere i pacchetti di qualsiasi release di Tails. È sufficiente avere una macchina virtuale della release in questione per esportare l'elenco dei pacchetti, e configurare le fonti APT per quella specifica release.

6.3.3 Pkghash

Ora che abbiamo a disposizione tutti i pacchetti di una release di Tails, è necessario costruire i dati di verifica di riferimento a partire da questi. Per questo scopo è stato sviluppato il tool *pkghash*, il quale elabora un insieme di pacchetti Debian, e per ogni pacchetto che installa degli eseguibili, crea un file contenente i percorsi dove gli eseguibili vengono installati e il corrispondente valore hash SHA256 del file.

Il tool distingue i pacchetti contenenti eseguibili in base ai percorsi del file system in cui installa i suoi file. In particolare i file che ci interessa verificare sono i file eseguibili dei comandi e le librerie condivise. Si è preso come riferimento lo standard *Filesystem Hierarchy Standard* [11] che specifica lo scopo e i contenuti delle varie cartelle presenti in un sistema che aderisce allo standard come Debian:

/bin, /sbin, /usr/bin, /usr/sbin
contengono i binari degli eseguibili per i comandi usati da terminale

/lib, /usr/lib
contengono le librerie condivise usate dai comandi e i moduli del kernel

Il programma `pkghash`, prende come parametro il percorso alla cartella contenente i pacchetti Debian da elaborare. Dopo un iniziale controllo sulla validità del percorso preso in input, vengono enumerati i file con estensione .deb presenti nella cartella. Per ogni pacchetto nella cartella, viene controllato se questo installa dei file ai percorsi appena elencati, in caso positivo, i suoi contenuti vengono estratti e si procede al calcolo degli hash SHA256 di tali file. Il tool crea una sotto cartella *PackageHashes* nel percorso preso come input, dove vengono salvati i file contenenti i dati di verifica dei pacchetti.

La classe *PkgHasher*, visibile in Figura 6.1, è il componente che si occupa dell'elaborazione dei pacchetti Debian e della produzione dei file di verifica. Prima di usare la classe è necessario inizialmente enumerare i nomi dei file dei pacchetti nella cartella presa come parametro e cambiare la cartella corrente in quella contente i pacchetti, perché la classe assume che i pacchetti sono nella cartella corrente. Il costruttore prende in ingresso un `vector<string>` contenente i nomi dei file dei pacchetti Debian da elaborare, crea nella cartella dei pacchetti 2 sotto cartelle, una chiamata *PackageHashes* dove salverà i dati prodotti, e una chiamata *extract* dove estrarrà i contenuti dei pacchetti. Una volta creata l'istanza di `PkgHasher`, è sufficiente chiamare il metodo `hashPackages`, e la classe si occuperà di elaborare i pacchetti del *vector* passato come parametro. Il metodo `hashPackages` itera sul contenuto del parametro preso nel

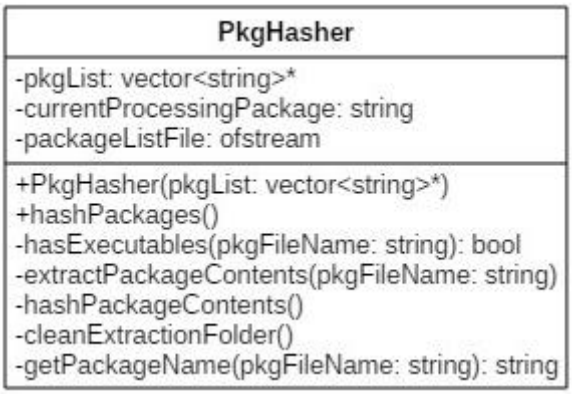


Figura 6.1: Diagramma della classe PkgHasher

costruttore, e per ogni pacchetto controlla se questo contiene degli eseguibili, in

caso positivo estrae i contenuti, calcola gli hash sui file di interesse e salva l'output in un file chiamato con il nome del pacchetto. Il metodo `hasExecutables` controlla se il pacchetto, il cui nome è passato come parametro, installa dei file nei percorsi degli eseguibili e librerie, per questo viene utilizzato il comando `dpkg-deb` [82] con l'opzione `--contents` che elenca appunto i file contenuti in un pacchetto Debian e dove questi verranno installati.

Per i pacchetti che contengono eseguibili, viene chiamato il metodo `extractPackageContents`, il quale estrarrà i suoi contenuti nella cartella `extract`. Usando il comando `dpkg-deb` con l'opzione `-x`, vengono estratti i contenuti di un pacchetto. Una volta terminata l'estrazione, si procede al calcolo degli hash dei contenuti con il metodo `hashPackageContents`.

