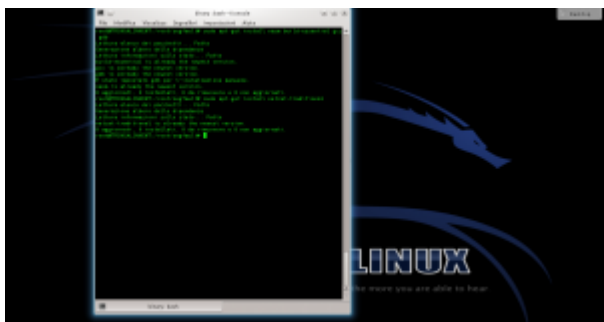


Hacking&Cracking: Buffer overflow, un tutorial passo passo

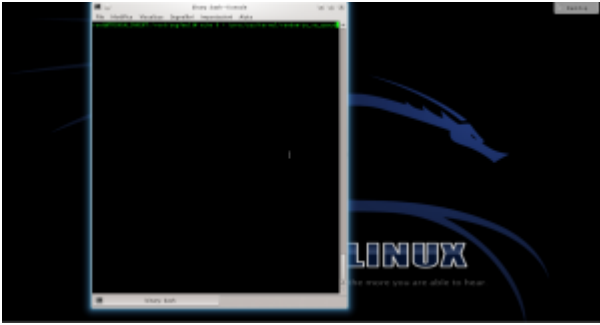
Nella puntata precedente di questa mini-serie (<https://www.codice-sorgente.it/2019/06/buffer-overflow-e-errori-di-segmentazione-della-memoria/>) abbiamo descritto il funzionamento della memoria di un computer, e in particolare gli overflow nello stack. In questo breve articolo presentiamo un tutorial passo passo per l'analisi di un programma buggato e lo sfruttamento della sua vulnerabilità per ottenere l'esecuzione di codice. Seguiremo la stessa procedura dell'articolo precedente, ma con una serie di screenshot che spiegano meglio i vari passaggi.

La preparazione

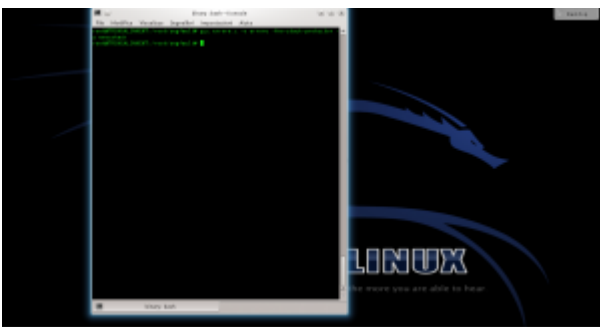
Per testare questi esempi bisogna innanzitutto avere a disposizione un sistema operativo a 32 bit, possibilmente su una macchina virtuale per mantenere stabile il proprio sistema host. Bisogna poi disabilitare alcune norme di sicurezza di Linux, altrimenti l'analisi della vulnerabilità e l'esecuzione dell'exploit non saranno per nulla facili.



Per prima cosa ci si deve assicurare che sul sistema sia installato il necessario per compilare del codice: lo si può fare dando il comando



Per disabilitare la protezione del kernel Linux, possiamo dare il comando `sysctl kernel.yama.stack-limit=0`. Questo non è necessario con Linux precedente al 2.6.12, anche se ormai è difficile trovare sistemi così vecchi su dispositivi ancora attivi.



Bisogna ora procurarsi il programma buggato: per esempio, si può scaricare il file `errore.c` (<https://pastebin.com/8DZQZzqx>). Il programma va compilato con il comando

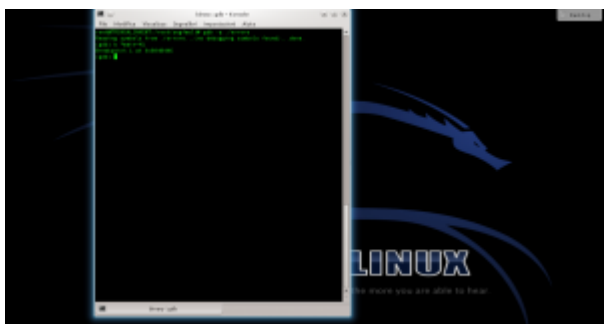
In questo modo, il programma viene compilato senza le protezioni per lo stack inserite automaticamente da GCC. Naturalmente, si potrebbe fare la stessa cosa con qualsiasi altro programma, utilizziamo questo solo perché è molto semplice e quindi è facile capire come funziona.

Analizzare il programma vulnerabile

In questo particolare caso possiamo leggere il codice del programma, perché è open source, ed è anche estremamente breve.

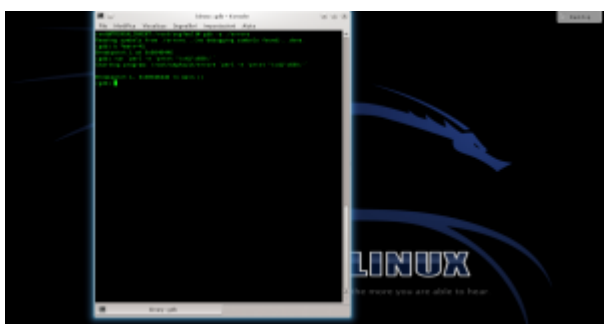
In una situazione reale il codice sorgente potrebbe non essere

disponibile. Ad ogni modo, il codice ci serve più che altro per capire se ci sia un bug e dove si trovi: possiamo facilmente capire che la vulnerabilità sta nell'assenza di un controllo sulla dimensione dell'argomento del programma, che viene caricato in un buffer da 500 byte senza però prima verificare se l'argomento in questione abbia una lunghezza maggiore di 500 byte.

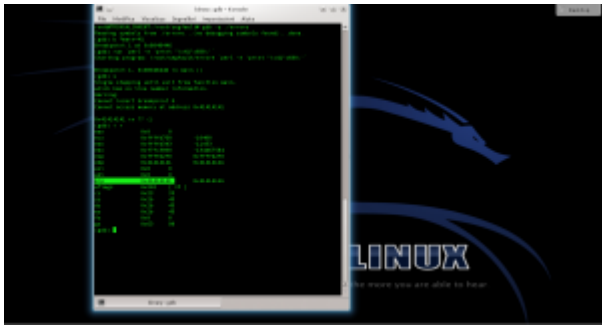


Ora, dobbiamo studiare il programma vulnerabile per capire quali indirizzi di memoria possiamo utilizzare. Serve un debugger quindi, supponendo di voler utilizzare il programma "errore" precedentemente compilato, il pirata da il comando Aperto il debugger, possiamo disassemblare il programma per leggere il suo codice assembly col comando e otterremo un listo di questo tipo.

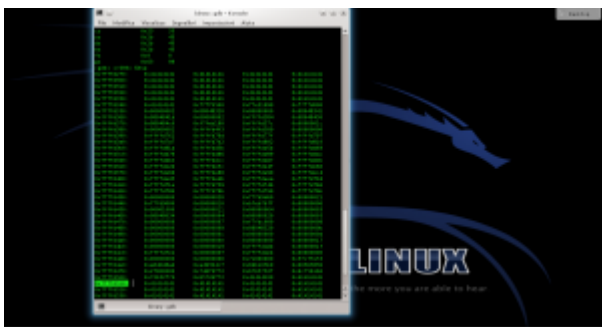
Dal listato si capisce che l'istruzione di ritorno della funzione (**leave**) è nel punto **+41**.



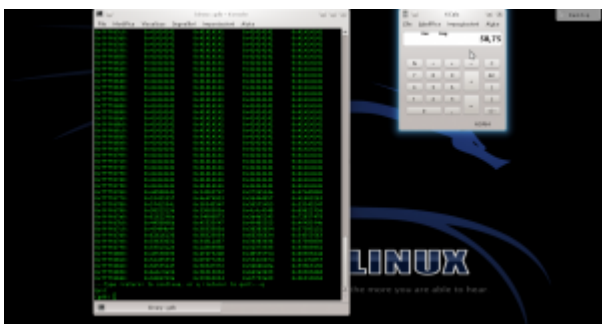
Impostiamo quindi un breakpoint per il controllo del programma prima dell'istruzione di ritorno della funzione buggata, scrivendo Poi proviamo a far crashare il programma fornendogli una stringa di 600 caratteri con il comando



Il programma andrà in crash, perché l'array può contenere solo 500 caratteri. Ma siamo in un debugger, quindi possiamo dare i comandi e poi per poter controllare i registri del processore poco prima del crash. Il registro EIP è stato riempito con 4 byte dal valore 41. EIP è il registro del puntatore per la funzione di ritorno, quindi il programma è andato in crash perché cercava di tornare a una funzione all'indirizzo 0x41414141, che ovviamente non esiste.



Ora diamo il comando per leggere i 600 byte successivi al puntatore ESP. A un certo punto, dovremmo trovare un blocco con tutti i byte di valore 41: l'indirizzo di inizio potrebbe essere, per esempio, **0xffffd510**.

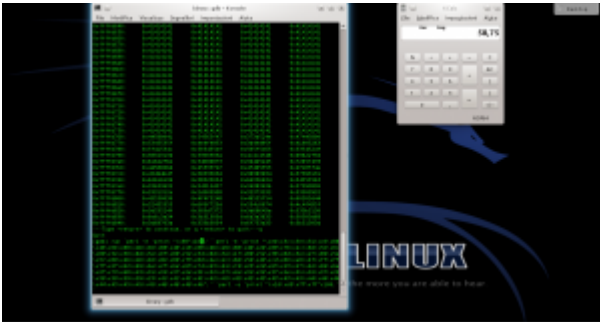


Questo è l'indirizzo in cui sarà inserita la nop sled. Una buona dimensione potrebbe essere 100 byte. Però, lo shellcode è lungo 135 byte, e la somma (235) non è divisibile per 4. Il numero 236, però, lo è. Quindi la nop sled dovrà contenere 101

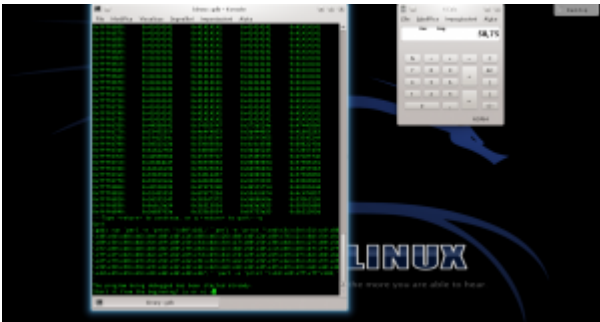
byte, per evitare sfasamenti.

Il payload

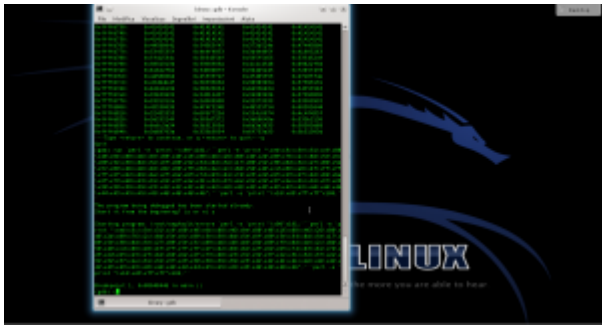
Ormai abbiamo la dimensione della NOP sled e anche l'indirizzo di ritorno. Ci manca soltanto lo shellcode, che possiamo recuperare da un elenco online (come quelli pubblicati su exploit-db.com).



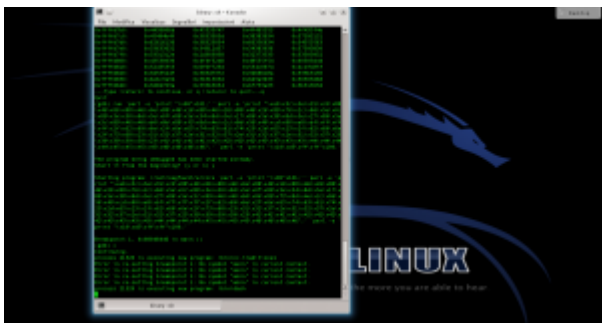
Possiamo quindi scrivere la stringa completa (<https://pastebin.com/biSxHhRT>): 101 byte del carattere NOP (90), seguiti dallo shellcode, e poi dall'indirizzo di ritorno scritto al contrario per mantenere la codifica little endian, ripetuto almeno un centinaio di volte.



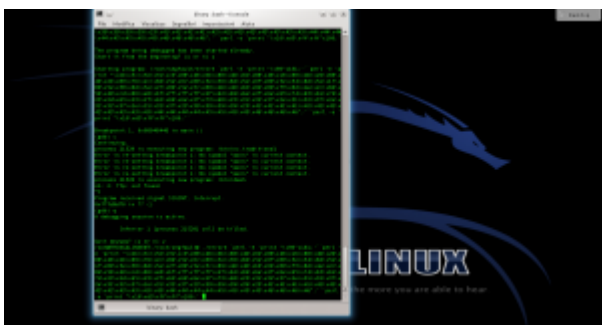
Basta eseguire il programma con il comando seguito dalla stringa completa: ovviamente, GDB chiederà conferma, visto che si deve riavviare il programma attualmente fermo al breakpoint. Digitiamo e il programma viene lanciato di nuovo ma con l'argomento costruito dai vari comandi Perl.



Il programma si fermerà nuovamente al breakpoint, esattamente come prima: se diamo ancora i comandi e dovremmo notare che EIP ha ora il valore **ffffd510**, o comunque un indirizzo nella NOP sled. Possiamo controllare il contenuto della memoria anche col comando



Se poi diamo il comando l'esecuzione del programma continua, ed il codice presente all'indirizzo di ritorno verrà eseguito: dovrebbe apparire il messaggio



Se la stringa funziona, possiamo ormai utilizzarla direttamente, senza gdb, eseguendo il programma con l'intera stringa.

Hacking&Cracking: buffer overflow e errori di segmentazione della memoria

Oggi, con la notevole diffusione dell'informatica e delle reti di computer, la sicurezza dei programmi non può assolutamente essere trascurata. Quando scriviamo un programma, dobbiamo tenere conto del fatto che esistono migliaia di persone (pirati informatici, anche detti "cracker" o "black hat") che cercheranno di utilizzare il nostro programma in modo improprio per ottenere il controllo del computer su cui tale programma viene eseguito. Quindi, dobbiamo realizzare i nostri programmi cercando di impedire che possa essere utilizzati in modo improprio. E, per farlo, dobbiamo conoscere le basi della pirateria informatica. Perché l'unico modo per rendere i nostri programmi non crackabili è sapere come possono essere crackati. Come vedremo, i sistemi moderni (GNU/Linux in particolare) hanno dei meccanismi per difendersi dagli attacchi a prescindere dal programma compromesso, ma è ovviamente meglio se i programmi non sono facilmente crackabili, perché come minimo si rischia un Denial of Service. Che magari per una applicazione desktop è poco importante, ma per un server web diventa un problema notevole. Una indicazione: alcune delle stringhe sono molto lunghe e difficilmente leggibili. Ho deciso di lasciarle così perché affinché il codice funzioni è necessario che non vi sia alcuna interruzione nella stringa, e questo rende più facile copiarle anche se sono scomode da leggere o da stampare.

Un problema di memoria

L'assoluta maggioranza delle vulnerabilità dei programmi riguardano l'utilizzo della memoria. Il problema è intrinseco

alla struttura di un computer: il componente fondamentale di un calcolatore è il processore, ovvero l'unità che esegue i calcoli. Per poter eseguire i calcoli, è necessario disporre anche di una memoria, nella quale memorizzare le informazioni necessarie. Banalmente, se vogliamo sommare due numeri, abbiamo bisogno di avere lo spazio necessario per memorizzare i due numeri in questione in modo da sapere su cosa eseguire l'operazione. Un computer dispone di una memoria molto rapida nota come RAM, che però può avere dimensioni diverse ed essere molto grande (si può facilmente aumentare lo spazio di memoria installando una scheda supplementare). Il processore, tuttavia, deve essere capace di funzionare a prescindere dalla dimensione e natura della memoria RAM, anche perché spesso sono componenti costruiti da aziende diverse. Inoltre, se le informazioni vengono scritte in forma "disordinata" (per rendere la scrittura più rapida) in uno spazio molto grande (diversi GB di memoria) può essere piuttosto difficile trovare le informazioni di cui si ha bisogno in un determinato momento tra tutte le altre informazioni memorizzate. Per questo scopo esistono i registri del processore.



La dimensione dei registri

La dimensione dei registri dipende dal numero di bit che possono contenere: 8, 16, 32, 64. Visto che i registri contengono gli indirizzi di memoria RAM, è ovvio che un registro contenente più bit potrà rappresentare un numero maggiore di diversi indirizzi. Per esempio, un registro a 16 bit potrà contenere al massimo 65536 diversi indirizzi (2 elevato alla 16esima, perché i bit possono avere solo due valori: 0 ed 1). Similmente, con un registro a 32 bit si potranno esprimere fino a 4294967296 indirizzi differenti (poco più di 4 miliardi, tradotto in termini di byte corrisponde a 4096 MB oppure esattamente 4GB), mentre con 64

bit a disposizione si può arrivare fino a circa $1,84 \cdot 10^{19}$ (cioè 184 seguito da 19 zeri). Ciò significa che se abbiamo dei registri a 32 bit, potremo considerare al massimo 4 GB: anche se disponiamo di una memoria da 8GB potremo utilizzarne soltanto la metà, perché non abbiamo abbastanza indirizzi per tutte le celle della memoria.

I registri sono un tipo di memoria di dimensioni ridotte e ad alta velocità. Le loro dimensioni ridotte fanno sì che in genere non vengano utilizzati per memorizzare le informazioni vere e proprie: queste vengono inserite nella memoria RAM. Nei registri vengono inseriti i puntatori alle celle della memoria RAM. Facciamo un esempio concreto: dobbiamo memorizzare il numero "1", fondamentalmente una variabile in uno dei nostri programmi. Tale numero viene memorizzato in una particolare cella della RAM, che viene contraddistinta dall'indirizzo: si tratta di un numero assegnato univocamente a tale cella. Per esempio, potrebbe essere al numero 6683968 o, scritto in base esadecimale, 0x0065fd40. A questo punto basta memorizzare in un registro del processore l'indirizzo 0x0065fd40, ed ogni volta che avremo bisogno di lavorare con il contenuto di tale cella della RAM il processore saprà esattamente a che indirizzo trovarla. Funziona un po' come la mappa di una città: ogni abitazione può contenere delle persone, ed ogni abitazione è contraddistinta da un indirizzo preciso. Se cerchiamo una particolare persona, non dobbiamo fare altro che cercare nell'apposito elenco il suo indirizzo, e sapremo in quale casa trovarla. Naturalmente, questo meccanismo diventa particolarmente vantaggioso quando vogliamo memorizzare molti bit di informazioni, perché nel registro del processore si inserisce soltanto l'indirizzo del primo bit.

Il processore procederà poi a leggere tale bit presente nella RAM assieme alle migliaia di bit che lo seguono finché non gli viene ordinato di smettere. Quindi, semplicemente utilizzando gli indirizzi, possiamo "riassumere" in un singolo numero piuttosto piccolo (il numero dell'indirizzo, per l'appunto) porzioni molto grandi della memoria RAM.