Il calcolo degli hash avviene usando il programma `hashdeep` [83], che viene eseguito dal metodo `hashPackageContents`, all'interno della cartella `extract` contenente i file estratti dal pacchetto attualmente in elaborazione. L'utility di hashing viene eseguita utilizzando diverse opzioni e parametri, ed il suo output è filtrato con i comandi `tail` [85] e `cut` [86]:

```
hashdeep -c sha256 -l -r -o f bin/ sbin/ usr/bin usr/sbin usr/lib
   2>/dev/null | tail -n +6 | cut -d, -f2-
```

- le opzioni -c sha256 indicano di usare l'algoritmo SHA256 per il calcolo degli hash
- l'opzione -r indica di attraversare ricorsivamente le cartelle
- l'opzione -l è necessaria per far stampare nell'output i percorsi relativi invece degli assoluti
- infine l'opzione -o f indica di calcolare gli hash solamente sui file regolari
- dopo le opzioni segue l'elenco delle cartelle in cui il programma dovrà cercare i file su cui calcolare gli hash, visto che un pacchetto non installa file necessariamente in tutti i percorsi elencati, il programma stamperà dei messaggi di errore per le cartelle inesistenti, perciò viene aggiunto `2>/dev/null` per ignorare semplicemente questi errori e non farli comparire nell'output del programma
- l'output del programma di hashing viene inviato al comando `tail` per eliminare le prime 6 righe di output che non sono utili perché queste contengono informazioni su come il comando hashdeep è stato invocato, elencando le opzioni usate e il percorso alla cartella in cui è stato invocato

- infine l'output del comando tail è una serie di righe del tipo

  `<dimensione file>,<valore hash>,<percorso relativo al file>`

  il comando `cut` elimina dall'output il primo campo con la dimensione del file

Il comando di hashing stamperà i percorsi relativi dei file rispetto alla cartella di estrazione, quindi è sufficiente aggiungere il carattere "/" all'inizio di ogni percorso per trasformarlo in un percorso assoluto a dove il file verrà installato. Ogni riga di output viene quindi ulteriormente elaborata per essere trasformata nel formato:

`<valore hash><doppio spazio vuoto><percorso assoluto di installazione>`

Infine ogni riga elaborata, viene scritta in un file chiamato con il nome del pacchetto da cui i file provengono. Tutti i file prodotti vengono salvati nella cartella `PackageHashes`. I file formattati nel modo appena descritto, permettono di essere dati in input al programma `sha256sum` [84] che verificherà l'integrità dei file elencati.

Al termine dell'elaborazione di ogni pacchetto e prima di iniziare ad elaborarne un altro, il metodo `cleanExtractionFolder` eliminerà i file estratti dal pacchetto nella cartella `extract`. Dopo la pulizia della cartella si procede a ripetere le operazioni descritte sul prossimo pacchetto nell'elenco `pkgList`.

I nomi dei pacchetti vengono estratti dal nome del file .deb, in quanto questi seguono lo schema `nome_versione_architettura`, quindi è sufficiente prendere la prima parte fino al primo carattere `_` (underscore).

Viene prodotto inoltre un file indice chiamato `packages.list`, nella cartella `PackageHashes`, in cui vengono scritti i nomi dei pacchetti che hanno prodotto dei file di verifica. Il file di indice sarò utile al client, dato che non tutti i pacchetti produrranno un file di verifica.

Una volta che `pkghash` ha terminato l'esecuzione, la cartella contenente i pacchetti, data inizialmente come parametro, conterrà la cartella `PackageHashes` in cui ci saranno i file di verifica prodotti e il file indice `packages.list`.

6.4 Test di pkghash

La Tabella 6.1 mostra i risultati ottenuti eseguendo il tool su diversi set di pacchetti. I risultati sono promettenti dato che permette di verificare la maggior parte dei pacchetti e soprattutto le parti più importanti, ovvero i file eseguibili e le librerie, tralasciando file di configurazione, documentazione e di

dati in generale. Di tutti i file presenti nel file system in un sistema Debian, la maggior parte vengono distribuiti con dei pacchetti, quindi questo fattore ha reso possibile la produzione di un set di dati così vasto e in grado di coprire la maggior parte dei file cruciali del sistema.

Tabella 6.1: Risultato esecuzione `pkghash` su set di pacchetti di diverse versioni di Tails

Versione Tails	Pacchetti installati	Pacchetti con file eseguibili	Totale file eseguibili
2.9.1	1850	1381	26732
2.10	1882	1411	27302
3.0-beta1	1887	1410	25336

È stato fatto anche il test sulla prima release beta della versione 3.0 perché questa segna importanti cambiamenti. Infatti la versione 3.0 è basata sulla prossima release stabile di Debian, nome in codice *stretch*, ormai prossima al rilascio. Inoltre viene abbandonato il supporto per l'architettura *i386* per passare alla versione a 64 bit *amd64* dato che al giorno d'oggi è l'architettura predominante.

Le versioni di Tails rilasciate fino ad oggi, sono compatibili solo con l'architettura i386, tuttavia ogni versione ha al suo interno 2 kernel: uno a 32 bit e uno a 64. Visto che la maggior parte dei computer odierni hanno processori a 64 bit, Tails a tempo di avvio caricherà la versione a 64 bit del kernel se il processore è a 64 bit.