Diversi tipi di registri

I registri disponibili in un processore con architettura x86 sono divisi in quattro categorie: i registri generali, i registri di segmento, i registri dei puntatori, e gli indicatori. I registri interessanti sono quelli generali e quelli dei puntatori. Questo tipo di registri, nei sistemi x86, sono una evoluzione dei corrispondenti registri presenti nei sistemi ad 8 e 16 bit. Infatti, i registri generali di un sistema ad 8 bit sono:

- A
- B
- C
- D

In un sistema a 16 bit i registri corrispondenti sono:

- AX
- BX
- CX
- DX

Ed infine in un sistema x86 i registri generali sono i seguenti:

- EAX
- EBX
- ECX
- EDX

Il bello è che il funzionamento dei registri è identico: ovvero, il codice macchina da fornire al processore per scrivere nel registro A è lo stesso che si utilizza per scrivere nel registro EAX, basta sostituire il nome del registro cui fare accesso. Quindi le regole che presentiamo nelle prossime pagine valgono per tutti i sistemi (inclusi quelli a 64 bit, grazie alla retrocompatibilità dell'architettura x86_64). Parlando dei sistemi x86, che sono ovviamente i più interessanti per noi programmatori in quanto più diffusi al giorno d'oggi, i compiti dei vari registri generali sono i seguenti:

- EAX: anche chiamato "Accumulatore", è utilizzato per accedere agli input/output, le operazioni aritmetiche, le chiamate interrupt del BIOS, ...
- EBX: anche chiamato "Base", contiene puntatori per l'accesso alla memoria RAM
- ECX: anche chiamato "Contatore", è utilizzato per memorizzare dei contatori
- EDX: anche chiamato "Dati", è utilizzato per accedere ad input/output, per operazioni aritmetiche, ed alcuni interrupt del BIOS

I registri dei puntatori di un sistema x86 sono invece i seguenti:

- EDI: anche detto "Destinazione", viene utilizzato per la copia e l'impostazione degli array e delle stringhe
- ESI: anche detto "Sorgente", viene utilizzato per la copia delle stringhe e degli array
- EBP: anche detto "Base dello Stack", memorizza gli indirizzi della base dello Stack
- ESP: anche detto "Stack", memorizza gli indirizzi della parte superiore dello Stack
- EIP: anche detto "Indice", memorizza la posizione della prossima istruzione da eseguire (Nota: può essere utilizzato soltanto in lettura)

Abbiamo accennato allo "Stack", ne parleremo tra poco, per ora basta sapere che è una porzione della memoria. Particolare attenzione deve essere riservata al puntatore EIP che può essere utilizzato da un programma soltanto in lettura. Lo scopo di questo puntatore è di far sapere sempre al processore che cosa dovrà fare immediatamente dopo l'istruzione che sta eseguendo. Il meccanismo è semplice: un programma è soltanto una sequenza di istruzioni in linguaggio macchina (Assembler), che a loro volta non sono altro che un testo, ovvero una sequenza di byte che devono essere scritti nella memoria RAM del computer. Il processore deve poi poter leggere le istruzioni in questione per eseguirle, e le legge una dopo l'altra. Possiamo immaginare ogni istruzione come una variabile oppure una stringa: vale il discorso che abbiamo già

fatto per le variabili in generale, ovvero vengono memorizzate in alcune celle della memoria RAM, e possono essere lette conoscendo la posizione della prima di tali celle. Questa posizione è chiamata "puntatore". Il registro EIP contiene dunque la posizione della cella di memoria in cui si trova il primo bit della prossima istruzione che il processore dovrà eseguire. Grazie al meccanismo del byte null, anche in questo caso sarà possibile per il processore leggere interamente l'istruzione successiva ed eseguirla.

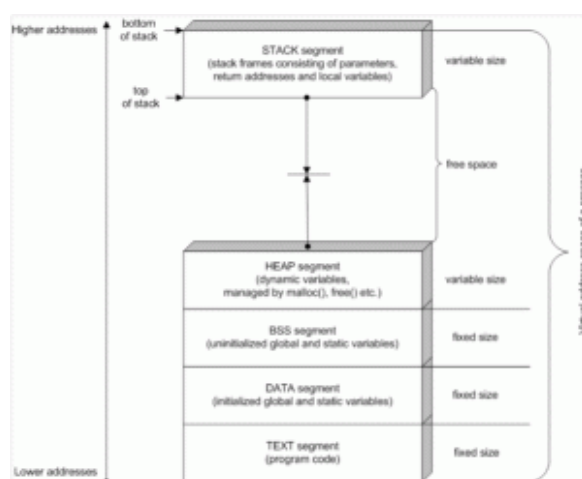


Il byte null: terminatore di stringa

Abbiamo detto che quando il processore riceve il comando di leggere una porzione della memoria, verifica l'indirizzo della prima cella da leggere e poi procede finché non gli viene detto di fermarsi. La domanda è: come si può dire al processore che l'informazione, ovvero la variabile, è terminata? Il metodo più utilizzato è il byte null (oppure NUL), che funge da terminatore di stringa. In altre parole, se il processore incontra un byte dal valore nullo termina automaticamente la lettura e considera conclusa l'informazione. In codifica esadecimale il byte nullo è `\x00`, mentre nella codifica ASCII è il semplice valore `0`, da non confondersi con il numero zero presente anche sulle tastiere dei computer (in ASCII, il numero zero è rappresentato dal valore `48`). Pensare in codice binario può essere più semplice: il byte nullo è semplicemente una sequenza di 8 bit tutti pari a zero (quindi il byte null è il seguente codice binario: `00000000`).

La segmentazione

Finora abbiamo parlato di “memoria RAM”. E probabilmente avete pensato che tale memoria sia un unico blocco, fondamentalmente un unico schedario pieno di cassette ai quali è possibile accedere in modo completamente disordinato. Non è proprio così. La memoria di un computer, per un programma, è divisa in cinque porzioni ben distinte: Text, Data, Bss, Heap, e Stack. Queste porzioni prendono il nome di “segmenti” e si parla di “segmentazione” della memoria.



La segmentazione della memoria nei segmenti Text, Data, BSS, Heap, e Stack.

Il segmento Text è quello che contiene il codice Assembly del programma in esecuzione. Naturalmente, l'esecuzione delle istruzioni del programma non è sequenziale: nonostante il codice sia scritto una riga dopo l'altra, è ovvio che il processore possa avere la necessità di saltare da una istruzione ad un'altra non immediatamente successiva o addirittura precedente. Del resto, è ciò che avviene nei cicli: se pensiamo ad un ciclo FOR del linguaggio C, al termine dell'ultima istruzione del ciclo si salta nuovamente alla prima. È per questo motivo che il puntatore EIP di cui abbiamo parlato poco fa è fondamentale: altrimenti il processore non saprebbe quale istruzione andare ad eseguire.

Il segmento Text è accessibile soltanto in lettura, a runtime. Vale a dire che mentre il programma è in esecuzione non è possibile scrivere in tale segmento. Il motivo è ovvio: il codice del programma non può cambiare durante l'esecuzione. Per tale motivo, questo segmento ha una dimensione fissa, che non può essere modificata dopo l'avvio del programma. Se qualcuno tentasse di scrivere in questo segmento di memoria, si verificherebbe un errore di segmentazione, ed il programma verrebbe immediatamente terminato (e non esiste possibilità di impedire la chiusura del programma). Quindi i pirati non possono sovrascrivere il codice sorgente del nostro programma durante l'esecuzione, ed almeno da questo punto di vista possiamo stare tranquilli. Il segmento Data viene utilizzato per memorizzare le variabili globali e le costanti che vengono inizializzate al momento della loro dichiarazione. Il segmento Bss, invece, si occupa dello stesso tipo di variabili, ma viene utilizzato nel caso in cui le variabili non siano state inizializzate.

Per capire la differenza, possiamo dire che la variabile:

Viene inserita nel segmento Data, mentre la variabile

viene inserita nel segmento Bss. In entrambe i casi, le variabili sono da considerarsi valide in tutto il programma (cioè in tutte le funzioni del programma, non sono prerogativa di una sola funzione). Queste variabili possono cambiare il loro contenuto nel corso del programma, ma non la loro dimensione (che dipende dal tipo di variabile: stringa, numero intero, numero con virgola, eccetera...). La dimensione dei segmenti Data e Bss è dunque fissa, proprio perché la dimensione delle variabili in essi contenute non può cambiare. Il segmento heap è utilizzato per tutte le altre variabili del programma. Questo segmento non ha una dimensione fissa, perché è ovvio che le variabili possono essere create e distrutte durante l'esecuzione del programma e la memoria deve essere allocata o deallocata con appositi algoritmi da ogni

linguaggio di programmazione. Per esempio, nel linguaggio C si utilizza l'algoritmo malloc per assegnare una porzione di memoria ad una variabile:

Un esempio più concreto, per costruire un array di numeri interi sarebbe il seguente:

Mentre per liberare la memoria del buffer in questione si sfrutta l'algoritmo free:

Se avete già avuto esperienze con il linguaggio C oppure C++, probabilmente non vi siete mai trovati a dover utilizzare questi due metodi. Infatti, generalmente un array viene dichiarato con la seguente sintassi:

E la memoria viene automaticamente allocata dal compilatore C. Ma il meccanismo è lo stesso: è solo un modo più rapido di scrivere lo stesso codice C, perché il codice macchina che ne risulta è quasi identico. Questo tipo di variabili ed array di variabili è dunque inserito nel segmento di memoria heap. Questo segmento, lo abbiamo detto, si espande man mano che le variabili vengono create, e la sua espansione procede verso indirizzi della memoria più alti.

Anche il segmento stack ha una dimensione variabile, e viene utilizzato per memorizzare delle variabili. Diversamente dal segmento heap, tuttavia, viene utilizzato più che altro come una sorta di "foglio di appunti". Nello stack vengono memorizzate infatti le variabili necessarie durante la chiamata delle funzioni. In qualsiasi programma, una parte fondamentale del lavoro è svolto dalle funzioni: possono essere fornite da particolari librerie oppure possiamo realizzarle noi stessi. Per esempio, in C esiste la funzione

che si occupa di copiare un array di caratteri in un altro (il

contenuto dell'array destinazione diventa uguale a quello dell'array sorgente). Ovviamente, tale libreria necessita dei puntatori ai due array, ed i puntatori sono delle variabili. Pensiamo, poi, alla funzione

che calcola il coseno dell'angolo che riceve in argomento. È ovvio che l'angolo deve essere memorizzato da qualche parte, affinché la funzione possa utilizzarlo. Il segmento di memoria utilizzato per registrare la variabile in argomento è lo stack. Ed è anche il segmento utilizzato per memorizzare il valore che deve essere restituito, ovvero il coseno dell'angolo calcolato dalla funzione che dovrà poi essere inserito nella variabile "valore".

Visto che le funzioni possono essere molto diverse ed essere richiamate un numero imprecisato di volte, è ovvio che lo stack non può avere una dimensione fissa, ma deve essere libero di aumentare o diminuire la propria dimensione a seconda delle variabili che devono essere memorizzate. È interessante che quando il segmento stack aumenta di dimensioni lo fa portandosi verso indirizzi più bassi di memoria, quindi nella direzione opposta rispetto al segmento heap.



Esadecimale e decimale

Gli indirizzi di memoria vengono solitamente scritti in base esadecimale, ma sono fondamentalmente dei numeri che possono ovviamente essere convertiti in base decimale. Siccome la base 10 è quella con cui siamo maggiormente abituati a ragionare, può essere utile tenere sottomano uno strumento di conversione delle basi. In effetti può essere poco intuitivo, se si è alle prime armi con la base 16, pensare che il numero esadecimale 210 corrisponda di fatto al decimale 528. Quando leggete un listato Assembly, può essere molto comodo convertire i numeri

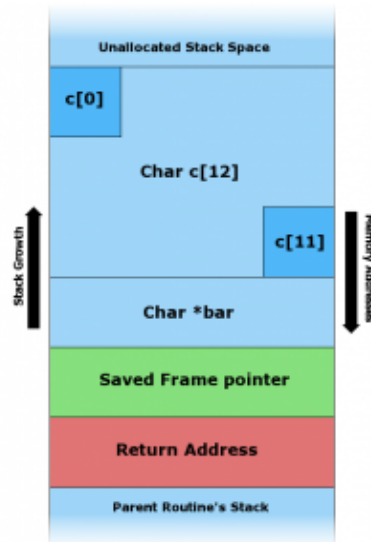
in forma decimale per comprendere la dimensione delle porzioni di memoria.

<http://www.binaryhexconverter.com/hex-to-decimal-converter>

Il contesto è importante

C'è qualcosa di importante da considerare riguardo il passaggio da una funzione ad un'altra e più in generale sul funzionamento dello stack. Ragioniamo sulla base di quanto abbiamo detto finora: il programma è una serie di istruzioni Assembly, memorizzate nel segmento della memoria Text. Queste istruzioni vengono eseguite dal processore in modo non perfettamente sequenziale: per esempio, quando viene lanciata una funzione il processore deve saltare alla cella di memoria che contiene la prima istruzione di tale funzione. Questo avviene grazie al registro EIP, che memorizza il puntatore di tale cella. Tuttavia, è anche abbastanza ovvio che appena la funzione termina, ovvero appena si raggiunge l'ultima istruzione della funzione, sia necessario ritornare al punto in cui ci si era interrotti. In pratica, il processore deve tornare ad eseguire l'istruzione immediatamente successiva a quella che aveva chiamato la funzione appena conclusa. Come fa il processore a sapere dove deve tornare? Ovviamente una tale informazione non può essere inserita direttamente nel codice della funzione, perché la stessa funzione può essere chiamata da punti diversi del codice del programma e quindi deve poter tornare automaticamente in ciascuno di questi punti. La risposta è molto semplice: sempre grazie al puntatore EIP, con un piccolo aiuto da parte dello stack. Ricapitoliamo: il codice del programma è contenuto nel segmento di memoria Text. Durante l'esecuzione del codice, il processore incontra una istruzione che richiede il lancio di una funzione. Il processore salta dunque all'indirizzo della memoria Text in cui è presente il codice di tale funzione. La funzione comincia a scrivere le proprie variabili nel segmento di

memoria Heap, ma prima di iniziare le operazioni vere e proprie vi sono delle istruzioni che indicano al processore quali sono le variabili che devono essere condivise tra la porzione del programma che ha chiamato la funzione e la funzione stessa. Queste variabili condivise vengono inserite nel segmento di memoria Stack. Assieme alle variabili condivise vi è anche un'altra informazione che va condivisa tra il codice "principale" e la funzione chiamata: l'indirizzo di ritorno. Ovvero, l'indirizzo di memoria in cui si trova l'istruzione da inserire nel registro EIP, affinché possa essere eseguita immediatamente al termine della funzione chiamata. Naturalmente, l'indirizzo di ritorno è un indirizzo che appartiene al segmento di memoria Text, perché si tratta di una istruzione del codice del programma (che abbiamo detto essere memorizzato interamente in tale segmento). Ma questo vale soltanto se il programma funziona correttamente: non c'è alcun sistema di controllo, un indirizzo di ritorno è soltanto un numero e niente più. Quindi, se viene scritto un indirizzo di ritorno errato, il programma al termine della funzione salterà in un punto della memoria che non è quello previsto originariamente dal programmatore. Questo indirizzo di ritorno è quindi un evidente punto debole del meccanismo: di solito viene scritto correttamente dal codice Assembly del programma, ma se qualcuno trovasse un modo per modificare l'indirizzo di ritorno al momento della chiamata della funzione, potrebbe di fatto dirottare l'esecuzione del programma verso una qualsiasi porzione di codice Assembly diversa da quella corretta. Ci si può chiedere: esiste un modo per modificare questo indirizzo di ritorno? Sì, ed è proprio la tecnica più comunemente utilizzata dai pirati per assumere il controllo di un programma.



In una situazione normale, lo spazio dedicato ad una variabile nello Stack contiene i byte della variabile, il byte nullo come terminatore di stringa, un puntatore del frame di memoria, e l'indirizzo di ritorno della funzione attuale.

I buffer overflow basati sullo Stack

Prima di capire come sia possibile sovrascrivere l'indirizzo di ritorno di una funzione per assumere il controllo di un programma, vediamo di capire meglio come funziona lo Stack. Il nome "Stack" è la traduzione inglese della parola "pila".

Possiamo pensare ad una pila di piatti: la formiamo aggiungendo un piatto sopra il precedente. Il principio del funzionamento è il cosiddetto FILO, First In Last Out, cioè "il primo elemento ad essere inserito è l'ultimo a poter essere estratto". Nell'analogia della pila di piatti, è abbastanza ovvio che il primo piatto che posizioniamo si trova sul fondo, e non possiamo prenderlo finché non abbiamo rimosso tutti i successivi che abbiamo posizionato sopra di esso. Per memorizzare una informazione nel segmento della memoria Stack si utilizza il comando Assembly push, mentre per leggere una informazione si sfrutta il comando pop. Naturalmente, a questo punto è necessario tenere in qualche modo traccia di quale sia l'ultima informazione registrata nello stack, cioè l'informazione che può al momento essere estratta oppure dopo la quale è possibile inserire una nuova informazione. Per memorizzare la posizione dell'ultima informazione registrata nello stack viene utilizzato il registro del processore ESP. Naturalmente è anche possibile leggere una particolare porzione dello Stack anche se essa non è l'ultima informazione registrata in esso: in fondo, basta conoscere l'indirizzo di memoria in cui è inserita l'informazione che si vuole leggere. Per memorizzare temporaneamente l'indirizzo dell'informazione che si vuole leggere si utilizza il registro EBP. Ricapitoliamo la funzione dei registri dei puntatori alla luce di quanto abbiamo detto:

- EIP: memorizza l'indirizzo di ritorno, che contiene l'istruzione da eseguire appena la funzione attuale sarà terminata
 - ESP: memorizza l'indirizzo dell'ultima informazione registrata nello Stack, così è possibile sapere dove finisce lo Stack al momento attuale e dove scrivere l'eventuale informazione successiva
 - EBP: memorizza la posizione di un indirizzo interno allo Stack (dove si trovano le variabili della funzione attuale)
- Ovviamente, i valori di ESP ed EBP vengono registrati nello Stack immediatamente prima della chiamata di una funzione, così sarà possibile recuperare i loro valori al termine della

funzione stessa (durante l'esecuzione della funzione tali registri infatti cambiano il contenuto). All'inizio di una funzione, il valore del registro EBP viene impostato dopo le variabili locali della funzione e prima degli argomenti della funzione. Per leggere le variabili locali basta sottrarre dal valore di EBP, mentre per leggere gli argomenti basta sommare. C'è un altro particolare interessante: alla fine delle variabili locali, prima degli argomenti, viene memorizzato l'indirizzo di ritorno, che come abbiamo già detto rappresenta la posizione della prossima istruzione da eseguire.

Nei sistemi x86, gli indirizzi "alti" sono quelli indicati da un numero più piccolo, mentre quelli "bassi" sono indicati da un numero più alto, e sono quelli più vicini al segmento di memoria Heap.



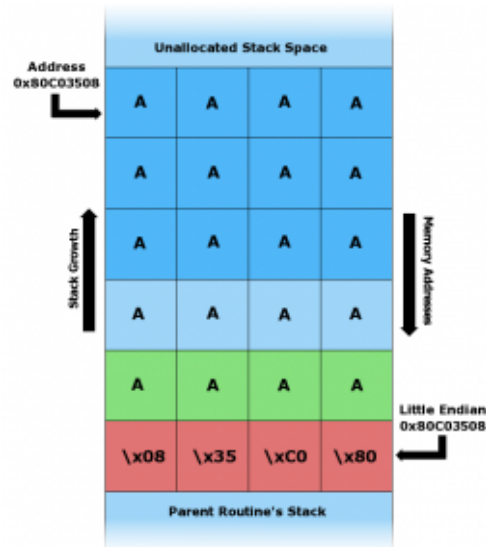
I registri a 64-bit

I registri a 64-bit sono più o meno gli stessi di un processore a 32-bit, con la differenza della prima lettera del nome, che cambia da E ad R: il registro EIP diventa RIP, mentre EBP diventa RBP e così via. L'altra differenza, abbastanza ovvia, è che ogni indirizzo a 64 bit richiede 8 byte. Inoltre, sono disponibili molti più registri, un totale di 16. I vari registri a 64 bit sono i seguenti: rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15. Questo permette di memorizzare più variabili locali nei vari registri piuttosto che nello Stack, ed ovviamente permette di ridurre in parte il problema dell'overflow nello Stack (meno variabili vengono scritte in questo segmento di memoria, meno probabile è che una di esse possa subire un overflow). Naturalmente, i programmi che necessitano di molte variabili molto grandi devono comunque utilizzare lo Stack per la loro memorizzazione, quindi sono comunque vulnerabili al tipo di attacco che presentiamo in queste pagine.

Un esempio semplice

Dopo tanta teoria, è il momento di un primo esempio pratico. Consideriamo il seguente codice:

Al momento di chiamare la funzione prova, lo Stack è così costituito: l'indirizzo più alto è riservato all'argomento 3. Sopra di esso viene registrato, con un indirizzo un po' meno alto (un numero un po' più piccolo) l'argomento 2, e successivamente l'argomento 1. A questo punto viene memorizzato l'indirizzo di ritorno. Si inserisce poi la variabile numero e l'array testo. La variabile testo è, nel nostro esempio, quella posizionata nell'indirizzo più alto dello Stack, il più vicino al segmento Heap della memoria. Fin qui tutto bene: il codice non fa nulla di particolare, il programma non svolge nessuna azione interessante, ma almeno non crea problemi. Prima di passare ad un codice che faccia davvero qualcosa è fondamentale, per il nostro discorso, notare un particolare: l'array testo dispone dello spazio di 10 caratteri. È vero che lo Stack aumenta le proprie dimensioni verso gli indirizzi più alti (cioè verso il segmento Heap) ma questo vale solo per l'operazione di allocazione. In altre parole, al momento di dover allocare lo spazio necessario all'array testo, il sistema verifica quale sia l'ultimo byte dello Stack (ovvero l'ultimo byte della variabile numero). Da essa vengono contati 10 byte verso lo Heap, e questo è lo spazio riservato alla variabile testo. Tuttavia, se si deciderà di scrivere il contenuto della variabile testo (nel nostro esempio ciò non avviene) la scrittura inizierà dal byte più vicino allo Heap, andando poi in direzione dell'ultimo byte dedicato alla variabile numero.



Durante la situazione di overflow, l'intero spazio dedicato alla variabile nello Stack è riempito dai byte della variabile stessa (nell'esempio il byte `\x41`, ovvero "A"). Il valore `\x41` va a sovrascrivere anche il byte nullo, il puntatore del frame di memoria, e l'indirizzo di ritorno della funzione.

I lettori più attenti si saranno già chiesti: che cosa succede se, per errore, viene inserito nell'array testo una quantità di byte maggiore 10? Per esempio che succede se vengono scritti 11 byte? Ciò che accade è che i primi 10 byte vengono scritti esattamente come è previsto, ma viene scritto anche l'undicesimo byte. E questo undicesimo byte va a sovrascrivere ciò che incontra, ovvero l'ultimo byte dedicato alla variabile numero. Consideriamo ora questo codice:

È evidente che questo codice fa qualcosa, anche se è molto

semplice. La funzione principale inizializza una variabile chiamata `dimensione`. Questa variabile registra il numero di caratteri che dovranno essere contenuti nell'array stringa. Con un ciclo `for` si riempie tale array con lettere "A". Infine, si chiama la funzione `prova`. Tale funzione prende in argomento l'array stringa, dichiara un nuovo array con dimensione fissa (pari a 10 caratteri) e copia in esso il contenuto dell'array ricevuto in argomento. La copia viene eseguita con l'apposita funzione `strcpy` della libreria standard "string.h". Se provate a compilare ed eseguire questo codice, vedrete che funziona. E questo perché la dimensione dei due array copiati è identica. Il codice funzionerebbe bene anche se l'array da copiare (cioè l'array stringa) fosse più piccolo dell'array di destinazione (cioè `testo`). Si può verificare semplicemente modificando il valore della variabile `dimensione`, per esempio nel seguente modo:

Se invece proviamo a rendere l'array di origine più grande di quello di destinazione, il programma viene terminato. Infatti, modificando la riga di codice con la seguente:

Otteniamo un "errore di segmentazione", anche chiamato "buffer overflow". Che cosa è successo? È successo che la funzione `prova` ha ricevuto in argomento un array contenente ben 11 caratteri "A", ed ha provato ad inserirle in un array che disponeva di spazio allocato per un massimo di 10 caratteri. Di conseguenza, l'undicesima "A" è andata a sovrascrivere l'informazione immediatamente precedente nello Stack. E questa informazione sovrascritta era, dovrete averlo capito, l'indirizzo di ritorno. In realtà, trattandosi di un solo carattere in più, ad essere stato sovrascritto è un valore chiamato `SFP`, che precede sempre l'indirizzo di ritorno. Se i caratteri fossero stati almeno 12, l'indirizzo di ritorno sarebbe stato sicuramente sovrascritto. Non abbiamo parlato del valore `SFP` perché non è particolarmente rilevante per i

nostri scopi, e possiamo considerarlo come un'altra variabile locale della funzione prova. Riassumendo: con una dimensione dell'array stringa maggiore di 10 si ottiene una sovrascrittura dell'indirizzo di ritorno. Dunque, appena la funzione termina il processore legge la cella che secondo le sue informazioni contiene l'indirizzo in cui si trova la prossima istruzione da eseguire. Purtroppo, in quella cella di memoria l'indirizzo di ritorno vero non è più presente, ed è inserito invece un valore errato: nel nostro esempio la lettera "A" che corrisponde al numero esadecimale \x41. Il processore è convinto che il numero \x41 rappresenti l'indirizzo di ritorno corretto, quindi lo inserisce nel registro EIP e si prepara a leggere l'istruzione memorizzata nella cella di memoria identificata da questo indirizzo. Naturalmente, è molto probabile che la cella di memoria presente all'indirizzo \x41 non contenga alcuna istruzione valida, quindi il processore si trova nell'impossibilità di procedere nell'elaborazione, e termina "brutalmente" (con un crash) il programma dichiarando per l'appunto un "errore di segmentazione", ovvero un errore nella gestione dei segmenti di memoria del programma. In questo caso, e del resto nella netta maggioranza dei crash dei programmi, si tratta di un errore del segmento Stack (esistono anche situazioni simili che si verificano nel segmento Heap, ma sono più rare).



Indirizzi a 32 bit

Un particolare: nell'esempio che realizzeremo d'ora in poi ci basiamo su un sistema a 32 bit. Di conseguenza, l'indirizzamento della memoria è basato su 4 byte (1 byte equivale ad 8 bit, per avere 32 bit servono 4 byte). Quindi, un indirizzo di memoria (come l'indirizzo di ritorno) deve essere scritto con 4 byte. Per esempio, un indirizzo di memoria in un sistema x86 potrebbe essere l'esadecimale

0x41414141, scritto anche come \x41\x41\x41\x41, che corrisponde alla stringa AAAA.

Facile... o quasi

Adesso che abbiamo capito come va in crash un programma per buffer overflow, ci si può chiedere: come fa un pirata a sfruttare questo tipo di errori per far eseguire al processore del codice a sua discrezione? La risposta dovrebbe già esservi balenata in mente sotto forma di un'altra domanda: nel nostro esempio l'indirizzo di ritorno veniva sovrascritto con un valore non valido, ma che cosa succederebbe se l'indirizzo di ritorno venisse sovrascritto con un valore che punta a delle istruzioni in codice Assembly effettivamente eseguibili da parte del processore? La risposta è drammaticamente semplice: il processore le eseguirebbe senza alcun problema. Ciò significa, di fatto, che è possibile dirottare l'esecuzione di un programma semplicemente sovrascrivendo l'indirizzo di ritorno in modo che punti ad una porzione della memoria nella quale è stato precedentemente inserito del codice macchina Assembly funzionante.

Insomma, la vita di un pirata sembra piuttosto semplice. In realtà, ci sono alcuni particolari che rendono le cose un po' più complicate. Riassumiamo ciò che un pirata deve fare:

0) trovare un programma con una funzione in cui ad una variabile viene assegnato un valore senza prima controllare che tale valore sia più piccolo dello spazio massimo allocato alla variabile stessa

1) capire dove si trova l'indirizzo di ritorno della funzione

2) scrivere del codice macchina nella memoria del computer

3) sovrascrivere l'indirizzo di ritorno inserendo al suo posto l'indirizzo in cui si trova il codice macchina appena scritto

I problemi sono dunque due: uno consiste nell'ottenere le informazioni necessarie (la posizione dell'indirizzo di ritorno e la posizione del proprio codice macchina), l'altro

nello scrivere tutto il necessario (sia il proprio codice macchina che il nuovo indirizzo di ritorno). Esiste un modo molto semplice per risolvere il problema della scrittura: si può fare tutto con la scrittura della variabile. Abbiamo detto che la vulnerabilità del programma deriva dal fatto che permette l'assegnazione di qualsiasi valore ad una certa variabile, anche se più grande del previsto. Quindi il pirata può decidere di assegnare alla variabile in questione un valore che di fatto corrisponde al codice macchina che vuole eseguire, sufficientemente lungo da sovrascrivere l'indirizzo di ritorno. La vulnerabilità può quindi essere sfruttata con una sola operazione: l'assegnazione di un valore, appositamente preparato, alla variabile. Un esempio pratico ci aiuterà a capire quanto semplice sia la questione, realizzando il file errore.c:

Se siete stati attenti, avrete capito che in questo esempio la variabile "vulnerabile" è stringa. Infatti, tale variabile viene inizializzata con una dimensione di 500 caratteri. Tuttavia, le viene poi assegnato (grazie alla funzione strcpy) il valore di argv[1], che rappresenta l'argomento con cui viene lanciato il programma, il quale è a discrezione dell'utente. Per capirci, possiamo compilare il programma utilizzando il compilatore GCC, che su un sistema GNU/Linux (oppure su Windows con l'ambiente Cygwin) si lancia nel seguente modo:

a cui deve seguire il comando



La sicurezza di Linux

Eseguendo il tentativo di cracking su un sistema GNU/Linux, probabilmente non funzionerà. Questo perché il kernel Linux ha

dei meccanismi di protezione, non presenti in Windows, che di fatto impediscono l'esecuzione di shellcode tramite errori di segmentazione. Affinché il nostro tentativo vada a buon fine, rimuoviamo la protezione dello stack da parte del Kernel Linux:

Rendendo dunque eseguibile il codice presente nel segmento di memoria Stack (nelle recenti versioni di Linux è eseguibile soltanto il segmento Text per ovvii motivi di sicurezza, ma altri sistemi operativi non offrono questo tipo di protezione). Potrebbe anche essere necessario disabilitare la randomizzazione dello Stack (ASLR):

È una particolare forma di protezione del kernel Linux (introdotta anche nelle versioni di Windows successive al 2007, ma solo per alcuni programmi): si occupa di rendere casuali e non consecutivi gli indirizzi della memoria della Stack, in modo da rendere molto difficile la stima dell'indirizzo in cui viene memorizzata la variabile "vulnerabile" (nel nostro esempio la variabile stringa).

<http://linux.die.net/man/8/execstack>

https://docs.oracle.com/cd/E37670_01/E36387/html/ol_aslr_sec.html

A questo punto possiamo avviare il programma fornendogli un argomento, per esempio:

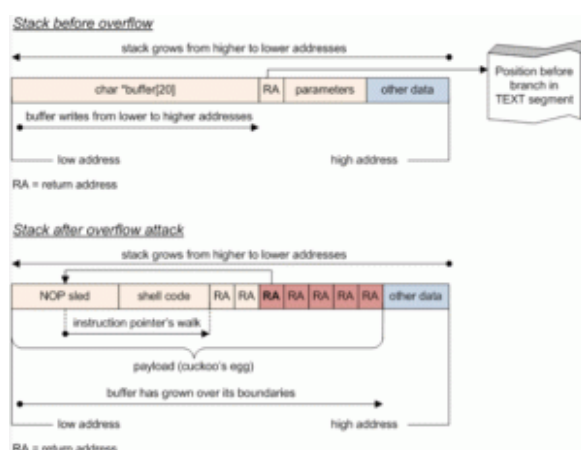
Il programma termina senza alcun problema, perché la parola "gatto", che è l'argomento del programma, ha meno di 500 caratteri. Ma se proviamo ad avviare il programma col seguente comando:

Il programma andrà in crash, con un "segmentation fault" (che significa "errore di segmentazione". Infatti, l'argomento che abbiamo appena scritto contiene ben 529 caratteri: 29 in più della dimensione massima accettabile dalla variabile stringa.

Siccome non esiste alcun controllo, l'argomento viene scritto dentro la variabile ed il suo contenuto straripa, per cui gli ultimi byte dell'argomento finiscono per sovrascrivere l'indirizzo di ritorno della funzione main e provocare il crash.

In realtà esiste anche un metodo più semplice per realizzare un stringa molto lunga nel terminale di GNU/Linux: utilizzare l'interprete del linguaggio Perl. Se, per esempio, scriviamo il comando:

Otterremmo lo stesso risultato del comando precedente, perché al programma errore è appena stato passato un argomento con ben 600 caratteri: il comando Perl che abbiamo indicato, infatti produce una sequenza di ben 600 caratteri "A" (infatti il valore esadecimale corrispondente al carattere A è \x41).



Ecco cosa avviene, nello Stack, dopo avere fornito al programma vulnerabile la nostra stringa malevola: prima c'è la NOP sled, poi lo shellcode, ed infine la ripetizione dell'indirizzo di ritorno che punta alla NOP sled.

La slitta NOP

Per quanto riguarda l'altro problema, ovvero la necessità di conoscere gli indirizzi di memoria da sovrascrivere e quelli in cui si scrive, non esiste modo per il pirata di ottenere le informazioni di cui ha bisogno, dal momento che in ogni computer gli indirizzi di memoria saranno diversi. Tuttavia, esiste un trucco grazie al quale queste informazioni risultano non più necessarie: si chiama NOP sled. La traduzione letterale è "slitta con nessuna operazione", ed è una istruzione in linguaggio macchina che, semplicemente, non fa niente (NOP significa "nessuna operazione"). È molto importante capire che una istruzione NOP fa in modo che il processore passi immediatamente all'istruzione successiva. Si può quindi facilmente costruire una "slitta": una lunga sequenza di istruzioni NOP non fa altro che portare il processore all'istruzione posizionata dopo l'ultimo NOP. Facciamo un esempio pratico: innanzitutto, ricordiamo che in un sistema x86 l'istruzione NOP è rappresentata dal numero esadecimale `\x90`.

L'istruzione:

consiste banalmente nell'istruzione:

Perché tutti i `\x90` vengono saltati dal processore appena li legge: banalmente, appena il processore incontra una di queste istruzioni il registro EIP viene incrementato di una unità, quindi il processore passa a leggere il byte immediatamente successivo. Non è inutile come può sembrare: può essere utilizzato per sincronizzare delle porzioni di memoria. Il lato più interessante della cosa è che, ovviamente, le istruzioni:

e

Sono perfettamente equivalenti, perché non ha alcuna importanza quanti `\x90` ci sono. Ecco dunque il trucco del pirata per evitare di dover capire dove si trova esattamente l'indirizzo di memoria: basta scrivere una slitta NOP (cioè una serie di `\x90`) abbastanza lunga immediatamente prima dell'istruzione da eseguire. In questo modo non serve conoscere esattamente in quale indirizzo di memoria è stata registrata l'istruzione da eseguire: basta avere una idea di massima di dove potrebbe trovarsi uno qualsiasi dei byte `\x90`, e la slitta NOP farà sì che il processore finisca con l'eseguire proprio l'istruzione che il pirata desidera. Naturalmente, si deve ancora risolvere il problema di sapere esattamente dove deve essere posizionato l'indirizzo di ritorno della funzione. Anche questo problema può essere risolto con una certa facilità: basta ripetere molte volte l'indirizzo desiderato (che va calcolato in modo che si riferisca ad almeno uno dei numerosi byte `\x90` scritti precedentemente). Infatti, per la legge probabilistica dei "grandi numeri", basta ripetere molte volte l'indirizzo di ritorno affinché almeno una di queste volte esso venga scritto proprio nel punto in cui deve trovarsi.



La dimensione della NOP sled

Nell'esempio abbiamo scelto di utilizzare una lunghezza di 200 byte per la slitta NOP. Naturalmente, avremmo potuto scegliere anche una dimensione di 204 byte per la nostra NOP sled, perché la somma ($204+28=232$) è comunque divisibile per 4. Il vantaggio di 232 byte rispetto a 228 è che il numero 232 è divisibile anche per 8, quindi può funzionare anche su un sistema a 64 bit (infatti per realizzare indirizzi a 64 bit servono 8 byte).

Ricapitolando, è possibile sfruttare la vulnerabilità di un programma come il nostro `errore.c` semplicemente inviandogli

una stringa costruita con una lunga sequenza di istruzioni NOP (\x90 in esadecimale), poi un codice Assembly da eseguire per ottenere il controllo del computer, ed infine l'indirizzo di ritorno, che punta proprio su una delle istruzioni NOP, ripetuto molte volte. La stringa sarà molto lunga, ma questo non è un problema. Anzi: in fondo, la vulnerabilità del programma dipende proprio dall'eccessiva lunghezza della stringhe che riceve.

Costruire la stringa

Proviamo, adesso, a costruire una stringa con queste caratteristiche, per sfruttare la vulnerabilità del programma errore.c che abbiamo realizzato poco fa. Utilizzeremo Perl per realizzare la NOP sled. Infatti, il comando:

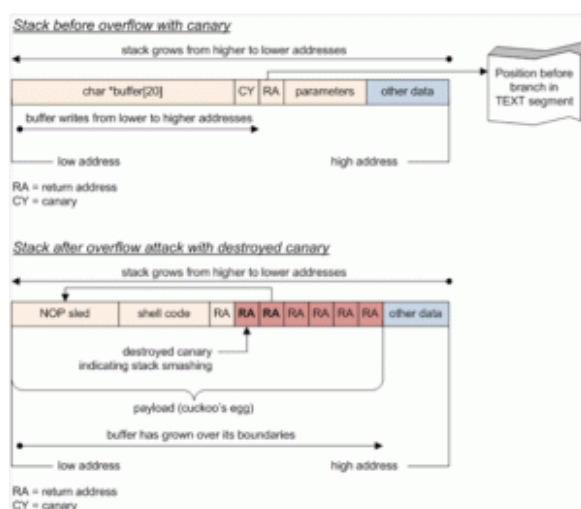
Produce una sequenza di 600 istruzioni NOP (l'esadecimale \x90), ovvero una NOP sled di 600 byte e la passa al programma errore. Naturalmente, questo non basta per sfruttare davvero la vulnerabilità del programma: ci servono anche un codice macchina Assembly da eseguire e l'indirizzo di ritorno. Il codice Assembly che un pirata vuole eseguire può essere qualcosa di simile al seguente:

Per il momento non entriamo troppo nei dettagli: ci accontentiamo di dire che questo tipo di codice è chiamato "shellcode", perché permette al pirata di ottenere una shell, ovvero un prompt dei comandi con cui avere il controllo del computer su cui era in esecuzione il programma vulnerabile. I codice shellcode sono di pubblico dominio, ed esistono siti web che li raccolgono: noi ci siamo basati sul seguente <http://shell-storm.org/shellcode/files/shellcode-811.php>.

Procediamo, dunque, a modificare il comando affinché contenga sia la NOP sled che lo shellcode:

Vi starete chiedendo: perché abbiamo realizzato una NOP sled di esattamente 200 byte? In realtà non c'è un motivo preciso per scegliere proprio questo numero, ma esiste una regola da rispettare: visto che l'indirizzamento della memoria nei sistemi a 32 bit richiede 4 byte, è ovvio che la somma dei byte della NOP sled e dello shellcode deve obbligatoriamente essere divisibile per 4, altrimenti l'indirizzo di ritorno (che scriveremo tra poco, finirebbe per essere disallineato (cioè non comincerebbe nell'esatta posizione in cui il processore si aspetterebbe di trovarlo). Se avete contato i byte dello shellcode, avrete notato che sono 28. Una NOP abbastanza grande deve avere almeno 100-200 byte. Potremmo scegliere un numero qualsiasi, per esempio 190. Tuttavia, la somma di 190+28, ovvero 218 byte, non è divisibile per 4. Un numero che possa essere divisibile per 4 è 228 quindi, visto che la dimensione dello shellcode non può cambiare, impostiamo una dimensione della NOP sled tale da ottenere una somma totale di 228 byte: la NOP sled deve avere una dimensione di 200 byte.

Ci manca, ormai, soltanto la parte dell'indirizzo di ritorno.



Il compilatore GCC inserisce nello Stack, prima dell'indirizzo di ritorno, un byte "canary" (canarino). Se l'indirizzo di ritorno viene sovrascritto, anche il

canary è sovrascritto.
Appena il programma si
accorge che il canary non ha
più il valore originale, si
interrompe impedendo
l'esecuzione dello
shellcode.

Trovare l'indirizzo giusto

L'indirizzo di ritorno che noi vogliamo scrivere è ovviamente un indirizzo che corrisponde ad almeno uno dei caratteri della NOP sled. Come facciamo a sapere dove si trova questo codice macchina? Semplice: la NOP sled è ora inserita nella memoria del computer tramite la variabile "vulnerabile", ovvero quella che nel nostro programma errore.c avevamo chiamato stringa, ed alla quale avevamo assegnato un massimo di 500 byte. Basterà trovare la posizione in memoria di tale stringa durante una esecuzione del programma errore e sapremo dove trovare la nostra NOP sled.

Iniziamo compilando il programma errore.c assicurandoci che il compilatore non aggiunga del codice per evitare la sovrascrittura dell'indirizzo di ritorno:



La protezione di GCC

L'opzione `-fno-stack-protector` serve ad evitare che il compilatore GCC inserisca del codice per evitare la sovrascrittura degli indirizzi di ritorno delle funzioni. Tale funzionalità è presente soltanto in GCC, per ora: è una buona forma di protezione, ma sono ancora pochi i programmatori che

ne fanno uso, quindi noi la disabilitiamo di proposito proprio per vedere cosa succede ai tutti i programmi che non dispongono di questo meccanismo di difesa. Potete provare ad eseguire nuovamente la procedura con il programma compilato senza l'opzione `-fno-stack-protector` per vedere che cosa succede se gli indirizzi di ritorno delle funzioni vengono protetti: al momento della sovrascrittura, il programma verrà terminato. Questo ci da una indicazione importante: dovendo scegliere un compilatore per i nostri programmi C, il compilatore GCC offre già una buona protezione automatica dai buffer overflow.