Osservando la Tabella 6.1 si può notare che nonostante i pacchetti installati e quelli con file eseguibili, dalla versione 2.10 alla beta 3.0 siano rimasti all'incirca gli stessi, c'è un calo di oltre 2000 file eseguibili nella versione 3.0. Dato che anche il kernel è distribuito con un pacchetto Debian, e i suoi moduli sono installati al percorso `/lib/modules/`, questi fanno parte dei file eseguibili da verificare, ma in quanto la versione 3.0 è compatibile solo con processori a 64 bit, il kernel a 32 bit è assente in questa versione. Il calo quindi è dovuto all'assenza della versione a 32 bit del kernel con i suoi 3000 e più file.

6.4.1 Pubblicazione dei dati di verifica

Come detto in precedenza, la distribuzione dei dati di verifica avverrà con un server web. Il motivo di questa scelta è dovuto al fatto che il set di dati è semplicemente un insieme di file di testo, ed il protocollo HTTP offre tutto ciò che serve per il trasferimento di file. Inoltre se si dispone di un certificato SSL/TLS valido, il server web può essere configurato per comunicare con i client in maniera sicura usando HTTPS, in questo modo il client potrà autenticare il server e avere la certezza che i dati di verifica non siano stati

alterati durante il transito. Il dataset potrà essere facilmente pubblicato su molteplici server web, in modo che gli utenti possano scegliere tra più fonti di dati di riferimento per la verifica. Nel caso in cui il client utilizzi il protocollo HTTP, i dati sono esposti ad alterazione durante il transito in rete, tuttavia considerando che il traffico di rete transita attraverso la rete Tor, e non è noto a priori quale server l'utente sceglierà di usare, diventa estremamente difficile se non impossibile prevedere dove i dati transiteranno su Internet per poterli intercettare e modificare.

A scopo di test, è stato registrato il dominio gratuito `pkgcheck.altervista.org` sul sito di hosting `altervista.org`. In maniera simile ai repository APT, i set di dati di ogni versione, risiederà in una apposita cartella alla radice del server web, in particolare il set di dati relativo alla versione 2.10 sarà disponibile all'URL `http://pkgcheck.altervista.org/2.10/`, allo stesso modo i dati della versione 3.0-beta1 sono disponibili all'URL `http://pkgcheck.altervista.org/3.0~beta1/`. Questo tipo di organizzazione permette di pubblicare diversi set di dati contemporaneamente, rendendo possibile quindi la verifica di più versioni di Tails.

La pubblicazione consiste semplicemente nel caricare i contenuti della cartella `PackageHashes`, prodotta dal tool `pkghash`, usando un client FTP e posizionando i file nella corretta cartella sul server web in modo che il suo nome coincida con la versione di Tails a cui i dati fanno riferimento.

Il dominio registrato attualmente offre i dati di verifica per le versioni 2.9.1, 2.10 e la versione beta 3.0.

6.5 Client

Ora che abbiamo disponibili i dati di verifica di diverse versioni, il client dovrà richiedere i file di verifica in base alla versione di Tails su cui è in esecuzione. Per poter accedere a Internet, il client dovrà usare la rete Tor, altrimenti le richieste di connessione potrebbero essere rifiutate.

Visto che il servizio di back-end offre i dati di verifica solo per i pacchetti che contengono eseguibili, il client dovrà richiedere i dati solamente per tali pacchetti. A questo scopo è disponibile il file indice `packages.list` in ogni set di dati.

6.5.1 Pkgcheck

Il tool lato client che esegue la verifica, si chiama `pkgcheck`. Il programma prende come parametro l'URL del server web da cui ottenere i dati di verifica, ed esegue la verifica dei pacchetti installati di cui sono disponibili i dati sul server.

Il tool deve conoscere la versione di Tails sulla quale è in esecuzione per poter indirizzare il corretto set di dati per la verifica. Ogni versione di Tails ha un file di testo al percorso `/etc/amnesia/version`, in cui è scritta la versione di Tails, quindi il tool non dovrà far altro che leggere tale file per sapere la versione su cui è in esecuzione.

Per quanto riguarda la comunicazione con il server web, è stato scelto di utilizzare la libreria `libcurl` [87] perché disponibile su tutte le versioni di Tails. La libreria offre una semplice interfaccia per il protocollo HTTP e supporta in maniera trasparente l'uso di HTTPS, infatti è sufficiente usare degli URL `https://` al posto di `http://` per abilitare l'uso di SSL/TLS. Inoltre la libreria supporta tutte le versioni del protocollo SOCKS, perciò potrà usare la rete Tor per effettuare le richieste HTTP.