Ora procederemo proprio come un pirata informatico: utilizzando GNU Debugger. Avviamo il programma con il comando:

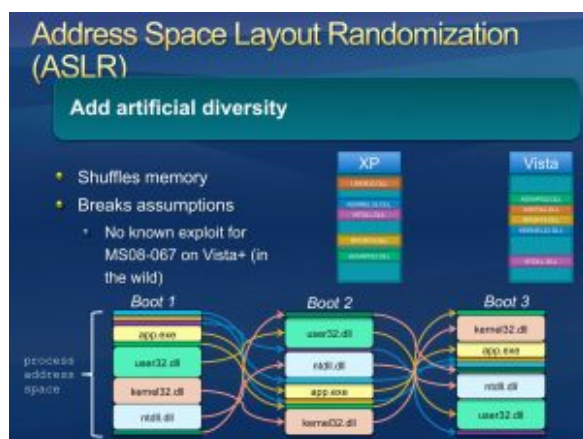
Otterremo il terminale di GDB. Controlliamo il codice Assembly del programma errore, in particolare quello della funzione `main` (che è il cuore di ogni programma):

Vedremo qualcosa del genere:

Naturalmente, noi conosciamo già il codice sorgente del programma. Ma facciamo finta di non averlo letto, esattamente come accade in genere per un pirata che vuole cracckare un nostro programma e non può leggere il codice sorgente, accontentandosi invece del codice Assembly. Innanzitutto, possiamo vedere che qui c'è una chiamata alla funzione `strcpy`, all'istruzione 31. Evidentemente, viene dichiarata una variabile, perché immediatamente prima di questa istruzione abbiamo l'istruzione `mov`, che sposta delle informazioni nel registro `eax` (che contiene le variabili di funzione). Qual è la dimensione della variabile? Semplice: dobbiamo vedere come è cambiato il puntatore `ESP`. Questa operazione viene fatta all'istruzione `main+6`:

Al registro vengono sottratti `0x210` byte, ovvero 528 byte,

riservandoli alla variabile che verrà poi passata alla funzione strcpy. Significa che la dimensione effettiva della variabile è sicuramente inferiore a 528, perché nello spazio riservato devono essere presenti i vari byte della variabile (che noi sappiamo essere 500 perché abbiamo indicato tale dimensione nel codice sorgente), un byte che funge da terminatore di stringa (cioè un carattere null), e poi alcuni byte per ottenere un corretto allineamento dello stack. L'allineamento dei byte è necessario per garantire la corretta lettura delle word (cioè gruppi di 4 byte in un sistema a 32 bit, oppure 2 byte in un sistema a 16 bit: https://en.wikipedia.org/wiki/Data_structure_alignment).



La protezione ASLR è presente su Windows da Vista in poi, ma ha dei difetti fino a Windows 8: <http://recx ltd.blogspot.co.uk/2012/03/partial-technique-against-aslr-multiple.html>

L'istruzione di ritorno della funzione è identificata come main+42. Per capire come si comporta il programma, dovremo naturalmente interrompere la sua esecuzione prima di tale istruzione: ci serve un "breakpoint" presso l'istruzione immediatamente precedente, ovvero l'istruzione leave che è identificata come main+41.

Fissiamo quindi il breakpoint con il comando:

Adesso, GDB metterà in pausa il programma appena arriva a tale istruzione, quindi un attimo prima di chiamare l'indirizzo di ritorno. Questa pausa ci darà la possibilità di vedere se l'indirizzo di ritorno viene sovrascritto e come. Ordiniamo l'esecuzione del programma con il comando:

il programma si è fermato al breakpoint. Controlliamo il contenuto attuale dei registri del processore con il comando:

che è una abbreviazione di `info registers`. Il risultato sarà il seguente:

Tutto normale, per ora. Procediamo adesso a eseguire soltanto la prossima istruzione Assembly: una sola istruzione, senza arrivare davvero al termine del programma.

Si può fare con il comando

GDB ci avviserà che qualcosa è andato storto:

Se diamo nuovamente il comando

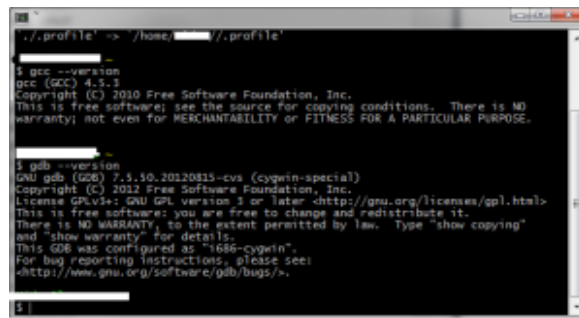
otterremo questo risultato:

Si può notare che i registri del processore sono stati sovrascritti. In particolare, sia il registro `ebp` che `eip` contengono il codice `\x41\x41\x41\x41`, che è parte della stringa che avevamo fornito al programma tramite il comando `Perl`. `EIP` è molto importante perché è il registro che contiene l'indirizzo della prossima istruzione da eseguire.

Ora possiamo cercare di capire dove, esattamente, venga memorizzata la variabile vulnerabile. Diamo dunque il comando

e otterremo gli ultimi 600 byte memorizzati nello Stack (cioè i 600 byte che precedono l'ultimo byte dello Stack,

identificato dal registro del processore ESP).



Sia GCC che GDB sono disponibili nell'ambiente Cygwin, che simula un terminale GNU/Linux in Windows

Dovremmo avere qualcosa del genere:

La variabile vulnerabile è registrata in quella porzione di memoria che ha valore 0x41414141 (perché è questo il valore che abbiamo scritto con il comando Perl). Quindi uno qualsiasi degli indirizzi che hanno tale valore andrà bene. È una buona idea scegliere uno degli indirizzi centrali, per esempio 0xffffd5a0.

Chiudiamo GDB e poi riapriamolo, in modo da ricominciare da capo, dando i comandi:

Abbiamo anche impostato nuovamente il breakpoint. È arrivato il momento di realizzare la stringa completa: ci eravamo fermati alla seguente:

Ora siamo finalmente pronti per aggiungere l'indirizzo di ritorno che vogliamo, ovvero 0xffffd5a0. In esadecimale viene scritto \xa0\xd5\xff\xff, con i byte scritti in senso inverso perché i processori x86 utilizzano la convenzione little endian, che prevede la scrittura in senso inverso degli indirizzi di memoria. Dobbiamo soltanto decidere quante volte

ripetere l'indirizzo di ritorno, per essere certi che almeno una volta vada a sovrascrivere quello originale.

Il calcolo è facile: la stringa attuale ha una lunghezza di $202+28=230$ byte. La variabile da riempire ne contiene 500, e noi vogliamo quindi che la nostra stringa abbia una lunghezza minima di 600 byte (è meglio abbondare). Servono quindi un minimo di 370 byte: l'indirizzo di ritorno ne ha 4, quindi se ripetiamo tale indirizzo per 93 volte avremo 372 byte. Siccome è meglio sbagliare per eccesso che per difetto, possiamo semplicemente ripetere l'indirizzo di ritorno per 100 volte, così da avere 400 byte che sommati ai precedenti portano la nostra stringa ad una lunghezza totale di 630 byte. Questo ci garantisce un buffer overflow.



La dimensione della variabile vulnerabile

Naturalmente visto che la posizione corretta dell'indirizzo di ritorno, ovvero la posizione in cui il processore si aspetta di trovarlo, dipende dalla dimensione della variabile, nel nostro caso il trucco funziona bene perché la variabile ha una dimensione di 500 byte, più che sufficienti per contenere lo shellcode ed almeno una piccola NOP sled. Se, tuttavia, la variabile avesse avuto soltanto 20 byte come dimensione massima, non avremmo potuto sfruttare questo metodo dal momento che lo shellcode che abbiamo usato occupa 28 byte, e di conseguenza sarebbe andato a sovrascrivere anche le celle di memoria dell'indirizzo di ritorno originale, le quali non avrebbero dunque potuto essere sovrascritte dall'indirizzo di ritorno falso. Una soluzione consiste nel realizzare un programma malevolo, in C, che costruisca una variabile contenente l'intero codice necessario (NOP sled e shellcode). Essendone il costruttore il programma malevolo può conoscere l'esatto indirizzo di memoria di tale variabile. Poi, lo stesso programma può avviare il programma vulnerabile (nel

nostro caso il programma errore), fornendogli come valore per la stringa vulnerabile una lunga sequenza costruita semplicemente ripetendo molte volte l'indirizzo di memoria in cui si trova la variabile del programma malevolo.

Ricapitolando, la stringa completa per sfruttare la vulnerabilità del programma errore è la seguente:

Per una maggiore leggibilità, la presentiamo con alcuni spazi in modo che possa andare a capo e essere letta agevolmente:

Ricordiamo che, affinché funzioni, la stringa deve essere su una sola riga e senza spazi. Possiamo provare la stringa in GDB dando il comando:

Grazie al breakpoint che abbiamo inserito, possiamo controllare lo svolgimento dando i comandi

e poi

Se tutto è andato bene, dovremmo notare che l'indirizzo di ritorno della funzione main è stato sostituito con 0xffffd5a0:

A questo punto possiamo anche analizzare il contenuto dell'area di memoria che inizia presso l'indirizzo 0xffffd5a0 sfruttando il seguente comando per GDB:

Otterremo il listato dei 250 byte successivi all'indirizzo che abbiamo indicato, interpretati come codice eseguibile Assembly:

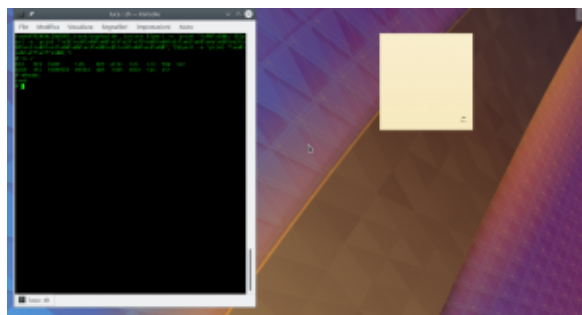
Possiamo infatti vedere una lunga sfilza di istruzioni NOP, seguite da un breve programma che è fondamentalmente lo shellcode.

Non rimane altro da fare che verificare l'effettivo funzionamento della stringa ordinando a GDB di proseguire con l'esecuzione del codice (che era in pausa grazie al nostro breakpoint). Basta dare il comando

E otterremo il seguente risultato:

Una shell perfettamente funzionante, tramite la quale dare comandi al sistema operativo. Naturalmente, adesso che abbiamo visto che la nostra stringa malevola funziona in GDB, possiamo chiudere il debugger (comando `exit` per chiudere la shell e poi `quit` per chiudere GDB) e provare l'effettivo funzionamento direttamente nel terminale:

Anche in questo caso, dovremmo ottenere una shell.



Fornendo al programma la stringa malevola, l'esecuzione viene dirottata e si apre un terminale

Anche in questo caso, dovremmo ottenere una shell.

Riassumendo, il codice sorgente incriminato, che possiamo salvare nel file `errore.c`, è questo:

Possiamo poi verificare la vulnerabilità compilandolo senza protezione dello stack, e disabilitando le protezioni di Linux prima di lanciare il programma con lo shellcode:

Volendo testare la vulnerabilità in GDB, possiamo lanciare il debug con questo comando:

e dare dal terminale di GDB i comandi per l'impostazione del breakpoint e l'esecuzione passo passo, utile per controllare i registri.

Lasciando proseguire l'esecuzione del programma, otterremo l'avvio di un terminale /bin/dash. Questo ci insegna quanto possa rivelarsi pericoloso un piccolo errore nella gestione di una variabile: se il programma non fosse così semplice, ma fosse per esempio un programma server accessibile tramite interfaccia web, e la variabile "incriminata", fosse impostabile dall'utente con un form HTML, un pirata potrebbe eseguire un qualsiasi comando sul server. Magari, persino una shell remota per prenderne il controllo.

Come funzionano i Bitcoin (e il Dark Web)

Uno dei punti di svolta fondamentali nel corso dell'evoluzione del genere umano è rappresentato dall'invenzione del denaro. Molti animali, infatti, conducono una vita da individui, senza mai costruire una società (pensiamo, per esempio, ai gatti). Altri animali, ma in numero inferiore, hanno imparato a vivere in società più o meno grandi: molte scimmie vivono in vere e proprie famiglie, più o meno organizzate. Ed è naturale che, quando c'è una organizzazione sociale in un gruppo, ogni individuo abbia un compito che serve alla comunità. L'individuo cede qualcosa al gruppo, ed il gruppo ridistribuisce i beni a tutti gli individui. Nasce quindi

l'esigenza del baratto: una scimmia raccoglie delle banane, ed un'altra raccoglie dei mango. La prima consegnerà un paio di banane alla seconda in cambio di un frutto di mango, e viceversa. Il baratto, dunque, è una componente fondamentale della società, perché grazie ad esso non è necessario che tutti sappiano fare tutto e ci si può specializzare in un compito particolare sapendo che qualcun altro si occuperà del resto. Gli animali che hanno implementato un modello di società tendono a seguire il meccanismo del baratto. Ma solo un animale è, finora, riuscito a superare il baratto con un trucco: fissare in modo univoco il valore di un certo oggetto. È il concetto di denaro, e l'animale in questione è ovviamente l'uomo. L'acquisto di beni o servizi tramite denaro è, fondamentalmente, una forma di baratto: il cliente prende dal venditore un casco di banane in cambio di un pezzo di metallo che, per convenzione, vale 2 euro. Ma l'uso del denaro ha due vantaggi importantissimi rispetto al baratto: il primo vantaggio è che in questo modo è più facile tenere sotto controllo il valore delle cose. Infatti, un venditore può barattare un casco di banane con un ananas, mentre un altro venditore baratta lo stesso casco di banane con una capra. Ma difficilmente si potrebbe vendere quel casco di banane per 2 euro in un negozio e 20 euro nell'altro. L'altro grande vantaggio è la possibilità di accumulare il denaro per tempi migliori: non si possono accumulare banane per aspettare di barattarle con qualcos'altro durante l'inverno o addirittura negli anni a venire, perché marcirebbero nel frattempo. Invece, nel caso del denaro, è possibile metterlo da parte per spenderlo in momenti di magra. È il concetto delle pensioni: durante la vita lavorativa si mettono da parte dei soldi, che vengono poi utilizzati per mantenersi quando non si è più in grado di lavorare. E se già così non ci piace molto l'idea di mettere da parte denaro per almeno vent'anni, mettere da parte banane e ananas per due decenni avrebbe risultati decisamente peggiori.

I “bug” del denaro

L'introduzione che avete appena letto è, probabilmente, inutile: tutti noi utilizziamo denaro ogni giorno. Ma, proprio perché ne siamo tanto abituati, lo diamo per scontato non pensiamo troppo a come funzioni davvero il meccanismo dell'acquisto, in contrapposizione a quello del baratto.

Ora la domanda è: quali sono le vulnerabilità del denaro? Sicuramente, la contraffazione: fin da quando è apparso per la prima volta, il denaro è stato soggetto a falsificazioni. I progettisti delle varie monete hanno sempre cercato, nel corso dei secoli, metodi per rendere la vita più difficile possibile ai falsari. Nel corso degli ultimi secoli si sono utilizzate monete realizzate con punzoni particolari, difficili da replicare fedelmente, mentre oggi siamo abituati a vedere sulle banconote filigrane molto particolari. Ma i falsari prima o poi riescono sempre a trovare un metodo per duplicare il denaro.

L'avvento della contabilità digitale ha in buona parte risolto questo problema, ma ne ha introdotto un altro: la tracciabilità, che può generare nella violazione della privacy.

C'è una precisazione da fare: abbiamo parlato di “contabilità digitale” e non di “denaro digitale”. Infatti, quando facciamo un acquisto online tramite Visa o Mastercard, il denaro è sempre reale: la contabilità delle transazioni è digitale. In pratica, non cambia il meccanismo di pagamento: alla fin fine se acquistiamo una penna su ebay, il denaro che paghiamo arriverà in mano al venditore. Possiamo schematizzare in questo modo: noi versiamo dei contanti alla nostra banca, la somma entra nel nostro conto corrente online, paghiamo la cifra dovuta al commerciante, tale cifra viene spostata sul conto del venditore, e questo va alla sede della sua banca per prelevare, tramite bancomat, il denaro. Il denaro rimane sempre sotto forma di monete o banconote, anche se le banconote che il venditore si ritroverà in mano non saranno fisicamente le

stesse che noi avevamo versato in banca. Il denaro in circolo è sempre rappresentato da banconote fisiche.

Quello che cambia è il modo in cui le transazioni vengono registrate: con la contabilità “classica” è facile perdere le tracce dopo un paio di intermediari. Per esempio, se acquistiamo un ombrello pagando in banconote innanzitutto non è possibile sapere che siamo stati davvero noi ad eseguire l’acquisto e non, piuttosto, qualcun altro. In secondo luogo, il commerciante avrà segnato la cifra che gli abbiamo corrisposto tra le sue entrate, ma a sua volta è difficile capire (anche se non impossibile) quale parte del denaro che gli abbiamo corrisposto rappresenta davvero il valore dell’ombrello e quale parte, invece, è il guadagno personale del commerciante che nulla ha a che vedere con il valore dell’oggetto in se.

Riassumendo, il problema sta nel fatto che gli acquisti sono facilmente tracciabili, e questo può rivelarsi un problema soprattutto in paesi privi delle libertà fondamentali. Un dissidente di un regime autoritario ha sicuramente il conto corrente sotto controllo.

Inoltre, il sistema è centralizzato, e i gestori possono commettere violazioni sui correntisti: l’esempio più noto è quello di Julian Assange, che dopo avere messo in imbarazzo il governo degli Stati Uniti si è ritrovato con i conti correnti congelati, impossibilitato ad eseguire qualsiasi pagamento, senza che vi fosse un provvedimento del tribunale ma per spontanea decisione degli istituti bancari.

La soluzione

Dunque, il denaro “analogico” ha due punti deboli: la falsificazione e la tracciabilità, e l’entità dei problemi varia a seconda del fatto che si usi una contabilità “tradizionale” oppure “digitale”. Qualcuno, però, ha pensato ad un modo per risolvere queste vulnerabilità: passare completamente al digitale. Il programmatore Satoshi Nakamoto

(è uno pseudonimo, non si conosce il nome reale di questa persona) ha proposto, nel 2008, una idea di **cryptocurrency**, cioè di “moneta virtuale cifrata”. Il concetto stesso di cryptocurrency era già stato suggerito nel 1998, ma nessuno aveva mai realizzato prima una implementazione ben funzionante come quella di Satoshi Nakamoto, conosciuta col nome di Bitcoin.



I bitcoin sono legali?

Ci si potrebbe chiedere se i Bitcoin siano legali o meno. In effetti, essendo anonimi e non rintracciabili vengono utilizzati soprattutto per acquisto di materiale illecito o per ripulire denaro sporco. In realtà, però, si tratta di un bene di consumo come potrebbe esserlo una fotografia digitale. Quindi non è facile per le autorità proibirne l'uso. In Europa e negli Stati Uniti, quindi, l'uso dei Bitcoin come forma di pagamento è tollerato, anche se i governi lo sconsigliano visto che è da considerarsi sempre un investimento ad alto rischio. È capitato che le autorità statunitensi abbiano chiuso alcuni siti che consentivano di tenere portafogli Bitcoin online, requisendo le monete digitali: l'operazione è però avvenuta soltanto nei casi in cui era stata dimostrata l'esecuzione di attività illegali tramite i siti web in questione.

Come funziona una moneta digitale? Abbiamo visto che una moneta è in realtà un qualsiasi oggetto a cui viene dato un valore preciso, e che viene scambiata con altri oggetti come in una sorta di baratto. Quindi, per realizzare una moneta virtuale potremmo utilizzare un qualsiasi oggetto digitale, ed assegnargli un valore: una immagine, un file audio, una stringa di testo. Naturalmente, questi esempi non vanno bene, perché sono troppo facili da replicare: è necessario qualcosa di univoco, cioè una tipologia di oggetto in cui ogni

esemplare è riconoscibile e non duplicabile (un meccanismo, quindi, simile a quello dei numeri di serie sulle banconote). La matematica ci viene in aiuto, ed ecco quindi il concetto di cryptocurrency. L'idea è di utilizzare particolari funzioni matematiche che consentono di calcolare coppie di numeri tra essi collegati ma tali da non poter risalire ad uno dei due anche se si conosce l'altro, di modo che l'unico a conoscerli entrambe sia chi li ha calcolati. È la stessa logica della crittografia asimmetrica.

Rivest, Shamir, Adleman

Tutti noi abbiamo utilizzato la crittografia simmetrica: si sfrutta la stessa chiave per cifrare e per decifrare un messaggio. È il caso di una cifratura del tipo Cesare (cioè le lettere del testo vengono spostate, per esempio la A diventa B, la B diventa C e così via). Si tratta della forma più semplice e comune di cifratura, molti algoritmi di crittografia di uso comune sono "semplici" (per esempio quello degli archivi Zip o Rar). Questo tipo di cifratura ha un grosso svantaggio: visto che la chiave necessaria per decifrare un messaggio è la stessa usata per criptarlo, è fondamentale che il mittente invii al destinatario anche la chiave, oltre al testo cifrato. E questo complica le cose, perché se era davvero necessario proteggere il messaggio con la crittografia, non si può certo spedire la chiave senza alcuna misura di sicurezza: se qualcuno intercetta la posta, potrebbe ottenere sia il testo che la chiave, risalendo quindi al contenuto originale in chiaro (cioè non cifrato). Una soluzione al problema della distribuzione della chiave è dato dalla crittografia asimmetrica, nella quale la chiave usata per cifrare e quella necessaria a decifrare il messaggio sono diverse. Tale meccanismo era stato ipotizzato dai crittografi Diffie, Hellman, e Merkle, ed è stato reso possibile dall'algoritmo sviluppato dai matematici Rivest, Shamir, Adleman (che è per l'appunto chiamato algoritmo RSA). Come è

possibile avere due chiavi differenti per cifrare e decifrare un messaggio? In realtà è piuttosto semplice, lavorando con l'aritmetica dei moduli: per capirlo meglio, vediamo come funziona l'algoritmo RSA.

Per poter utilizzare questa cifratura abbiamo innanzitutto bisogno di due numeri primi, che chiameremo **p** e **q**. In teoria, questi dovrebbero essere molto grandi (altrimenti sarebbe troppo facile riuscire a scoprirli). Ci serve, poi un altro numero primo, più piccolo degli altri due: lo chiameremo **e**. Adesso possiamo calcolare il numero **N** come un semplice prodotto tra **p** e **q**. La chiave pubblica sarà data semplicemente dalla coppia **N** ed **e**. Quella privata, invece, si basa numeri **N** e **d**. Qui c'è un piccolo problema: il numero **d** si deve calcolare in modo tale che

$$(e*d) \text{ mod } ((p-1)*(q-1)) = 1$$

possiamo, per semplificare, chiamare

$$(p-1)*(q-1) = \text{phi}$$

La relazione da soddisfare per ottenere **d** deve quindi essere:

$$(e*d) \text{ mod } (\text{phi}) = 1$$

La cosa migliore per trovare **d** è andare per tentativi finché non si identifica un numero in grado di soddisfare l'equazione. L'operazione "mod" è quella di modulo (cioè il resto della divisione). Per esempio,

$$7 \text{ mod } 5 = 2$$

Poiché 7 diviso 5 fa 1 con il resto di 2. Ovviamente, **d** non dovrà mai essere rivelato perché serve a decifrare i messaggi cifrati con **e**.

Come si può capire, il punto debole della crittografia RSA sta nel fatto di basare la propria sicurezza sulla difficoltà di scomporre il numero **N** in fattori primi e quindi risalire a **p** e **q**. Il problema è che la potenza dei computer aumenta ogni anno, e con essa la loro velocità nel fattorizzare un numero. Per tale motivo si cerca continuamente di costruire chiavi sempre più grandi: è una sorta di grande corsa contro il tempo in cui, per ora, i "buoni" sono avvantaggiati rispetto ai "cattivi". Ma è probabile che entro qualche decina di anni la tendenza sarà invertita, e ci resterà solo la crittografia a

blocco monouso come strumento per proteggere la privacy. Ad ogni modo, per cifrare usiamo (per ogni lettera) questa relazione:

$$C = T^e \text{ mod } N$$

e per decifrare quest'altra:

$$T = C^d \text{ mod } N$$

Dove **C** è il testo cifrato e **T** quello in chiaro.



L'RSA è stata scoperta due volte

Rivest, Shamir, ed Adleman non sono stati i primi a scoprire la crittografia asimmetrica. Ma sono stati i primi a poterlo rendere noto. La crittografia tipo RSA venne infatti ideata da James Ellis, Clifford Cocks, e Malcolm Williamson nel 1974-1975. questi erano, però, dipendenti del centro segreto di crittografia GCHQ (servizi segreti del Regno Unito). Nonostante la loro scoperta non fosse stata immediatamente utilizzata dall'intelligence britannica, anche perchè i calcolatori non erano ancora in grado di eseguire conti così complessi, essi furono obbligati a mantenere il silenzio sulla crittografia asimmetrica, ed assistere quindi impotenti alla riscoperta dell'algoritmo da parte dei due trii di crittografi e matematici che abbiamo citato nell'articolo. Ovviamente, ciò non toglie merito ai ricercatori che oggi ricordiamo come scopritori "ufficiali" di questa tecnologia, perchè essi hanno eseguito le loro ricerche in totale autonomia, partendo da zero. Dovremmo, però, ricordare anche i dipendenti del GCHQ, che non hanno mai potuto ricevere i dovuti onori.

L'RSA è la più importante forma di cifratura attualmente esistente. Fondamentalmente tutta la sicurezza informatica si basa su questo algoritmo (per esempio la cifratura SSL delle pagine web, la firma digitale, ecc...). In particolare, è importante capire come funziona il meccanismo di firma digitale. Nel caso della crittografia, si rende pubblica la chiave per cifrare, mentre rimane privata la chiave per

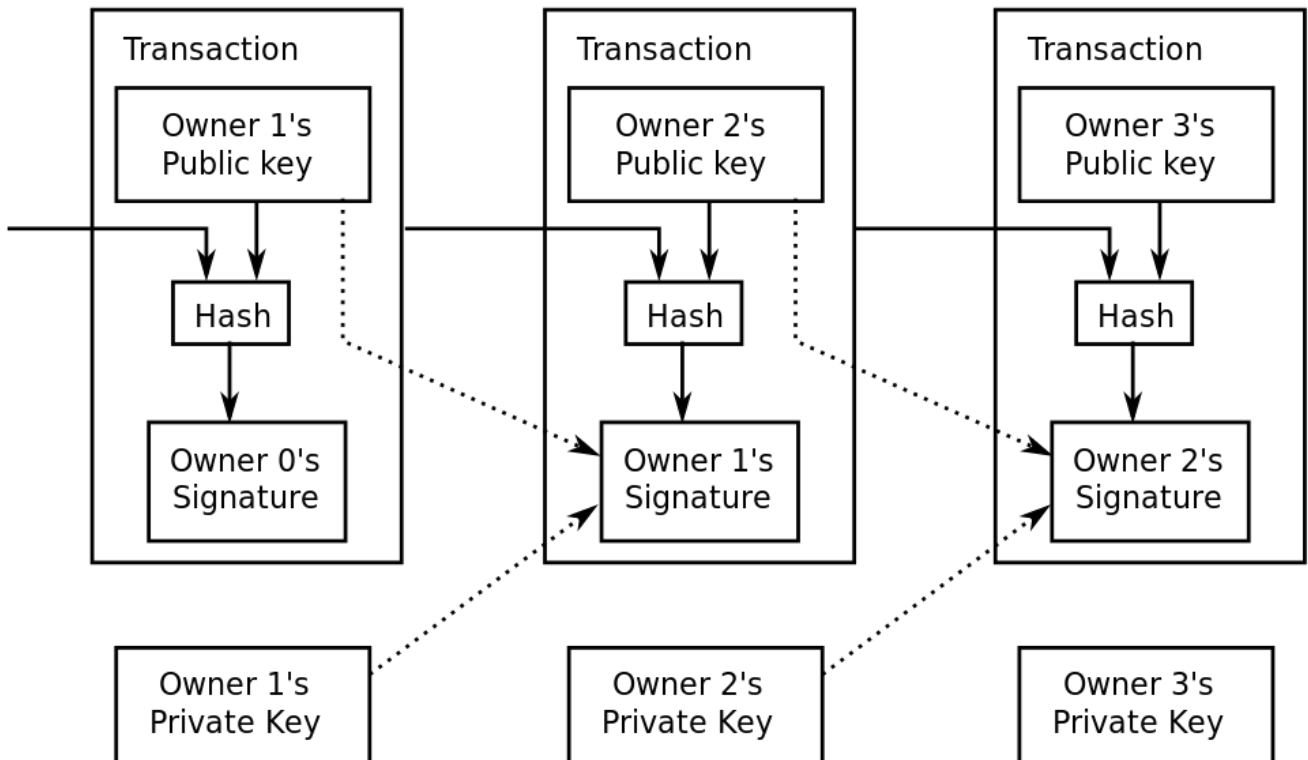
decifrare. In questo modo tutti possono scrivere un messaggio criptato ad una persona, ma solo questa persona potrà leggerlo (nemmeno il mittente originale potrà più decifrare il messaggio perché dispone solo della chiave pubblica). Invece, nel caso della firma digitale, si fa esattamente l'opposto. In pratica, la chiave utilizzata per cifrare è privata, l'altra è pubblica. Questo fa sì che un utente possa prendere un documento e cifrarlo con la propria chiave privata: l'operazione garantisce che a scrivere il documento sia stato proprio questo utente, perché è l'unico a conoscere la chiave di cifratura. Chiunque, invece, può leggere il documento utilizzando la chiave pubblica per decifrarlo: il meccanismo garantisce anche che nessuno abbia modificato il testo dopo l'apposizione della firma perché, se il testo viene modificato, l'algoritmo di decifrazione restituirà un testo errato.

Dai messaggi alle monete

Ora che abbiamo capito come funziona la crittografia asimmetrica per i messaggi, sarà facile comprendere il meccanismo su cui si basa Bitcoin. La rete Bitcoin è costruita come una rete P2P (come Kadmillà, per esempio). Ogni utente della rete ha diverse (potenzialmente illimitate) coppie di chiavi, che può raccogliere in un portafoglio digitale. In ciascuna di queste coppie, la chiave pubblica è resa nota a tutti e funziona da punto di invio o di ricezione per i vari pagamenti: è in un certo senso un po' come un IBAN. Per tale motivo la chiave pubblica, che per essere precisi è costituita da 33 caratteri (scelti tra lettere e numeri), viene chiamata indirizzo bitcoin. Invece, la chiave privata corrispondente è nota esclusivamente al suo proprietario e serve a consentire il pagamento (cioè la cessione, non la riscossione). In questo modo, solo il proprietario della coppia di chiavi può davvero autorizzare la cessione di alcune sue monete Bitcoin. Un Bitcoin viene "creato" come ricompensa per il calcolo di un blocco (vedremo tra poco che significa) ed è una stringa di testo, contenente l'indirizzo del proprietario, cifrata con la

chiave privata del proprietario stesso.

In seguito, se il proprietario vorrà cedere la sua moneta a qualcun altro per eseguire un acquisto, dovrà aggiungere ad essa l'indirizzo (la chiave pubblica) del nuovo proprietario e firmare il tutto con la propria chiave privata. Il fatto che si debba cifrare con la propria chiave privata fa sì che solo l'attuale proprietario possa porre la firma necessaria a concludere l'acquisto. Ciò significa che è sempre possibile risalire alla storia di una moneta: tramite l'ultima chiave pubblica è possibile decifrare la parte criptata. Questa è costituita da due parti: un'altra chiave pubblica e un altro "blob" criptato (da decifrare con l'indirizzo bitcoin appena citato). A sua volta, questo blob contiene un ulteriore indirizzo bitcoin e relativo blocco di testo criptato con esso. È quindi possibile proseguire fino alla stringa di testo originale, ricostruendo l'intera storia della moneta dal suo ultimo acquirente fino al creatore ottenendo tutti i vari indirizzi bitcoin attraverso cui la moneta in questione è passata.



In pratica, ogni moneta Bitcoin esiste in quanto oggetto di

una transazione economica: è addirittura costruita dall'insieme dei vari indirizzi che l'hanno posseduta. In fondo, ciò che conta di una moneta è poter eseguire delle transizioni: sono proprio queste il punto focale del meccanismo, quindi incentrare su queste il protocollo Bitcoin è la soluzione più semplice. Le transizioni sono pubbliche, anzi: quando si esegue un pagamento, le informazioni sulle monete (quantità di Bitcoin spesi, indirizzo del mittente e del destinatario) vengono inviate all'intera rete. Inoltre ogni client Bitcoin, per poter funzionare, deve scaricare dalla rete gli aggiornamenti in tempo reale sulle transizioni eseguite nel mondo. In questo modo è impossibile che un utente spenda due volte una propria moneta: quando esegue la prima transazione, tutti sanno che quel particolare Bitcoin è passato ad un altro utente e dunque non è più di sua proprietà.

Ma è davvero sicuro?

Si potrebbe obiettare che se davvero è possibile conoscere tutti gli indirizzi proprietari di una moneta, e tutti conoscono tutte le monete esistenti (grazie al database di informazioni), il sistema sia tracciabile. In realtà, però, gli indirizzi bitcoin vengono generati in modo casuale, e non possono essere collegati direttamente ad una persona: è infatti consigliabile utilizzare un indirizzo diverso per ogni transazione. Facciamo un esempio: se vogliamo permettere ai lettori del nostro blog di inviarci un pagamento tramite bitcoin, dovremo fornire pubblicamente un indirizzo a cui far arrivare i pagamenti. Ciò significa che tutti, Guardia di Finanza e Polizia Postale incluse saranno in grado di conoscere questo indirizzo ed i suoi movimenti, compresi e soprattutto quelli futuri: potrebbero facilmente scoprire quali oggetti servizi acquisteremo con quelle monete. La soluzione consiste nell'avere a disposizione diversi indirizzi bitcoin ma non dirlo a nessuno. Dal momento che le chiavi

pubbliche non si possono collegare alle persone in modo automatico, ma solamente se il proprietario decidere di farlo sapere a tutti, è sufficiente avere un indirizzo "pubblico", a cui tutti possono inviare denaro, e diversi indirizzi "privati" su cui trasferire le monete che arrivano a quello pubblico. In questo modo, eventuali inquirenti potrebbero sapere quante monete ci sono state cedute, ed anche quante ne abbiamo inviate ad altri indirizzi Bitcoin. Ma non potrebbero in alcun modo sapere chi sia il reale proprietario di questi indirizzi di arrivo: certo, potrebbero sospettare che si tratti di indirizzi che appartengono comunque a noi e che sia tutta una messa in scena per far perdere le tracce, ma non potrebbero dimostrarlo. Va anche detto che in realtà la "cronologia" di una moneta si può ricostruire solo grazie al database delle transazione, e non direttamente dalla moneta stessa. Questo perchè, di norma, invece di cifrare l'intero testo che costituisce la moneta per eseguire la firma che garantisce una transazione economica, si cifra soltanto un hash (con algoritmo SHA-2) del testo in questione. Un hash è un'altra stringa di testo ottenuta con una funzione di hash dalla stringa che rappresenta la moneta. La funzione di hash è per definizione non iniettiva, quindi non è biettiva e non si può invertire. Di conseguenza, dalla moneta si ottiene l'hash, ma dall'hash non si può ottenere la moneta. Tuttavia, ogni moneta realizza un unico hash, e questo può essere realizzato da una solo moneta: due stringhe di testo, cioè due monete bitcoin, differenti non possono generare lo stesso hash. Ciò significa che chi possiede una moneta può risalire fino all'hash generato dal suo precedente proprietario, ma lì deve fermarsi: non può ricostruire il contenuto della moneta prima che questa arrivasse nelle mani della persona da cui gli stesso l'ha ottenuta. Certo, può comunque risalire all'intera cronologia senza nemmeno bisogno di avere la moneta "in mano": gli basta controllare il database comune delle varie transazioni.

Come si producono

L'ultima cosa che dobbiamo ancora scoprire sui Bitcoin è in che modo si possono ottenere. In fondo, per ogni moneta è fondamentale stabilire in che modo possa essere prodotta, per indicare anche un limite di produzione (anche nella realtà, la banca centrale europea non può stampare tutti gli euro che vuole, ci sono delle regole da rispettare).

Di solito nessuno si preoccupa di come vengano creati i bitcoin, perché esistono già strumenti in grado di farlo. Il fatto è che per degli sviluppatori è fondamentale saperlo, soprattutto perché man mano che passa il tempo servono sistemi sempre più efficienti per produrre nuovi blocchi, e quindi è necessario che qualcuno sappia come programmare questi "generatori di blocchi".

Cos'è un blocco? Abbiamo detto che tutte le transizioni vengono comunicate, sottoforma di messaggio, all'intera rete Bitcoin. E la rete memorizza questi messaggi in diversi "blocchi", che sono quindi semplicemente dei contenitori realizzati periodicamente.

Questo permette a nuovi client di scaricare facilmente l'intero elenco delle transizioni mai eseguite: basta ottenere tutti i blocchi, che tra l'altro sono identificabili (ci sono i blocchi della settimana scorsa, quelli di due mesi fa, quelli di tre anni cinque mesi e tre giorni fa, eccetera).

I Bitcoin vengono forniti agli utenti come ricompensa per avere risolto un problema matematico, operazione chiamata "mining" (letteralmente "scavare in miniera"). Il problema in questione consiste nell'identificare un numero (chiamato **nonce**) tale che dopo essere stato sottoposto due volte all'algoritmo di hash SHA-2 si ottenga una stringa che inizia con un certo numero di zeri. Naturalmente, visto che la funzione hash non è invertibile, si deve procedere per tentativi (un po' come nel caso del brute force). Il numero di zeri che devono essere presenti all'inizio della stringa viene variato automaticamente per rendere l'operazione più semplice

o più complessa in modo che venga sempre generato, in media, un nuovo blocco ogni 10 minuti. La difficoltà di trovare il numero **nonce** aumenta esponenzialmente con la quantità di zeri che devono essere presenti all'inizio della stringa ottenuta con il doppio hash.

Ogni nodo della rete, rappresentato da un utente con il proprio client, si occupa quindi di calcolare questo numero ed utilizzarlo per ottenere assieme al contenuto del blocco su cui sta lavorando un hash. Quando riesce ad ottenere l'hash che inizia con il giusto numero di zeri, comunica l'avvenuta scoperta a tutta la rete. Quando gli altri nodi della rete riconoscono che la soluzione proposta è corretta, "archiviano" il blocco, che entra dunque a far parte del passato, e cominciano ad inserire le prossime transizioni che riceveranno in un blocco completamente nuovo. L'insieme in ordine cronologico dei vari blocchi è chiamato "catena dei blocchi". In pratica, la rete Bitcoin si comporta in questo modo:

- Quando due utenti eseguono una transazione, questa viene comunicata a tutti i nodi
- Ogni nodo minatore raccoglie le nuove transizioni di cui viene informato in un blocco
- Ogni minatore calcola la funzione per risolvere il problema matematico (questo è il mining vero e proprio)
- Quando un nodo trova la soluzione, cioè il numero **nonce**, la invia a tutto il resto della rete
- I nodi accettano il blocco soltanto se le transizioni in esso contenute sono valide
- Il minatore ottiene una ricompensa in bitcoin per il blocco contenente la soluzione che ha appena scoperto
- I nodi dichiarano di avere accettato il blocco ricevuto e cominciano a lavorare su un altro blocco utilizzando l'hash del blocco appena accettato.
- Si ricomincia da capo

Per ogni blocco vengono forniti dei Bitcoin, in numero

decescente nel tempo. Infatti, nel 2008 per ogni blocco venivano forniti 50 Bitcoin, mentre dal 2012 ne vengono prodotti 25 (e così sarà fino al 2016). La ricompensa verrà dimezzata ogni 4 anni fino ad arrivare a zero: secondo questo meccanismo, sarà possibile generare al massimo 21 milioni di monete. Visto che la complessità di calcolo aumenta sempre più, è oggi praticamente impossibile ottenere qualcosa con il proprio computer: è necessario utilizzare dei minicomputer dedicati (per esempio **ASIC**) oppure collaborare con altri minatori (per esempio tramite il sito www.bitminter.com). Nel caso della collaborazione, il sito si occupa di distribuire i bitcoin tra i vari utenti che hanno contribuito a risolvere il problema sul blocco. Ovviamente, visto che vengono forniti solo 25 Bitcoin e probabilmente ci sono ben più di 25 collaboratori tra cui distribuirli, ogni utente otterrà una frazione di moneta. In pratica, invece di ottenere 1 Bitcoin, riceverà 0,00X Bitcoin. Questo non è un problema, sia perché i Bitcoin possono essere divisi fino all'ottava cifra decimale, sia perché il valore dei Bitcoin può aumentare man mano che la gente li richiede.