Per rendere più agevole l'uso della libreria, si è preferito costruire una classe wrapper che incapsula la configurazione e il suo utilizzo. In Figura 6.2 è possibile vedere il diagramma delle classi della classe `HTTPClient` che implementa il wrapper attorno alla libreria. Il costruttore si occupa di inizializzare la libreria, allocando la struct CURL che permette di configurare i parametri delle richieste, e configura la struttura per usare il proxy SOCKS di Tor, il quale è in ascolto su `localhost:9050` in Tails. Il metodo `makeGET` prende in ingresso l'URL a cui effettuare la richiesta e un puntatore a `long`, esegue la richiesta HTTP e ritorna una stringa contenente il corpo della risposta, collocando il codice di risposta HTTP nel puntatore a long.

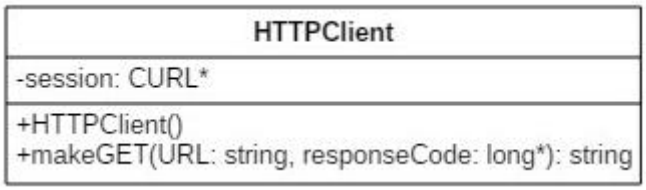


Figura 6.2: Diagramma della classe HTTPClient

Un grande vantaggio di libcurl è che rende possibile l'uso di connessioni persistenti, semplicemente riutilizzando sempre la stessa istanza di struttura CURL, e dato che nel nostro caso le richieste saranno effettuate sempre allo stesso host, nel metodo `makeGET` la struttura viene riutilizzata cambiando solamente l'URL. Il client della classe `HTTPClient` dovrà creare una sola istanza della classe e riutilizzarla per fare più richieste in modo che libcurl possa utilizzare le connessioni persistenti.

Il componente che si occupa di caricare i dati di verifica e verificare i pacchetti è implementato con la classe `PkgCheck` il cui diagramma è visibile in Figura 6.3. Prima di creare un'istanza di `PkgChecker`, viene letto il file della versione di Tails dato che il costruttore prende come parametri una stringa

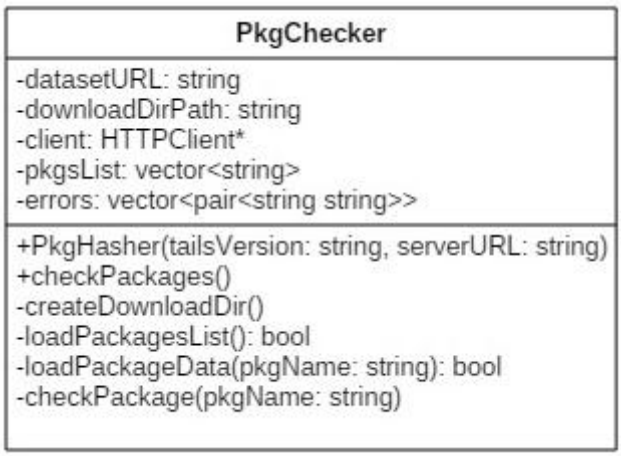


Figura 6.3: Diagramma della classe PkgChecker

con la versione di Tails sulla quale è in esecuzione, e l'URL del server da cui ottenere i dati di verifica, che l'utente ha dato come parametro al programma. Il costruttore inizializza il campo `datasetURL` concatenando l'URL del server alla versione di Tails, dato che questo sarà l'URL di partenza per richiedere i file della corretta versione di Tails. Durante la costruzione di `PkgChecker` viene creata l'istanza di `HTTPClient` che verrà riutilizzata per effettuare tutte le richieste al server, in modo da permettere l'uso di una connessione persistente da parte di libcurl. Il cliente delle classe `PkgChecker`, una volta costruita l'istanza, non dovrà fare altro che chiamare il metodo `checkPackages` e la classe effettuerà la verifica di tutti i pacchetti di cui sono disponibili i dati sul server.

Il metodo `checkPackages` inizialmente proverà ad ottenere la lista dei pacchetti disponibili sul server, usando il metodo `loadPackegesList` che richiederà il file indice `packages.list`. Se il file indice viene caricato con successo, il `vector pkgsList` viene inizializzato con i nomi dei pacchetti elencati in esso.

Prima di iniziare la verifica viene creata una cartella temporanea con il metodo `createDownloadDir` dove verranno salvati i file ottenuti dal server. A questo punto la procedura di verifica consiste nel ripetere i seguenti passi per ogni pacchetto nella lista `pkgsList`:

1. **Caricamento dati di verifica del pacchetto** con il metodo `loadPackageData` che richiederà al server il corrispondente file e lo salverà nella cartella temporanea inizialmente creata

2. **Verifica di integrità dei file del pacchetto** con il metodo `checkPackage`, che avvertirà l'utente in caso di errori nella verifica del pacchetto in questione

I file di verifica sono stati formattati da `pkghash` nel formato del programma `sha256sum`, in modo da poter utilizzare tale programma per verificare l'integrità dei file dei pacchetti. Il metodo `checkPackage` quindi userà il programma `sha256sum` a cui darà in input il file caricato dal server, il quale effettuerà la verifica dei file elencati e restituirà output solo in caso di errori.