Il valore dei Bitcoin

Il punto dolente dei Bitcoin sta proprio nel valore della moneta, ed è legato alla loro logica decentralizzata. Il motivo per cui i Bitcoin sono da considerarsi un investimento ad alto rischio è l'incredibile fluttuazione che il suo valore ha avuto e continua ad avere. Un bitcoin può passare dal valore di 50 euro a quello di diverse centinaia di euro nel giro di pochi mesi. Un andamento semplicemente impossibile per le monete tradizionali. Il fatto è che, non esistendo un organismo centrale, non esiste nessuno che possa stabilire uno standard per il valore della moneta e che possa quindi prendere le dovute misure contro l'inflazione e la deflazione. Il valore dei Bitcoin è legato quasi esclusivamente al

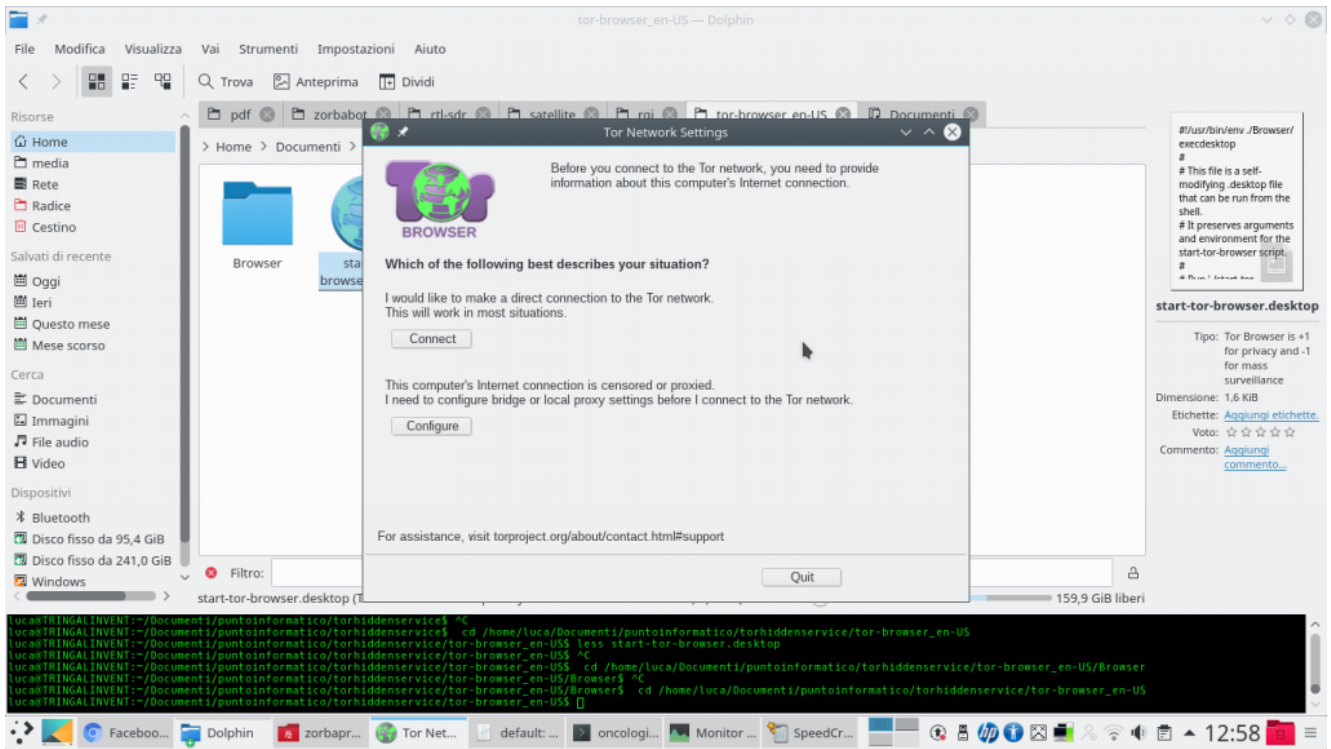
rapporto di domanda ed offerta: sono dunque gli utenti, con le loro decisioni di comprare o vendere le proprie monete che fanno salire o scendere il prezzo di un Bitcoin. E, considerando che ogni utente ragiona con la propria testa e per motivi personali, la variazione del valore della moneta è praticamente casuale.

Quando i bitcoin non potranno più essere prodotti (perché la ricompensa per ogni blocco sarà diventata pari a zero) i nodi potranno comunque continuare a realizzare i blocchi finanziandosi in altri modi, per esempio guadagnando dalle varie transazioni gestite.

Vendere online in Bitcoin

Per accettare pagamenti in Bitcoin sul proprio sito, è possibile sfruttare diversi servizi, come accept-bitcoin, che di fatto sono il PayPal per Bitcoin. Basta aggiungere al proprio sito un codice di questo tipo:

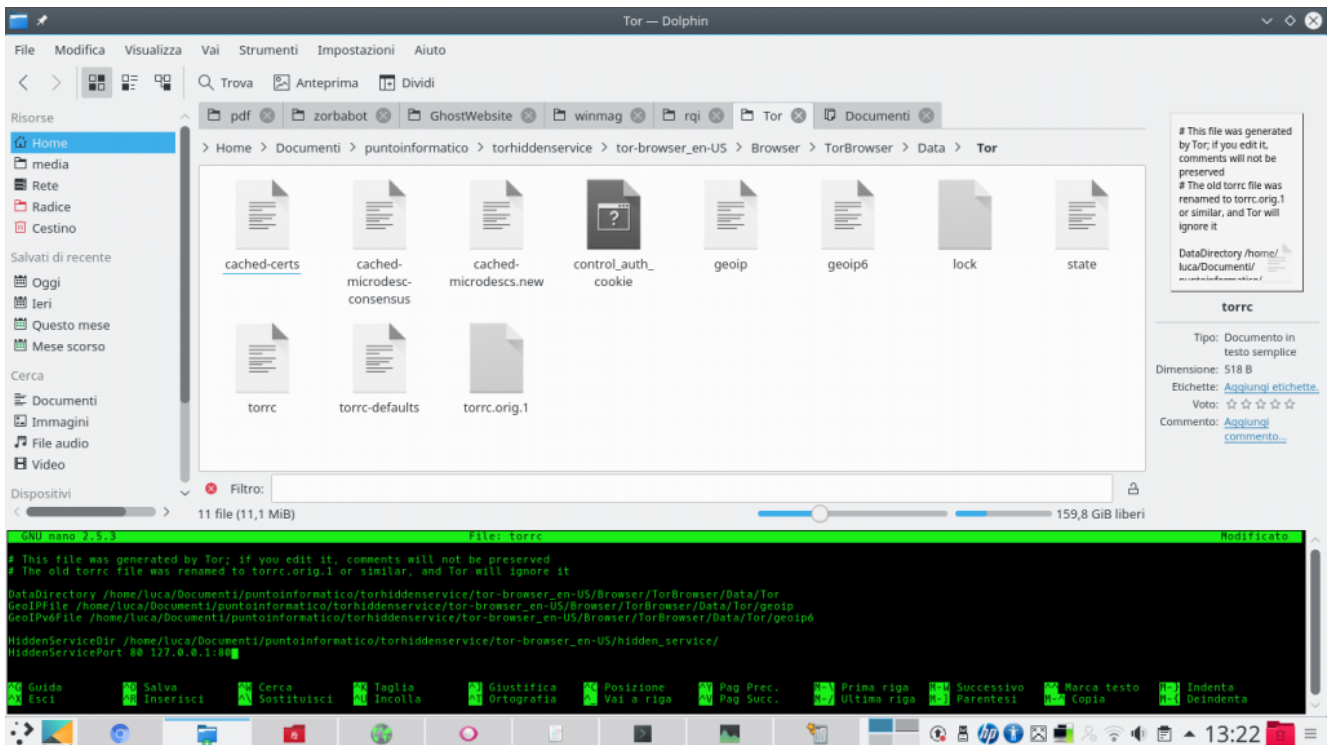
Nell'esempio stiamo vendendo 1Kg di Nduja per l'equivalente in Bitcoin di 20 euro. C'è anche una procedura guidata per realizzare questo codice, la si trova sul sito <https://accept-bitcoin.net/bitcoin-payment-button.php>. Ovviamente, pubblicare sul proprio sito web del materiale da pagare con Bitcoin ha poco senso: si viene facilmente tracciati. Molto meglio realizzare il proprio sito nel Dark Web, con Tor. Il bello di avere un server web che funziona sulla rete Tor, cioè un cosiddetto sito web Onion, è che non siamo identificabili: gli utenti del nostro sito web non possono risalire al nostro indirizzo IP reale, e quindi non ci possono identificare. Allo stesso modo, nessuno (noi compresi) può identificare gli utenti del nostro sito: tutti sono assolutamente anonimi.



Quando si scarica Tor Browser, la prima finestra che appare permette di configurare la connessione ad internet: nel caso ci si trovi dietro ad un proxy (come nelle reti aziendali), può essere necessario cliccare su Configure. Altrimenti, basta cliccare su Connect per iniziare la connessione a Tor. Il servizio Tor rimarrà attivo finché Tor Browser è aperto: se lo si chiude, anche la connessione a Tor viene terminata.

I siti web con dominio .onion sono per l'appunto anonimi e costituiscono il dark web, che è un sottoelemento del deep web. Il deep web più in generale rappresenta tutti quei siti non rintracciabili dai normali motori di ricerca, ma all'interno del deep web il dark web è statisticamente la parte più ampia e interessante (esistono comunque anche siti privati, che non sono accessibili da chiunque pur non essendo anonimi) e per questo i due termini sono spesso usati come sinonimi. Questo anonimato offerto dai server .onion può non essere particolarmente importante per un normale cittadino italiano, ma è fondamentale per chi si trova in paesi che limitano la libertà di stampa: un cittadino ucraino poteva (durante la rivoluzione) pubblicare un sito Tor con notizie sui crimini del governo, senza il rischio di essere scoperto ed arrestato. Per fare un altro esempio, la procedura di invio

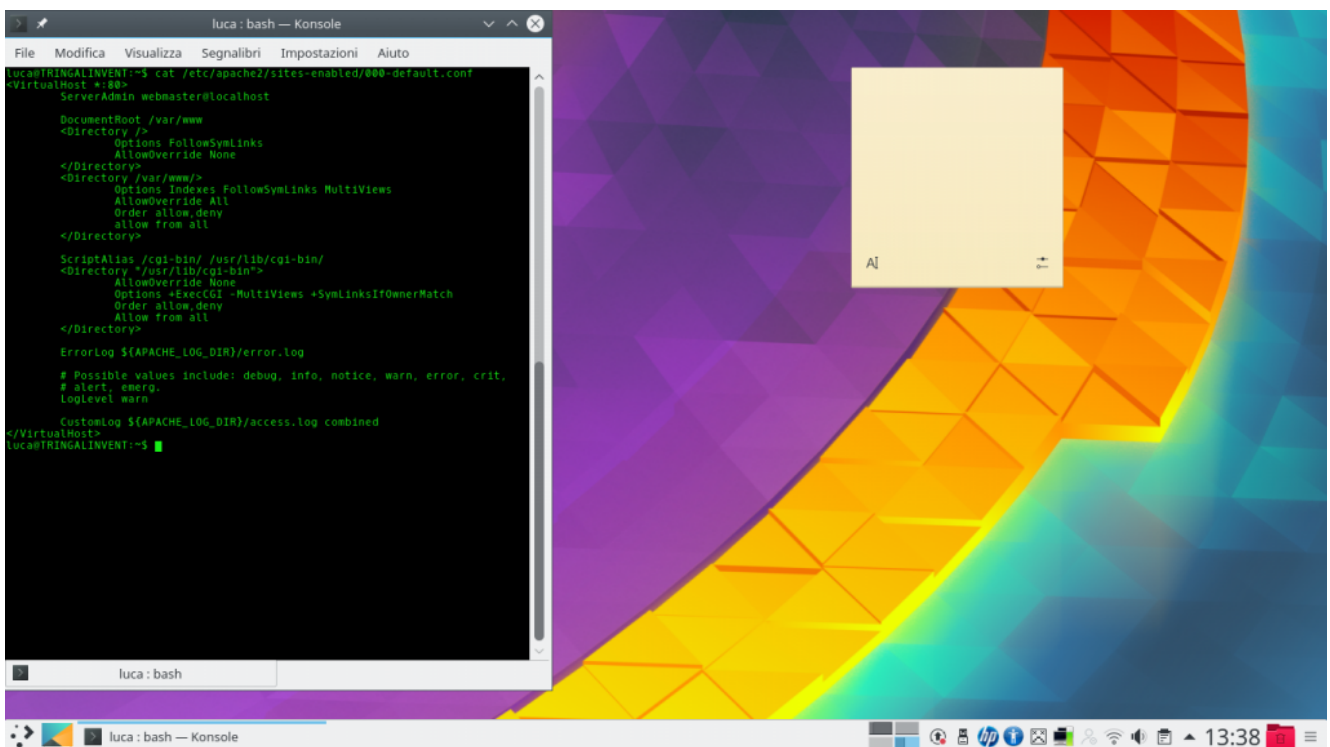
dei file da parte degli informatori a Wikileaks funzionava proprio usando un server web Tor.



Per abilitare il proprio Hidden Service è necessario aprire il file che si trova nella cartella **Browser/TorBrowser/Data/Tor/torrc** con un editor di testo, e inserire alla fine di esso le righe <https://pastebin.com/EEa4PegH>. In questo modo si indica come server quello che viene ospitato sulla porta 80, e come cartella dei dati `/home/luca/tor-browser_en-US/hidden_service/`. Ovviamente, si può scegliere una cartella qualsiasi, e una qualsiasi porta (FTP funzionerebbe sulla porta 21).

Considerata la struttura della rete Tor stessa il client Tor può essere utilizzato sia per accedere a dei servizi che per proporre i propri servizi al resto della rete. Infatti Tor è un protocollo di Onion routing, un tipo di rete simile a una VPN, con una differenza importante: ci sono molti passaggi intermedi. Una VPN è di fatto una sorta di rete locale molto estesa sul territorio, tutti i computer si collegano a uno stesso server centrale che funziona come un router domestico e

tiene assieme tutti i vari computer permettendo loro di condividere informazioni e un unico indirizzo IP pubblico. L'indirizzo IP pubblico è quello del server, quindi ogni client appare sul web con il suo indirizzo. Nell'Onion routing esistono molti passaggi intermedi: in pratica, un client si collega a un server, il quale si collega a un altro, poi un altro, e così via fino a un server finale che funziona come "punto di uscita" (un exit node). Il client apparirà sul web con l'indirizzo IP del punto di uscita, ma nessuno dei vari punti intermedi può risalire a chi sia il client e quale il nodo di uscita: ogni nodo intermedio della rete sa soltanto chi c'è prima di lui e chi dopo, ma non sa in che punto esatto della rete si trovi, quindi non sa se sta dialogando con il primo o l'ultimo computer della sequenza.



```
luca@TRINGALINVENTI:~$ cat /etc/apache2/sites-enabled/000-default.conf
<VirtualHost *:80>
    ServerAdmin webmaster@localhost

    DocumentRoot /var/www
    <Directory />
        Options FollowSymLinks
        AllowOverride None
    </Directory>
    <Directory /var/www/>
        Options Indexes FollowSymLinks MultiViews
        AllowOverride All
        Order allow,deny
        allow from all
    </Directory>

    ScriptAlias /cgi-bin/ /usr/lib/cgi-bin/
    <Directory /usr/lib/cgi-bin>
        AllowOverride None
        Options +ExecCGI -MultiViews +SymLinksIfOwnerMatch
        Order allow,deny
        Allow from all
    </Directory>

    ErrorLog ${APACHE_LOG_DIR}/error.log
    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    LogLevel warn

    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
luca@TRINGALINVENTI:~$
```

Ora è ovviamente necessario installare il server: se si vuole un server web, l'opzione probabilmente più scontata è Apache2, che può essere installato su sistemi Debian con il comando: *sudo apt-get install apache2 php5 libapache2-mod-php5 php5-mcrypt* Dando il comando *cat /etc/apache2/sites-enabled/000-default.conf* la riga DocumentRoot indica il percorso nel quale si possono inserire i vari file, per esempio la cartella

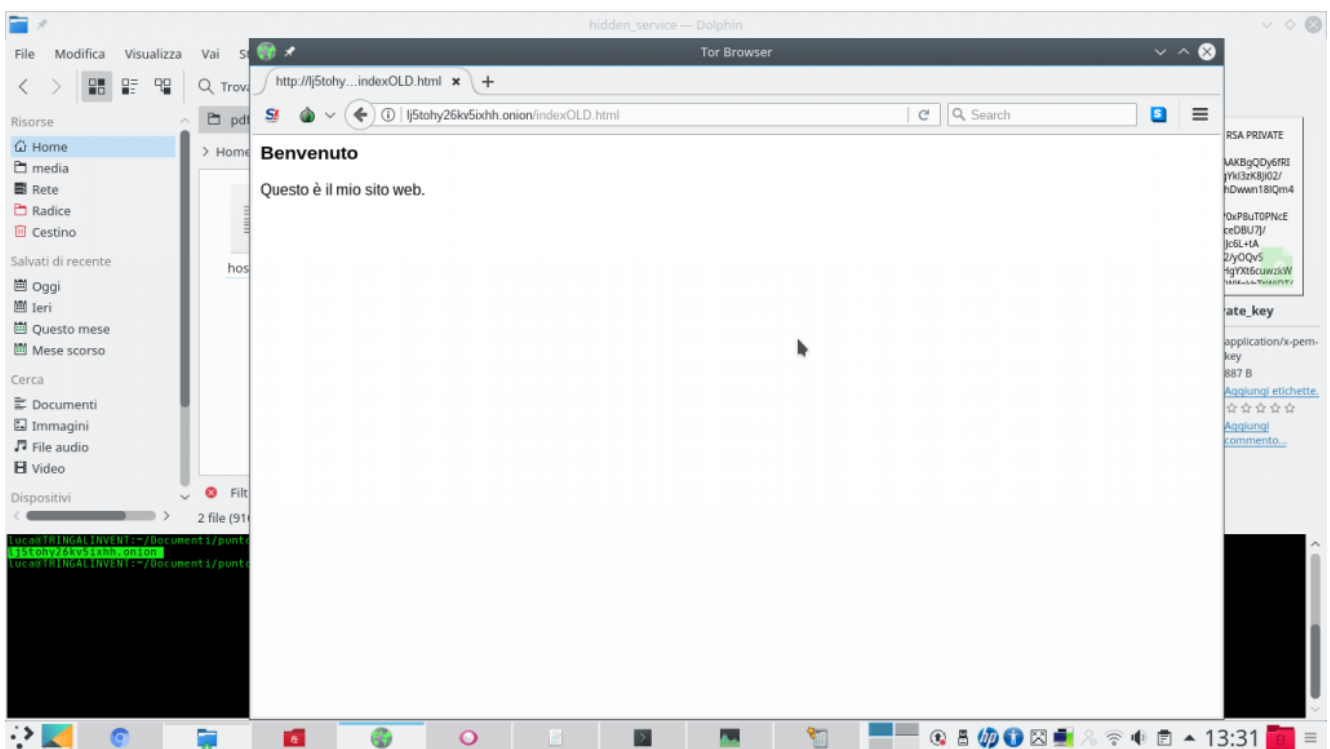
`/var/www/html/`. Il file `index.html` dovrebbe già esistere.

L'idea ricorda un po' le reti mesh, con la differenza che nelle reti mesh solitamente è possibile per tutti i computer essere punti di uscita per qualcun altro e non si può scegliere il proprio punto di uscita. Invece, nell'Onion routing i punti di uscita sono solo alcuni dei computer coinvolti, e un client può cambiare exit node se desidera apparire con un indirizzo IP differente da quello che ha avuto finora.

Inoltre, nell'Onion routing c'è un apposito sistema crittografico: un sistema a cipolla (il nome non è casuale, del resto). Ogni router intermedio si presenta al client con una chiave crittografica pubblica, che può essere usata per crittografare i pacchetti. Immaginiamo che un client riceva le chiavi di 4 diversi router: il client preparerà un pacchetto crittografandolo con la chiave 4, la chiave 3, la 2, e infine la numero 1. Il pacchetto verrà poi inviato ai vari router in sequenza, che lo potranno decifrare man mano: il primo router toglierà soltanto la crittografia con la propria chiave privata 1, il secondo con la chiave privata 2, eccetera. In questo modo, gli unici a poter vedere i dati non crittografati sono il client e l'exit node (e se si utilizza un protocollo come HTTPS nemmeno l'exit code può vedere i dati completamente decifrati, potrà farlo soltanto il vero destinatario). Le varie crittografie sono infatti applicate a strati come in una cipolla, e devono essere tolte esattamente in quell'ordine. Se qualcuno provasse a intromettersi il procedimento salterebbe e i dati diventerebbero automaticamente illeggibili, rendendo quindi tutta la comunicazione privata e di fatto anonima perché nessuno sa chi sia davvero ad avere cominciato la connessione.

Come si accede al server Onion

Per accedere ad un server Onion non si utilizza un indirizzo IP, ma un nome di dominio che viene creato in modo casuale sulla base di una altrettanto casuale e univoca chiave crittografica. Si tratta della chiave crittografica pubblica che permette ad altri client di crittografare, come abbiamo spiegato, i pacchetti di dati in modo che soltanto il nostro server sia in grado di leggerli usando la chiave privata di decifratura. Tutto quello che si deve fare per rendere il proprio client accessibile dalla rete Tor è selezionare le porte da “aprire” al resto del dark web (nel tutorial suggeriamo come fare per la porta 80).



Per conoscere l'indirizzo del sito web sulla rete Onion, accessibile anonimamente tramite rete Tor, basta leggere il contenuto del file `tor-browser_en-US/hidden_service/hostname`. È l'indirizzo che si può condividere, anche sui motori di ricerca. Inserendo l'indirizzo su un Tor Browser è possibile visualizzare il sito appena creato. Per assicurarsi che il

sito sia accessibile solo tramite rete Tor, e non sul web, è una buona idea chiudere la porta 80 sul proprio router.

C'è però una differenza importante con i server che si abilitano sul proprio computer per l'accesso dal web convenzionale: non si passa attraverso il meccanismo del NAT, perché il client Tor è già connesso direttamente alla rete Onion. Le normali reti locali casalinghe sono infatti gestite da un router, e sono quindi "nattate": questo significa, in parole povere, che l'unico computer visibile direttamente da internet è il router stesso. Gli altri dispositivi della rete non sono normalmente visibili, e le loro porte di comunicazione (il server web utilizza la numero 80) sono chiuse. È quindi fondamentale, se vogliamo che il nostro server sia raggiungibile da internet, aprire la porta 80 del nostro computer. Questa procedura è detta "port forwarding": se controlliamo il manuale del nostro router troveremo sicuramente una pagina in cui viene spiegato come eseguirla. Tuttavia, su rete Tor il server funziona anche se non abbiamo abilitato il port forwarding, perché la connessione viene gestita direttamente dalla rete Onion. Anzi: è molto meglio non aprire la porta 80 sul router. Così, l'unico modo per accedere al server sarà tramite la rete Tor: un utente (e questo vale anche per i programmi di scansione automatica come Echelon o Prism) che si trova su internet (e che quindi può leggere il nostro vero indirizzo ip) non sarebbe in grado di entrare nel sito. Se aprissimo la porta sul nostro router casalingo, il nostro server diventerebbe accessibile dal web normale, non anonimo, e prima o poi qualcuno lo troverebbe (potrebbe finire sul motore di ricerca Shodan). Se la teniamo chiusa, l'unico modo per arrivare al nostro server web sarebbe proprio attraverso il client Tor che abbiamo attivo sul computer, quindi saremmo protetti dal meccanismo di anonimato.



Le ultime versioni di Tor Browser contengono Selfrando, sistema sviluppato dall'università di Padova per proteggere il programma dall'esecuzione remota di codice

Facebook Scraping: scaricare tutti i post delle pagine Facebook

Da quando sono esplosi gli scandali relativi all'uso dei dati dei social network, come quello di [Cambridge Analytica](#), Facebook ha messo in piedi un sistema di controllo delle applicazioni. In poche parole, ora qualsiasi applicazione voglia accedere a ogni tipo di dato degli utenti deve prima superare un controllo in cui, in teoria, Facebook dovrebbe verificare che l'app non usi i dati in modo contrario alle norme di condotta previste dal social network.

Il problema è che, al momento, il sistema non funziona bene: ovviamente è appena partito, e si presume che in futuro verrà

migliorato, ma ha una serie di difetti fondamentali che saranno difficili da correggere a meno che Facebook non sia pronto a spendere davvero molte risorse finanziarie. Già adesso, infatti, ci sono delle persone incaricate di analizzare ogni app che viene sottoposta alla verifica, ma hanno migliaia di app da seguire e non hanno quindi il tempo di entrare nei dettagli. Soprattutto, nessuno controlla il codice sorgente delle app, quindi non c'è davvero una garanzia che questo controllo serva a impedire utilizzi impropri dei dati del social network. Allo stesso tempo, inoltre, i meccanismi di autorizzazione delle app offerti al momento sono insufficienti a coprire alcune delle app più legittime: parliamo di quelle che collezionano dati pubblici per realizzare statistiche (per pubblica amministrazione, università, e ricerca). Al momento è molto difficile farsi approvare una app di tipo desktop, l'opzione non è prevista e gli script non sono visti di buon occhio. Tuttavia, una università, un istituto di statistica, o una redazione giornalistica hanno in genere bisogno di accedere soltanto a dati come i vari post delle pagine dei personaggi famosi (per analizzare il loro linguaggio e capire come cambi la comunicazione in caso di eventi importanti, o controllare la veridicità delle affermazioni). Un esempio semplice è un team di ricercatori che voglia controllare sistematicamente la percentuale di verità dei post di un politico, il cosiddetto fact checking. In questi casi si vuole accedere soltanto a dati che sono già pubblici, e che quindi possono essere raccolti senza violare la privacy di nessuno.



Uno script con Python

Per il nostro script utilizziamo Python3: nonostante sul [sito ufficiale](#) venga ancora presentato il vecchio Python2 per questioni di retrocompatibilità, la versione 3 presenta delle

differenze importanti che rendono alcune funzioni incompatibili. Per essere sicuri di poter utilizzare lo script che presentiamo (alla fine dell'articolo trovare un link all'intero codice sorgente), bisogna installare sul proprio pc almeno la [versione 3.6 di Python](#).

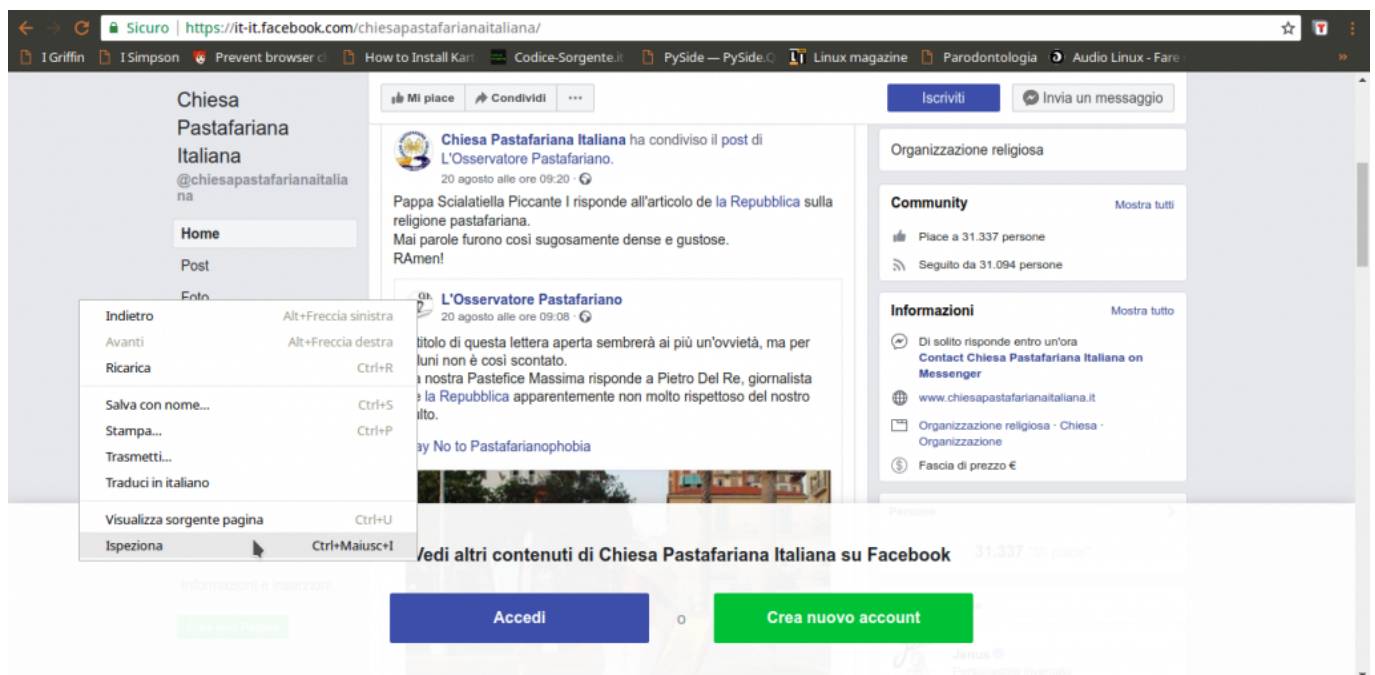
Abbiamo quindi pensato di realizzare uno script in Python che si occupi di eseguire lo scraping delle pagine Facebook pubbliche. Lo scraping è, per chi non lo sapesse, un insieme di tecniche di estrazione di informazioni da pagine web e altri documenti, in modo automatico, ripulendole da ciò che non serve. Un ricercatore universitario potrebbe scaricarsi i post di una pagina Facebook aprendola col browser e scorrendola verso il basso fino a visualizzarli tutti, selezionando il testo di ciascuno e copiandoselo. Ma sarebbe una operazione lunghissima e noiosa. Analizzando il codice delle pagine HTML che Facebook fornisce, invece, possiamo automatizzare l'estrazione dei testi (o delle immagini, se volete scaricarvi i meme delle vostre pagine preferite, basta modificare lo script per cercare i tag `img` invece dei tag `p`). E ovviamente lo script che realizziamo non richiede alcun accesso alle API o approvazione da parte di Facebook, perché di fatto faremo la stessa cosa che fa ogni utente che vuole guardare una pagina Facebook, permettendoci di bypassare tutte le verifiche che Facebook ha messo in piedi per le app.

Non dobbiamo, infatti, dimenticare un concetto fondamentale: se una informazione è disponibile, c'è sempre un modo non ufficiale per ottenerla. Quando carichiamo una pagina Facebook in Google Chrome, l'interfaccia realizzata con HTML e Javascript carica solo un certo numero di post. Quando "scrolliamo" la pagina, scendendo verso il basso, deve esserci una qualche funzione che si accorge che stiamo scendendo e quindi richiede al server un certo numero di nuovi post da visualizzare. È abbastanza ovvio che questa richiesta debba essere fatta, dalla pagina HTML+JS, con una richiesta HTTP (usando il meccanismo Ajax, quindi la funzione `xmlhttprequest`

di Javascript). Se ne deduce che da qualche parte all'interno della pagina ci deve essere un riferimento a un'altra pagina che fornisce un elenco di post da visualizzare. L'accesso a questi dati da parte di script automatici invece che dai normali browser web è una cosa che, a prescindere dai suoi sforzi, Facebook non potrà mai impedire.

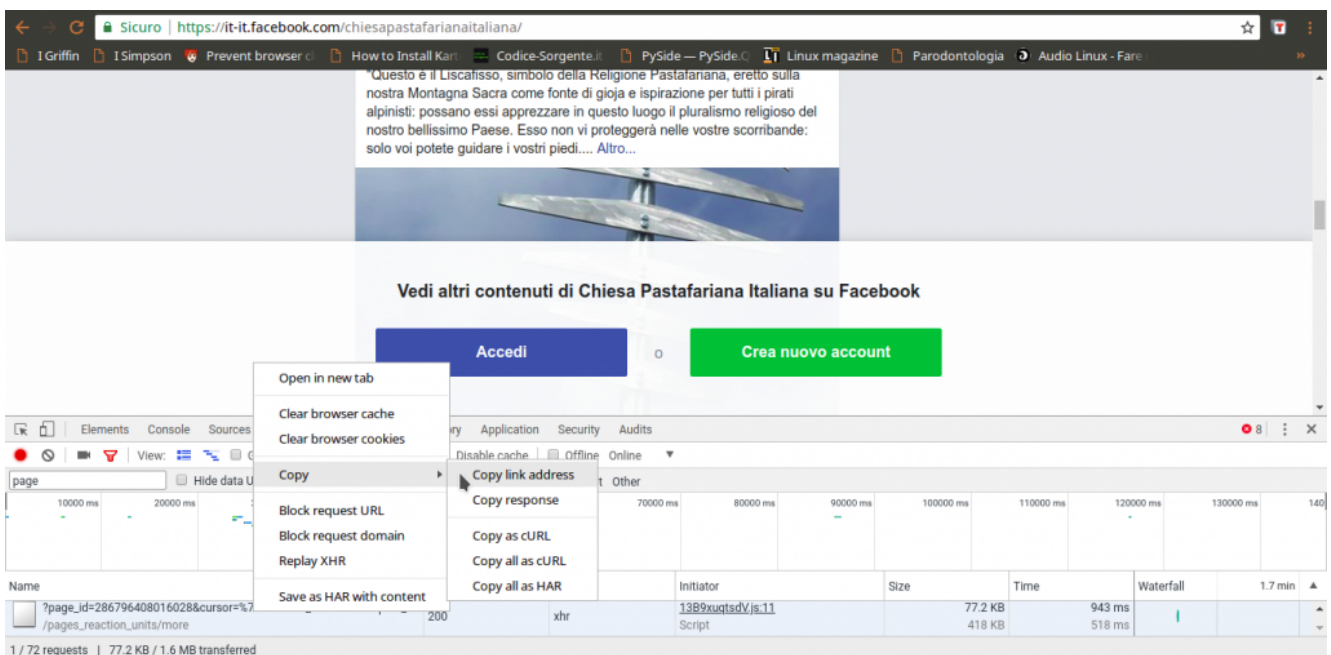
Scoprire l'indirizzo per ottenere i post

Per prima cosa dobbiamo scoprire come funzionano le API di Facebook, cioè come vengono recuperati i vari post. In poche parole, bisogna conoscere il proprio obiettivo. Siccome si tratta di un sito web, la cosa migliore da fare è aprire una pagina Facebook (per esempio <https://it-it.facebook.com/chiesapastafarianaitaliana/>) col proprio browser, come Google Chrome, cliccando poi col tasto destro sulla pagina per scegliere la voce **Ispezione**.



Tra le varie schede disponibili, quella che serve per capire cosa succede è quella chiamata **Network**: si occupa di

presentare in tempo reale le varie richieste HTTP che vengono fatte. Siccome è ovvio che la pagina di Facebook, per caricare altri post, abbia bisogno di fare una richiesta al server di Facebook per ottenerli, è anche ovvio che apparirà qui. Tutto quello che dobbiamo fare a questo punto è scorrere la pagina verso il basso, per obbligarla a caricare altri post: nel pannello vedremo comparire una richiesta a una pagina chiamata **page_reaction_units**. Sembra proprio che abbiamo trovato ciò che ci interessava: le altre eventuali richieste sono tutte relative a file accessori, come le immagini.



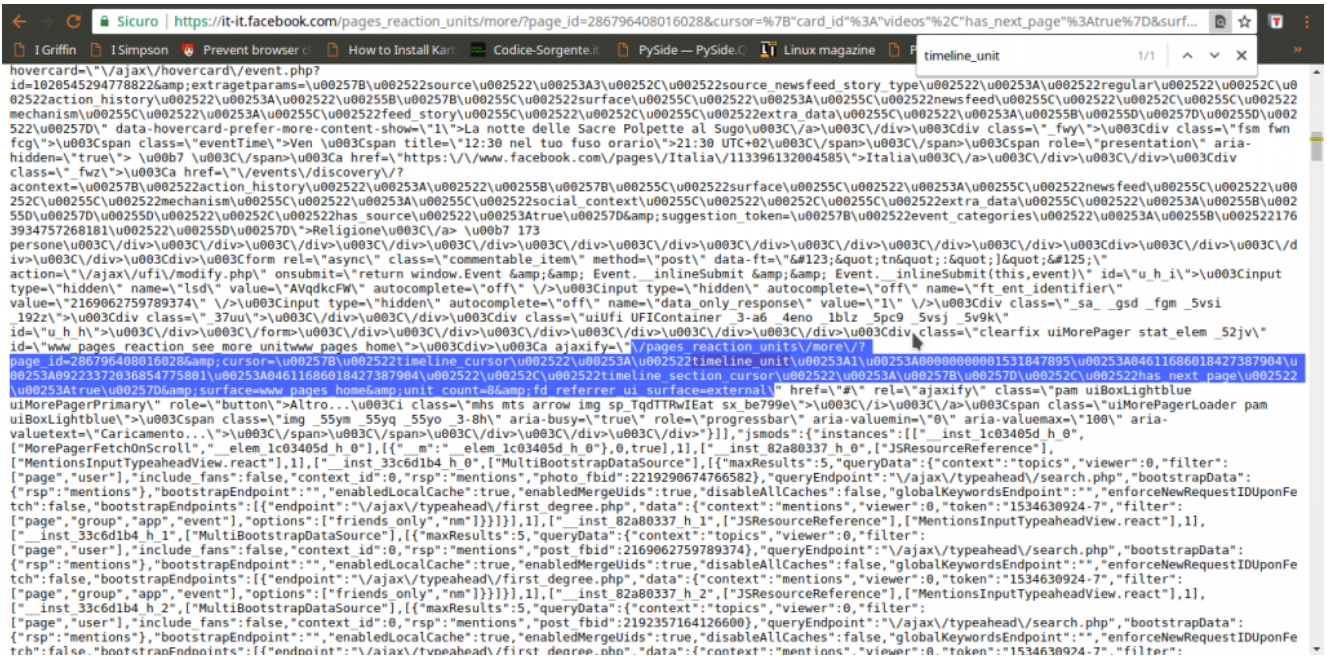
L'indirizzo della pagina contiene una serie di informazioni importanti, in particolare l'ID della pagina, ed è l'unica richiesta di questo tipo. Possiamo leggere il suo intero URL cliccandoci sopra col tasto destro del mouse e scegliendo **Copy link address**. Aprendo il link, si può capire che forma abbia la risposta: è una sorta di array JSON, una lista di oggetti vari, tra i quali il codice HTML necessario a presentare i post che sono stati richiesti. Si può facilmente distinguere il testo dei post in mezzo a tutto il codice. Alcuni caratteri vengono codificati come Unicode, inclusi alcuni pezzi dei tag HTML, e c'è sempre l'escape per i simboli /, che appaiono come \/, quindi è importante ricordarsi di convertirli in caratteri veri e propri, per poterli riconoscere facilmente (per

esempio, `\u003C/p>` è in realtà

).



Guardando meglio la risposta di `pages_reaction_units` si scoprono alcune cose interessanti. Innanzitutto, tutti i post che si sono ottenuti vengono presentati tra il testo `{“__html”:` e il testo `]]}`, quindi possiamo selezionarlo facilmente. Inoltre, ogni post è preceduto dalla sua data, in vari formati. In particolare, c'è la forma Unix Time, che è molto comoda da gestire ed è sempre identificata dalla dicitura `data-utime`, quindi potremo distinguere i vari post dividendo l'intero HTML in più pezzi dopo ogni occorrenza della parola `data-utime`. E non solo: si può anche capire come ottenere ulteriori post, facendo un'altra richiesta a questa stessa pagina. Se infatti cerchiamo `pages_reaction_units` all'interno della risposta possiamo notare che c'è l'intero indirizzo di una richiesta con quella che abbiamo appena inviato, ma contenente anche il blocco `timeline_cursor`, con i riferimenti della timeline di Facebook relativi ai prossimi post. Possiamo quindi facilmente estrarre questi riferimenti per confezionare la nostra prossima richiesta.



Ora, tutto quello che è rimasto da scoprire è come costruire l'indirizzo da contattare per ottenere i vari post: sappiamo che serve `pages_reaction_units` più una serie di argomenti (che vediamo nella richiesta extrapolata dal browser). Non tutti gli argomenti sono però necessari, e possiamo scoprire quali siano superflui semplicemente provando a cancellarli uno alla volta e vedendo in quali casi si ottiene comunque il risultato desiderato. Scopriamo quindi che l'indirizzo necessario è qualcosa del tipo: `https://it-it.facebook.com/pages_reaction_units/more/?page_id=286796408016028&cursor={\"timeline_cursor\": \"09223372036854775793:04611686018427387904\", \"has_next_page\": true}&surface=www_pages_home&unit_count=8&dpr=1&__user=0&__a=1`. Di questo indirizzo, abbiamo capito che la `timeline_unit` può essere scoperta all'interno di una richiesta precedente, mentre per scoprire l'ID della pagina Facebook basta scorrere il codice HTML della pagina stessa (che si può vedere nel browser Chrome con il prefisso `view-source:`) e cercare proprio `pages_reaction_units`, e subito dopo la parola `page_id`. Giocando un po' con `unit_count` scopriamo che per la prima richiesta possiamo ottenere fino a 300 post, mentre per tutte le successive (quelle in cui si specifica la `timeline_unit`) il massimo che

si può chiedere è 8 post. Tutto il resto è solo un insieme di argomenti vari sempre uguali, che nel nostro programma potremo quindi memorizzare sotto forma di variabili.



La procedura per scaricare tutti i post sarà quindi intuitiva: si accede alla pagina leggendo il suo codice HTML per scoprire l'ID. Con questo si forma il primo URL da contattare per ottenere gli ultimi post. All'interno della risposta si prendono i post dividendoli e salvandoli separatamente, e si cercano anche i riferimenti della timeline_unit per poter fare una nuova richiesta e ottenere altri post, più vecchi. Poi si ripetono continuamente gli ultimi passaggi, leggendo la risposta, salvando i post, costruendo il nuovo indirizzo, e facendo una nuova richiesta, finché Facebook non fornisce più alcun post (il che significa che siamo arrivati all'inizio della pagina e i post sono finiti). Ora dobbiamo tradurre questa idea in uno script Python.