Per ogni pacchetto che viene verificato, il programma stamperà "OK" in caso di mancanza di errori, mentre in caso contrario stamperà l'output di `sha256sum` che contiene gli errori riscontrati. Gli errori prodotti durante la verifica vengono raccolti e mappati al pacchetto che li ha prodotti, per una stampa degli errori totali a fine dell'esecuzione del programma.

6.6 Test di pkgcheck

Il programma è stato testato sulle versioni di Tails di cui sono stati costruiti i set di dati di verifica, quindi le versioni 2.9.1, 2.10 e 3.0 beta. In Figura 6.4 è possibile vedere un esempio di utilizzo di `pkgcheck` su Tails 2.10, utilizzando i dati sul server web preparato a scopo di test.

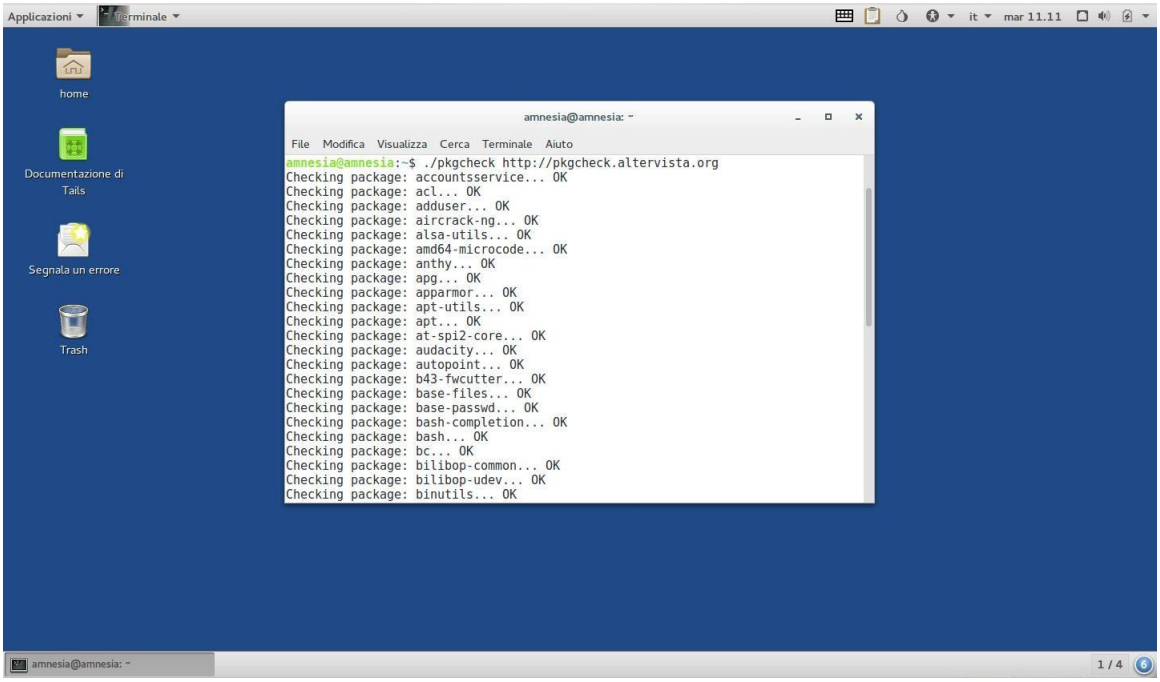


Figura 6.4: Esempio di esecuzione di `pkgcheck`

La Tabella 6.2 mostra i risultati dei test della verifica di integrità sulle versioni di Tails prese in considerazione. Il tool ha impiegato dagli 8 ai 10 minuti per terminare l'esecuzione sulle varie versioni di Tails. La maggior

parte del tempo di esecuzione è impiegato nel caricamento dei file di verifica dal server attraverso Tor, la cui velocità di trasferimento è molto variabile e spesso intermittente, infatti abilitando l'output di debug di libcurl è possibile osservare che la connessione al server cade spesso.

Tabella 6.2: Risultati dell'esecuzione di `pkgcheck` su diverse versioni di Tails

Versione Tails	Totale pacchetti verificati	Pacchetti con errori	Totale file errati
2.9.1	1381	17	31
2.10	1411	17	31
3.0-beta1	1410	11	24

Gli errori prodotti durante la verifica, sono dovuti alle personalizzazioni fatte dagli sviluppatori di Tails e sono dei falsi positivi. Durante la costruzione dell'ISO di Tails, vengono eseguiti diversi script che modificano, rimuovono e spostano diversi file che appartengono a diversi pacchetti. Ad esempio il pacchetto `libpurple0` [88] contiene un insieme di librerie che implementano diversi protocolli di messaggistica istantanea, tuttavia alcuni di questi protocolli non supportano l'uso di un proxy SOCKS e quindi non possono essere usati attraverso Tor, perciò le librerie di tali protocolli vengono rimosse. Nel caso di `libpurple0`, presente in tutte le versioni di Tails, vengono rimosse 13 librerie condivise installate da questo pacchetto e quindi al momento della verifica, `sha256sum` tenterà di calcolare l'hash su dei file inesistenti causando 13 errori, quasi la metà del totale dei file errati.