Leggere le pagine web

Cominciamo lo script, tutto in un unico file che chiamiamo `scrapefb.py`:

L'inizio è dato dalla shebang (`#!`), che su sistemi Unix è utile per automatizzare l'avvio dello script trattandolo come un eseguibile. Poi si devono importare tutte le librerie necessarie: `urllib` permette di scaricare il contenuto degli URL, e `socket` permette di stabilire un timeout sulle connessioni per chiuderle se sono inattive. Per fare il parsing della pagina web, cioè per leggere il suo contenuto distinguendo i vari "pezzi", utilizziamo le espressioni regolari con la libreria `re`. Abbiamo anche bisogno di lavorare con data e ora, e accedere a funzioni relative al sistema operativo (per lettura e scrittura dei file).

Definiamo uno user agent: si tratta di una semplice stringa di testo che ogni sito richiede a chi vuole ricevere le pagine web, per capire di chi si tratti. Siccome possiamo scriverla come vogliamo, nessuno controlla davvero che stiamo dicendo la verità, possiamo scriverla in modo da convincere Facebook che il nostro script è in realtà il browser web Mozilla Firefox.

Definiamo una funzione che ci aiuti a scaricare tutto il contenuto di una pagina web, e la chiamiamo **`geturl`**. Prima di tutto, specifichiamo che ci serve lo useragent che abbiamo dichiarato nella sezione globale dello script. Poi ci assicuriamo di non procedere se l'url fornito alla funzione è vuoto, così evitiamo errori inutili. Costruiamo la richiesta HTTP utilizzando l'url. Servono anche un array di dati, che però in questo caso non è necessario visto che non abbiamo un form HTML da fornire alla pagina, e una intestazione. L'intestazione viene costruita con lo user agent, così Facebook ci scambierà per un browser web e non bloccherà la richiesta.

La richiesta HTTP può essere inviata usando la famosa funzione `urlopen`, e impostiamo anche un timeout. Il timeout è utile per non rimanere bloccati in eterno nel caso la connessione dovesse essere troppo lenta. Con un tempo di 300 secondi,

sappiamo che al massimo dopo 5 minuti la situazione verrà sbloccata. La risposta del server alla nostra richiesta può essere letta con la funzione `read`, e nel caso qualcosa non abbia funzionato impostiamo la risposta (variabile `ft`) come vuota.

In teoria potremmo tenere la risposta così com'è, ma non è una buona idea: il web è una giungla di codifiche, e se non gestiamo la cosa rischiamo di ottenere testi illeggibili. Soprattutto per Facebook, che spesso codifica le varie emoticon sotto forma di caratteri speciali Unicode. Cerchiamo quindi prima di tutto di capire se il server ci suggerisca la codifica della pagina che ci sta inviando. In caso negativo, proviamo a decodificare il testo con la classica `codepage` di Windows 1252, uno standard sui sistemi Microsoft precedenti a Windows 10. Se non dovesse funzionare, proviamo a decodificare tutti i caratteri usando l'`utf-8` togliendo però gli slash inutili (che spesso i server web forniscono per facilitare i browser), e altrimenti cerchiamo di tradurre direttamente l'intera pagina in una stringa python. Comunque sia andata, quindi, avremo una più o meno corretta stringa python piena di tutto il codice html della pagina. Per leggere meglio il suo contenuto, utilizziamo la funzione `html.unescape` per decodificare anche le varie entità dell'html (per esempio, `>` e `<` sono rispettivamente `>` e `<`, preziosi per interpretare il codice). L'`unescape` delle entità html non è fondamentale, ma rende il nostro lavoro più comodo.

Cercare l'ID della pagina Facebook

Cominciamo a scrivere la funzione vera e propria per lo scraping delle pagine di Facebook. La funzione richiede, come argomenti, l'indirizzo della pagina da scaricare, la cartella in cui salvare il risultato, e se si debba salvare il

risultato come tabella CSV invece che come testo TXT.

Innanzitutto ci sono un paio di informazioni, che possiamo memorizzare in alcune variabili. Potremmo anche scriverle direttamente nelle funzioni che le usano, come vedremo, ma tenendole nelle variabili è molto più facile modificarle in futuro se dovesse essere necessario a causa di modifiche nel funzionamento di Facebook. La variabile `TOSELECT_FB` contiene la stringa da cercare dentro la pagina Facebook per conoscere l'URL che fornisce i vari post. Le due successive variabili sono le stringhe che delimitano l'inizio e la fine dei post nella risposta. Infatti, Facebook non fornisce solo l'elenco dei post della pagina, ma anche una serie di altre informazioni che non ci servono. Per non complicarsi la vita, bisogna avere un output pulito, quindi toglieremo tutto ciò che non ci serve isolando solo il testo presente tra quei due delimitatori. Stabiliamo poi il numero di risultati che vogliamo: il massimo consentito da Facebook (al momento) per la prima richiesta è di 300 post. Inoltre, specifichiamo un periodo di attesa prima di inviare le richieste successive, per evitare che il server possa accorgersi che ne stiamo facendo troppe tutte assieme. Le ultime due rappresentano l'inizio e la fine del link per ottenere i vari post: vedremo tra un po' come costruirlo nella sua interezza.

In questo momento siamo pronti per eseguire la prima richiesta e scaricare la pagina Facebook. L'indirizzo che contattiamo è qualcosa del tipo **`https://it-it.facebook.com/chiesapastafarianaitaliana/`**. Ovviamente otteniamo soltanto gli ultimi post, proprio quello che un utente normale vede quando carica la pagina. Di per se i post che appaiono non ci interessano, li otterremo contattando direttamente l'URL che fornisce tutti i post. Il codice HTML di questa pagina ci interessa soltanto perché possiamo estrarre delle informazioni. In particolare, vogliamo scoprire il dominio di Facebook, cioè tutto quello che è

compreso tra **https://** e il primo / successivo. Nel caso in esempio è **it-it.facebook.com**, ovviamente è diverso per ogni paese (una pagina spagnola non inizierà con it-it). Cerchiamo anche di capire il nome della pagina, che è tutto ciò che segue il dominio: siccome lo useremo come nome del file in cui salvare i risultati è fondamentale che non ci siano caratteri strani. Usando una espressione regolare, cancelliamo (sostituiamo con "") tutti i caratteri che non siano lettere o numeri. Fondamentale per poter proseguire è il **pageid**, cioè il numero identificativo della pagina che vogliamo scaricare: questa informazione si può recuperare dalla pagina stessa perché è sempre presente in essa un link che contiene tale numero. Il link in questione ha la forma **?page_id=286796408016028&cursor**, quindi possiamo scoprire l'ID cercando ciò che segue la parola **page_id=** e arriva fino al simbolo **&**. Ci si potrebbe chiedere come mai per cercare i vari delimitatori utilizziamo direttamente la funzione `index`, molto pratica e veloce, mentre per cercare la posizione del **'pages_reaction_units'**, che determina l'inizio del link in cui troviamo la `pageid`, usiamo le `Regex`. La risposta è semplice: per ora trovare questa stringa è facile, ma in futuro potrebbe essere necessario usare una espressione regolare. In questo modo, lo script è già pronto per future modifiche.

Ora che abbiamo tutte le informazioni necessarie, possiamo costruire il nome del file in cui andremo a scrivere i post recuperati. Il nome è dato dalla cartella in cui salvare i file più **fb_** e il nome della pagina. Ovviamente, se l'utente vuole un `TXT` l'estensione del file sarà `TXT`, e se vuole un `CSV` l'estensione sarà `CSV`. Creiamo anche un altro file, con stesso nome la estensione **.tmp**. Questo è il file in cui andremo ad inserire i vari link già visitati, così se si deve riprendere lo scaricamento dei post di una pagina Facebook non lo si ricomincia da capo ogni volta, ma si riprende da dove ci si era interrotti. Per l'appunto, nel caso il file esista già vuol dire che non si deve ricominciare da capo, quindi si

carica l'intero contenuto del file in una lista, chiamata **alllinks**. In questa lista ogni elemento è un link, perché il file è stato diviso riga per riga (e quando lo scriveremo, metteremo un link in ogni riga). Definiamo anche una variabile che faccia da contatore, per sapere quante richieste di post siano state fatte, e una che stabilisca se stiamo ripristinando un download interrotto o se dobbiamo ricominciare da capo.

Richiedere i post della pagina al server di Facebook

Siamo al momento della raccolta vera e propria dei post della pagina. Siccome dobbiamo fare tante richieste una dopo l'altra, utilizziamo un ciclo. Il ciclo **while** andrà avanti finché la variabile **active** sarà True. Ne consegue che per fermare il ciclo, se necessario, non dovremo fare altro che porre tale variabile uguale a False.

Il link viene costruito unendo il dominio di Facebook, la parte iniziale del link, l'id della pagina, e la parte finale. Sarà quindi qualcosa del tipo **https://it-it.facebook.com/pages_reaction_units/more/?page_id=286796408016028&cursor={"card_id":"videos","has_next_page":true}&surface=www_pages_home&unit_count=300&referrer&dpr=1&__user=0&__a=1**, come si può notare ci sono tutti i vari pezzi che abbiamo costruito finora. Se provate ad aprire questo indirizzo col browser vi accorgete che fornisce una serie di informazioni, tra cui l'html dei vari post che sono stati richiesti (cioè gli ultimi 300 post della pagina). Inseriamo il link appena costruito nel file che li deve memorizzare, così se lo script dovesse bloccarsi mentre cercare di recuperare i post sapremo di dover ricominciare da questo preciso link, e non doverli rifare tutti da capo. Usando la

modalità di accesso al file "a" eseguiamo un "append", cioè inseriamo direttamente questo link alla fine del file, in una nuova riga, senza bisogno di preoccuparci di quali altri link ci fossero prima (non dobbiamo quindi aprire il file, leggerlo, aggiungere il nuovo link, e poi salvarlo). È un risparmio di risorse importante.

Sempre utilizzando la funzione `geturl` possiamo recuperare anche con il nostro script tutta la risposta del server di Facebook. Siccome ci interessa soltanto la parte con i vari post che abbiamo richiesto, la estraiamo e la memorizziamo nella variabile `postshtml`. Il codice HTML dei vari post va un po' ripulito: Facebook usa molti caratteri che non sono UTF-8 per gestire le emoticon, in genere sono utf-16. Però per il nostro scopo sono fastidiosi, le emoticon non ci interessano affatto e l'elaborazione dei testi è molto più facile con l'utf-8. Quindi ci assicuriamo di tradurre tutti i caratteri in UTF-8, togliendo anche l'escape dei caratteri speciali. Facebook, infatti, decide che alcuni caratteri sono particolari e li presenta con al loro notazione Unicode, una cosa del tipo `\u0001`. Questo è molto scomodo per noi, quindi forziamo la trasformazione in caratteri leggibili. A questo punto potrebbero essere rimasti dei simboli che UTF-8 non è in grado di gestire, perché si tratta delle famigerate emoticon UTF-16. Si riconoscono perché il codice Unicode è compreso tra `\uD800` e `\uDFFF`. Siccome non ci interessano, usiamo una semplice espressione regolare per cancellarli, sostituendoli con la stringa vuota `""`. Ora abbiamo finalmente l'intero codice HTML dei post, pulito e pronto per essere letto e interpretato. Siccome ogni post di Facebook è contrassegnato da un orario nel formato Unix Time (uno standard di internet), possiamo spezzare il contenuto dell'HTML nei singoli post dividendo proprio in base a `'data-utime'`, che è la stringa che Facebook usa per indicare l'orario di un post.

In questo momento, la lista `postsarray` contiene i vari post:

in realtà, il primo elemento della lista non contiene post, perché ha tutto l'HTML precedente. Comunque, possiamo scorrere la lista e individuare i post banalmente cercando il loro timestamp, cioè l'orario della pubblicazione. È facile da identificare, perché come dicevamo ogni post viene preceduto da una span (elemento HTML) che contiene una dicitura di questo tipo: `data-utime="1531306802" data-shorten="1" class="_5ptz">`. Siccome noi stiamo dividendo l'HTML a ogni "data-utime", è ovvio che ogni post inizierà con `con="1531306802" data-shorten="1"...`, e quindi l'orario in formato Unix sarà il primo numero tra virgolette (nell'esempio è **1531306802**). Per essere sicuri di non avere problemi, usiamo una RegEx per cancellare dal timestamp ottenuto qualsiasi cosa non sia un numero, e convertiamo il risultato in un `int`, cioè un numero intero. Nel caso non sia possibile risalire a questo numero, come per il primo elemento della lista che non è un vero post, consideriamo il numero pari a zero. Poi, usando `datetime`, possiamo convertire questo timestamp in una data facilmente leggibile, nel formato anno-mese-giorno ore:minuti:secondi. La data viene quindi aggiunta alla lista `timearray`, che abbiamo appositamente creato. Ciò significa che per ogni elemento di `postsarray`, cioè ogni post della pagina Facebook, abbiamo un corrispondente elemento di `timearray`, cioè la data della pubblicazione del post stesso.

Tutto il testo (se c'è) del post numero `i` si trova dentro all'elemento `postsarray[i]`, ma è ovviamente circondato da un sacco di altri pezzi di HTML che non ci servono. Per estrapolare soltanto il testo dei post basta prelevare tutto ciò che si trova all'interno dei paragrafi (che nella risposta di Facebook sono i tag

`</p>`). Bisogna ricordare che nello scrivere l'espressione regolare per trovare i tag il carattere `\` ha bisogno di una sequenza di escape lunga, e va scritto come `\\`. La funzione `finditer` crea l'array `indexes`, che contiene tutte le posizioni in cui si trovano i vari paragrafi: un post di Facebook può

infatti essere diviso in tanti paragrafi, e noi li vogliamo tutti. Ciascun elemento di **indexes**, contiene in realtà due informazioni: la prima (cioè **0**) è l'inizio del paragrafo, e la seconda (cioè **1**) è la fine del paragrafo. Usando il classico sistema di slicing delle stringhe di Python, si può banalmente estrarre il testo di ogni paragrafo semplicemente partendo dal carattere iniziale e finale (quindi **postsarray[i][start:end]**, perché la stringa è **postsarray[i]**). Alla fine del ciclo for che legge tutti i vari **indexes**, avremo la variabile **thispost** che contiene tutti i vari paragrafi uniti, senza gli altri tag inutili.

Possiamo assegnare tutto il testo del paragrafo all'elemento stesso da cui eravamo partiti, così lo avremo ripulito. Prima, però, togliamo i tag che ancora esistono. Per esempio, il grassetto viene realizzato con i tag ``, quindi noi cancelliamo tutto ciò che si trova tra i simboli `<` e `>`. E cancelliamo anche gli slash non necessari. Quindi, **gatti**/cani**** diventa **gatti/cani**. Alla fine presentiamo l'array sul terminale, così è più facile fare il debug e capire se qualcosa non vada bene. Lo scraping è pur sempre legato a qualcosa di molto casuale, e può capitare che in situazioni particolari qualcosa improvvisamente non funzioni.

Scoprire gli ID dei prossimi post da scaricare

Ora abbiamo ricostruito la lista **postsarray**, che contiene tutti i post presenti nell'attuale risposta di Facebook. Dobbiamo ancora capire come costruire la prossima richiesta, per ottenere una nuova risposta.

Le varie richieste che vengono inviate sono qualcosa del tipo https://it-it.facebook.com/pages_reaction_units/more/?page_id=

286796408016028&cursor={"timeline_cursor":"timeline_unit:1:0000000001528624041: 04611686018427387904:09223372036854775793:04611686018427387904", "timeline_section_cursor":{}}, "has_next_page":true}&surface=www_pages_home&unit_count=8&dpr=1&__user=0&__a=1. Se ci si fa caso, è praticamente identica alla prima richiesta, con due differenze fondamentali: l'argomento **cursor** contiene i riferimenti della timeline di Facebook che indica da dove iniziano i post da scaricare. E poi c'è la **unit_count** che è limitata a 8, quindi si possono scaricare al massimo 8 post per volta. Siccome lo stesso Facebook ha bisogno di sapere quali siano i riferimenti della timeline della pagina da scaricare, è ovvio che nella attuale risposta (quella che abbiamo appena ricevuto) ci debbano essere. E infatti ci sono, si possono trovare proprio nella forma dell'url con i due argomenti **cursor** e **unit_count**, quindi possiamo ottenerli cercando questi pezzi dell'URL dentro la stringa **newhtml** (che contiene l'ultima risposta che abbiamo ottenuto da Facebook). Siccome la prima parte dell'url è sempre la stessa, non dobbiamo fare altro che modificare la parte finale includendo i riferimenti della timeline appena ottenuti nella variabile **landing**. In questo modo, al prossimo ciclo verrà di nuovo costruito il **link**, ma usando questo **landing** come parte finale, e si potrà fare la nuova richiesta a Facebook per i successivi 8 post della pagina. Ovviamente, il testo della timeline estrapolato va un po' pulito, con le funzioni che avevamo già visto per la rimozione dei caratteri Unicode inutili e per la decodifica dell'url. Se non riusciamo a trovare un url per i prossimi post, vuol dire che sono finiti, quindi dobbiamo interrompere il ciclo impostando la variabile **active** come falsa.

Se per qualche motivo non è stato possibile recuperare i post e le date dei post, le due apposite liste vengono inizializzate come vuote, così il programma non si bloccherà.

È arrivato il momento di salvare il risultato in un file.

L'elenco dei post scaricati durante questo ciclo è nella lista **postsarray**, possiamo trasformarla in un testo da salvare nel file prendendo i vari elementi della lista e aggiungendoli alla variabile **postsfile**, uno in ogni riga (`\n` indica un invio a capo riga). Se si desidera che il file sia un CSV, il testo del post viene preceduto dalla data di pubblicazione del post, che si trova nella lista `timearray`. La data e il testo del post sono separati da una tabulazione, cioè `\t`, perché se utilizzassimo altri simboli come virgola e punto virgola il risultato sarebbe inaffidabile: un post di Facebook può facilmente contenere della punteggiatura, ma non una tabulazione.

Ora che il testo da scrivere nel file è tutto nella variabile **postsfile**, dobbiamo capire se sia necessario creare il file da capo oppure no. Se questa è la prima iterazione del ciclo, e non si sta eseguendo il ripristino di un download interrotto precedentemente, bisogna scrivere il testo nel file, dunque sovrascrivendo qualsiasi cosa ci fosse (se il file esisteva già). Altrimenti, bisogna soltanto aggiungere l'attuale testo a ciò che era già stato scaricato in precedenza, usando per la modalità di scrittura `append` (cioè **a**) che avevamo già visto.

Ovviamente, alla fine del ciclo si incrementa di 1 il contatore **timelineiter**, che ha per l'appunto la funzione di farci sapere quante iterazioni sta facendo il programma alla ricerca di altri post da scaricare. Inoltre, prima di concludere le operazioni del ciclo, e ricominciare da capo, attendiamo un certo numero di secondi. Questo è importante perché se riprendessimo immediatamente a fare richieste a Facebook, il server potrebbe accorgersi che stiamo insistendo troppo e magari bloccare la connessione.

Il blocco principale dello script

Terminata la funzione per lo scraping di Facebook, ricomincia il blocco principale dello script. Per sicurezza, controlliamo che in questo momento sia stata specificamente richiesta, dall'interprete Python, la funzione main. Questo avviene soltanto se lo script è stato lanciato direttamente dal terminale, e non se è stato importato in un altro script. Questo controllo facilita un eventuale utilizzo del nostro script come libreria per altri programmi.

Ora, si definisce la pagina Facebook da cui si parte, che può essere fornita dall'utente come primo argomento dello script. Il secondo argomento, se esiste, deve rappresentare la cartella in cui si vuole ottenere il file di output, e se non è specificato si suppone che sia la cartella attuale (cioè ./, presumibilmente la stessa in cui si trova anche lo script). Il terzo argomento, se esiste, può essere la parola "CSV": in questo caso, significa che l'utente vuole ottenere il CSV invece del TXT. Le varie informazioni vengono passate alla funzione di scraping per cominciare il download dei vari post. L'utilizzo è quindi molto semplice, e richiede praticamente soltanto l'indirizzo completo della pagina che si vuole scaricare, così come ogni utente può leggerlo in un browser web. Per esempio, si può lanciare lo script col comando **python3 scrapefb.py https://it-it.facebook.com/chiesapastafarianaitaliana/ ./ CSV** in modo da ottenere nella cartella corrente un CSV con data e testo di tutti i post della pagina della Chiesa Pastafariana Italiana, oppure si può usare il comando **python3.exe scrapefb.py https://it-it.facebook.com/chiesapastafarianaitaliana/ C:\Temp** per ottenere il TXT nella cartella **C:\Temp**.

Il codice completo



Potete trovare il codice completo dello script all'indirizzo <https://gist.github.com/zorbaproject/c1f8fff28cd0becea3a0fb6d0badd159>. Per utilizzarlo è necessario avere Python3 installato sul proprio sistema, ed è stato provato sia su GNU/Linux che su Windows.