I falsi positivi non sono prevedibili in maniera automatica perché le modifiche che causano gli errori sono dovute a degli script eseguiti a tempo di costruzione dell'ISO di Tails. Tuttavia gli errori possono essere corretti sfruttando le informazioni restituite sugli errori, per modificare il dataset in base al tipo di errore (assenza del file, mancanza dei permessi di lettura o valore hash errato). Gli errori dovuti all'assenza dei file possono essere risolti eliminando i dati di tali file dal dataset in modo che questi non vengano verificati dal client e quindi non si tenti l'accesso a file inesistenti. Per quanto riguarda la mancanza di permessi, un solo file presente nel dataset è leggibile esclusivamente dall'utente root, perciò solo se all'inizio della sessione viene abilitato l'utente root, sarà possibile verificare tale file eseguendo `pkgcheck` come utente root o rendendo il file leggibile da tutti gli utenti. Infine per i file il cui valore hash non corrisponde, è necessario calcolare il loro valore hash corretto da un'istanza Tails ed aggiornare il dataset con i valori corretti.

Per automatizzare la correzione degli errori, `pkgcheck` può essere esteso per salvare i dati sugli errori in dei file in base al tipo di errore. Gli errori correggibili sono quelli dovuti all'assenza dei file o valori hash errati, quindi per i file assenti, si può produrre un file in cui vengono elencati i percorsi dei

file assenti, mentre per gli hash errati è necessario produrre un file contenente gli hash corretti e i percorsi ai file a cui fanno riferimento. A questo punto con un programma di elaborazione del testo, i due file possono essere utilizzati per correggere gli errori nel dataset, eliminando prima i dati di verifica dei file assenti e poi sostituendo i valori hash errati con quelli corretti.

Conclusioni

Il sistema sviluppato ha reso possibile raggiungere l'obbiettivo inizialmente preposto, permettendo di verificare l'integrità dei software installati nella distribuzione Tails. In particolare la prima parte del lavoro è stata dedicata alla realizzazione del set di dati di riferimento per la verifica. Considerando che una nuova release di Tails avviene a periodi abbastanza brevi, doveva essere possibile costruire il set di dati per ogni nuova release e rendere possibile la verifica di più release. Dato che Tails rende disponibile i pacchetti Debian installati in ogni release, è stato realizzato il tool `pkghash` che costruisce il set di dati per la verifica a partire da un'insieme di pacchetti Debian. Il tool include nel set di dati solamente le parti più rilevanti da verificare, ovvero i file eseguibili e le librerie condivise, in quanto queste sono le parti critiche dei sistemi software ed i principali target di attacchi. I dati di verifica sono distribuiti con un server web che posiziona, ogni set di dati in una cartella che identifica a quale release di Tails fa riferimento tale set. Grazie a questo tipo di organizzazione per la distribuzione dei dati è possibile verificare più versioni di Tails contemporaneamente. Inoltre grazie al fatto che il set di dati è semplicemente un insieme di file, non è necessario alcun tipo di configurazione particolare sul server web.

Il tool `pkgcheck`, che esegue la verifica di integrità, è indipendente dalla particolare release di Tails ed è sufficiente un server web che abbia i dati di verifica per la specifica release. Testando il tool su diverse release, i risultati ottenuti sono soddisfacenti considerando la quantità di errori rispetto alla quantità di file verificati. Gli errori riscontrati sono dei falsi positivi dovuti a delle modifiche che gli sviluppatori di Tails effettuano per proteggere i propri utenti. Dato il numero basso di falsi positivi, è possibile correggerli nel set di dati, estraendo i valori corretti da un'istanza Tails.

Attualmente `pkgcheck` è dipendente da elementi presenti in Tails per la richiesta dei dati (la libreria libcurl) e per la verifica di integrità (il comando `sha256sum`). Considerando che il tool è nato per verificare alterazioni malevole dei software, non è da escludere che le sue dipendenze possano essere alterate. Nonostante sia garantito che le dipendenze saranno soddisfatte su tutte le versioni di Tails, uno degli sviluppi futuri per il tool potrebbe essere proprio

la totale rimozione delle dipendenze.

La realizzazione dei tool è stata preceduta da uno studio del sistema operativo Debian in quanto alla base di Tails. Debian ha dato un grande contributo al mondo del software, dando vita ai concetti di pacchetto software e gestore di pacchetti, permettendo agli utenti di ottenere e installare il software in maniera semplice e intuitiva. Tuttavia la nascita di questo modello di gestione software ha dato anche vita a nuovi problemi dal punto di vista della sicurezza. È stato interessante analizzare l'infrastruttura di Debian e i meccanismi crittografici che utilizza al fine di garantire l'integrità e l'autenticazione dei pacchetti che entrano nell'archivio e poi vengono distribuiti agli utenti. Gli schemi crittografici utilizzati in Debian sono sicuramente utili e necessari, ma come abbiamo visto non sono esenti da problemi.

Bibliografia

[1] William Stallings, Lawrie Brown, *Computer Security: Principles and Practice (3rd Edition)*, Pearson, 18 Luglio 2014.

[2] Tails, `https://tails.boum.org/index.en.html`.

[3] Debian, `https://www.debian.org/`.

[4] Raphaël Hertzog, Roland Mas, *Il Manuale dell'Amministratore Debian, Debian Jessie dalla Scoperta alla Padronanza*, Freexian SARL, 21 ottobre 2015, `https://debian-handbook.info/browse/it-IT/stable/index.html`.

[5] Ian A. Murdock, *Il manifesto Debian*, 1994.

[6] Linux kernel, `https://www.kernel.org`.

[7] GNU, `https://www.gnu.org/`.

[8] Ean Schuessler, Bruce Perens, Debian Developers, *Debian Social Contract*, `https://www.debian.org/social_contract`.

[9] Open Source Initiative, *The Open Source Definition (Annotated)*, `https://opensource.org/osd-annotated`.

[10] Ian Jackson, Christian Schwarz, Russ Allbery, Bill Allombert, Andrew McMillan, Manoj Srivastava, Colin Watson, *Debian Policy Manual*, 2013, `http://www.debian.org/doc/debian-policy/policy.pdf.gz`.

[11] Filesystem Hierarchy Standard, `http://www.pathname.com/fhs/`.

[12] Michael Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010, p. 767.

[13] crontab, `http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html`.

[14] buildd, https://www.debian.org/devel/buildd/.

[15] Debian Release Team, https://wiki.debian.org/Teams/ReleaseTeam.

[16] Ronald Rivest, *The MD4 Message-Digest Algorithm*, RFC 1320 https://tools.ietf.org/html/rfc1320.

[17] Ronald Rivest, *The MD5 Message-Digest Algotithm*, RFC 1321 https://tools.ietf.org/html/rfc1321.

[18] Xiaoyun Wang, Hongbo Yu, *How to Break MD5 and Other Hash Functions*, 2005.

[19] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu, *Finding collisions in the Full SHA-1*, 2005. http://www.iacr.org/cryptodb/archive/2005/CRYPTO/1826/1826.pdf.

[20] Philip Zimmermann, *Pretty Good Privacy*, http://philzimmermann.com/EN/background/index.html.

[21] J. Callas et al, *OpenPGP Message Format*, RFC4880 https://tools.ietf.org/html/rfc4880.

[22] Linux man page, *bash(1)*, https://linux.die.net/man/1/bash.

[23] Linux man page, *ar(1)*, https://linux.die.net/man/1/ar.

[24] Linux man page, *deb(5)*, https://linux.die.net/man/5/deb.

[25] Linux man page, *deb-shlibs(5)*, https://linux.die.net/man/5/deb-shlibs.

[26] Linux man page, *deb-symbols(5)*, https://linux.die.net/man/5/deb-symbols.

[27] Wichert Akkerman, Joey Hess, *Configuration management. Protocol version 2.1*, The Debian Project. https://www.debian.org/doc/packaging-manuals/debconf_specification.html.

[28] Ubuntu man page, *debconf(7)*, http://manpages.ubuntu.com/manpages/zesty/man7/debconf.7.html.

[29] Linux man page, *dpkg(1)*, https://linux.die.net/man/1/dpkg.

[30] Manoj Srivastava, *Maintainer scripts*, https://people.debian.org/~srivasta/MaintainerScripts.html.

[31] Linux man page, *apt-get(8)*, `https://linux.die.net/man/8/apt-get`.

[32] Linux man page, *sources.list(5)*, `https://linux.die.net/man/5/sources.list`.

[33] Debian Wiki, *Repository Format*, `https://wiki.debian.org/RepositoryFormat`.

[34] Debian Wiki, *Secure Apt*, `https://wiki.debian.org/SecureApt`.

[35] Ubuntu man page, *apt-secure(8)*, `http://manpages.ubuntu.com/manpages/wily/it/man8/apt-secure.8.html`.

[36] GNU Privacy Guard, `https://www.gnupg.org/`.

[37] Debian Wiki, *Source-only uploads*, `https://wiki.debian.org/SourceOnlyUpload`.

[38] Debian Wiki, *Reproducible Builds*, `https://wiki.debian.org/ReproducibleBuilds`.

[39] Ubuntu man page, *dpkg-sig(1)*, `http://manpages.ubuntu.com/manpages/trusty/man1/dpkg-sig.1.html`.

[40] Man pages, *debsigs(1)*, `http://manpages.org/debsigs/1`.

[41] Man pages, *debsig-verify(1)*, `http://manpages.org/debsigs/1`.

[42] Wikipedia, *Trojan*, `https://it.wikipedia.org/wiki/Trojan_(informatica)`.

[43] Google Linux Software Repositories, `https://www.google.com/linuxrepositories/`.

[44] Ken Thompson, *Reflections on Trusting Trust*, Communications of the ACM, Agosto 1984. `http://dl.acm.org/citation.cfm?id=358210`.

[45] Linux man page, *login(1)*, `http://man7.org/linux/man-pages/man1/login.1.html`.

[46] Linux Mint, `https://www.linuxmint.com/`.

[47] Ubuntu, `https://www.ubuntu.com/`.

[48] Lohit Mehta. *Rootkits: User Mode*, `http://resources.infosecinstitute.com/rootkits-user-mode-kernel-mode-part-1/`.

[49] Linux man page, *sshd(8)*, `https://linux.die.net/man/8/sshd`.

[50] Linux man page, *ps(1)*, `https://linux.die.net/man/1/ps`.

[51] Linux man page, *netstat(8)*, `https://linux.die.net/man/8/netstat`.

[52] Linux man page, *ls(1)*, `https://linux.die.net/man/1/ls`.

[53] Daniel D. Nerenberg, *A Study of Rootkit Stealth Techniques and Associated Detection Methods*, Sezione 2.6 Rootkits, pp. 17-20, `http://www.dtic.mil/dtic/tr/fulltext/u2/a519999.pdf`.

[54] Man pages, *debsums(1)*, `http://manpages.org/debsums`.

[55] Open Source Tripwire, `https://github.com/Tripwire/tripwire-open-source`.

[56] L. Catuogno, I. Visconti, *A Format-Independent Architecture for Run-Time Integrity Checking of Executable Code*, Security in Communication Networks. SCN 2002. Lecture Notes in Computer Science vol 2576 pp. 219-233. Springer. 2002.

[57] W.A. Arbaugh, G. Ballintijn, L. van Doorn, *Signed Executables for Linux*. Tech. Report CS-TR-4259. University of Maryland. 2001.

[58] The DigSig development team, *The DigSig project*, `http://disec.sourceforge.net/docs/digsig.pdf`, 2005.

[59] Debian Packages, *bsign*, `https://packages.debian.org/jessie/bsign`.

[60] Tails, *Design: specification and implementation*, `https://tails.boum.org/contribute/design/`.

[61] Tor Project, *Tor*, `https://www.torproject.org/`.

[62] M. Leech et al, *SOCKS Protocol Version 5*, RFC 1928, `https://tools.ietf.org/html/rfc1928`.

[63] Reproducible Builds, *Reproducible Builds Documentation*, `https://reproducible-builds.org/docs/`.

[64] Claud Xiao, *Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store*, Palo Alto Networks. 2015. `http://researchcenter.paloaltonetworks.com/2015/09/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/`.

[65] Roman Hodek, Philipp Kern, *Outline of operation of the autobuilder network*, https://www.debian.org/devel/buildd/operation.html.

[66] Bitcoin Core, https://bitcoin.org/.

[67] Gitian, https://gitian.org/.

[68] Debian Wiki, *Reproducible Builds Buildinfo Specification*, https://wiki.debian.org/ReproducibleBuilds/BuildinfoSpecification.

[69] Overview of various statistics about reproducible builds, https://tests.reproducible-builds.org/debian/reproducible.html.

[70] Tails, *Features and included software*, https://tails.boum.org/doc/about/features/index.en.html.

[71] Linux man page, *synaptic(8)*, https://linux.die.net/man/8/synaptic.

[72] Vagrant, https://www.vagrantup.com/.

[73] Live Systems Project, *Live Systems Manual*, https://debian-live.alioth.debian.org/live-manual/stable/manual/html/live-manual.en.html.

[74] Tails, *APT repository*, https://tails.boum.org/contribute/APT_repository/.

[75] Squashfs, http://squashfs.sourceforge.net/.

[76] Wikipedia, *Cold boot attack*, https://en.wikipedia.org/wiki/Cold_boot_attack.

[77] Ian Murdock, *Debian announcement*, flickr, https://www.flickr.com/photos/iamurdock/20006308374/in/photostream/.

[78] Debian Packages, *coreutils*, https://packages.debian.org/stable/coreutils.

[79] The Eclipse Foundation, *Eclipse IDE*, https://www.eclipse.org/.

[80] Linux man page, *dpkq-query(1)*, http://man7.org/linux/man-pages/man1/dpkg-query.1.html.

[81] Linux man page, *sed(1)*, https://linux.die.net/man/1/sed.

[82] Linux man page, *dpkg-deb(1)*, http://man7.org/linux/man-pages/man1/dpkg-deb.1.html.

[83] Linux man page, *hashdeep(1)*, https://linux.die.net/man/1/hashdeep.

[84] Linux man page, *sha256sum(1)*, https://linux.die.net/man/1/sha256sum.

[85] Linux man page, *tail(1)*, http://man7.org/linux/man-pages/man1/tail.1.html.

[86] Linux man page, *cut(1)*, http://man7.org/linux/man-pages/man1/cut.1.html.

[87] Daniel Stenberg et al, *libcurl*, https://curl.haxx.se/libcurl/.

[88] Debian Packages, *libpurple0*, https://packages.debian.org/jessie/libpurple0.