UNIVERSITÀ DEGLI STUDI DI CATANIA

DIPARTIMENTO DI MATEMATICA E INFORMATICA

CORSO DI LAUREA MAGISTRALE IN INFORMATICA

Distributed Dictionary Attack

RELAZIONE

Studenti

Pietro Biondi

Giuseppe Parasiliti

Professore

Emiliano Tramontana

ANNO ACCADEMICO 2017/2018

Indice

Capitolo 1

Introduzione

Al giorno d'oggi si sente sempre più parlare di account compromessi. Spesso la causa di tutto ciò è la poca attenzione che l'utente mette nella scelta di password banali o comunque poco robuste e facilmente indovinabili. L'autenticazione basata su conoscenza è vulnerabile sotto diversi attacchi tra cui quelli a forza bruta e dizionario. Il progetto ha come obbiettivo quello di effettuare un attacco a dizionario in ambiente distribuito, dove coesistono diversi attaccanti con lo scopo di violare il sistema target ottenendo migliori risultati soprattutto dal punto di vista della velocità.

1.1 Tassonomia di attacchi alle password

1.1.1 Attacco a forza bruta

In informatica il metodo "forza bruta" è noto anche come ricerca esaustiva della soluzione. Esso è un algoritmo di risoluzione di un problema dato che consiste nel verificare tutte le soluzioni teoricamente possibili fino a che si trova quella effettivamente corretta. Nell'ambito della sicurezza informatica, questo metodo si utilizza soprattutto per trovare la password di accesso a un sistema. Utilizzando una parola italiana di 8 caratteri come password, la sua robustezza è data dal

numero totale di parole italiane di 8 caratteri. È quindi palese l'importanza di utilizzare password molto lunghe.

1.1.2 Attacco a dizionario

Un'altra tipologia di attacco per violare la password di un sistema può essere quello di sferrare un attacco a dizionario, che consiste nel provare le password contenute in un determinato dizionario. L'efficacia di un attacco a dizionario è direttamente proporzionale alla dimensione del dizionario stesso, in altre parole più è grande il glossario delle parole più è alta la probabilità di indovinare la password del sistema. L'efficacia di un attacco a dizionario può essere notevolmente ridotta limitando il numero massimo di tentativi di autenticazione che possono essere effettuati sul sistema e bloccando anche i tentativi che superano una certa soglia di esiti negativi. Generalmente, tre tentativi sono considerati sufficienti per permettere ad un utente legittimo di correggere i propri errori di battitura e accedere correttamente al sistema. Superata questa soglia, l'utente potrebbe essere malintenzionato.

1.2 Traccia dei contenuti

La relazione è strutturata come segue:

Capitolo 2 in questo capitolo vengono spiegati tutti gli strumenti utilizzati per lo sviluppo del progetto.

Capitolo 3 verranno affrontante in dettaglio l'implementazione del server.

Capitolo 4 sarà spiegato in dettaglio l'implementazione della classe Attaccante.

Capitolo 2

Strumenti utilizzati

Per questo progetto sono stati utilizzati diversi strumenti e tecnologie. Tutto il progetto è stato sviluppato con il linguaggio di programmazione Java ed inoltre è stato utilizzato AspectJ. AspectJ[1] è un'estensione di Java per aggiungere a Java stesso i cosiddetti aspetti. È uno dei modi utilizzati per avvalersi dell'Aspect Oriented Programming. La programmazione orientata agli aspetti è un paradigma di programmazione basato sulla creazione di entità software - denominate aspetti - che sovrintendono alle interazioni fra oggetti finalizzate ad eseguire un compito comune. Il vantaggio rispetto alla tradizionale Programmazione orientata agli oggetti consiste nel non dover implementare separatamente in ciascuna classe il codice necessario ad eseguire questo compito comune. Inoltre, per la stesura del codice è stato utilizzato Eclipse[2] un Integrated Development Environment(IDE). L'intera parte server-side è stata implementata con la tecnologia delle servlet. La comunicazione tra gli attaccanti avviene attraverso RabbitMQ[3], esso è un message-oriented middleware. Il server RabbitMQ è scritto in Erlang e si basa sul framework Open Telecom Platform(OTP) per la gestione del clustering e del failover. Le librerie client per interfacciarsi a questo broker sono disponibili per diversi linguaggi tra cui Java.

Capitolo 3

Sviluppo del server

Per lo sviluppo del server è stata utilizzata la tecnologia delle Servlet[4]. Per fare ciò è necessario un contenitore web che supporti la tecnologia Servlet, per questo motivo è necessario configurare un server Apache Tomcat. Eclipse mette a disposizione un wizard che guida lo sviluppatore nella configurazione totale del server web Apache Tomcat.

3.1 Sviluppo LoginServlet

Nel momento in cui il server riceve una richiesta di tipo POST, esso invocherà il metodo *doPost* [3.2]. Tale metodo recupera l'username e la password ed assegna un cookie al client che ha inviato la richiesta. Successivamente inserisce il client all'interno di una struttura dati che gestisce il controllo d'accesso al sistema.
Una funzione principale per il corretto funzionamento del server è *unBan*[3.1]. Tale metodo controlla se il client, identificato attraverso il suo cookie, è stato bannato. In caso affermativo si controlla se è il momento di rimuovere il ban, questo controllo viene effettuato sul numero di tentativi e sulla differenza di tempo trascorsa tra il momento del ban e il momento corrente. Come si può notare dalla [3.1], all'interno del server vi è un sistema di ban incrementale direttamente proporzionale al numero di tentativi effettuati. Il metodo unBan può ritornare tre

valori:

- Valore 1: Indica che il client ha aspettato il tempo necessario e potrà essere sbloccato.
- Valore 2: Indica che il client non ha ancora terminato il suo tempo di blocco.
- Valore 3: Indica che il client non è bloccato.

```
private int unBan(String cookie) {
    String user = getUserACL(cookie);
    int tentativi = getAttemptsACL(cookie);
    int flag;
    if (isBanned(cookie) == BANNED) {
        if( tentativi == 3 && diffTime( getTimeACL(cookie) ) >= 20 ) {
            ACL.put(cookie, new Object[] { getTimeinMills(), tentativi, NOTBANNED, user});
            flag = 1;
        }else if( tentativi == 7 && diffTime( getTimeACL(cookie) ) >=30 ) {
            ACL.put(cookie, new Object[] { getTimeinMills(), tentativi, NOTBANNED, user});
            flag = 1;
        }else if( tentativi == 11 && diffTime( getTimeACL(cookie) ) >= 40 ) {
            ACL.put(cookie, new Object[] { getTimeinMills(), RESET_TENTATIVI, NOTBANNED, user});
            flag = 1;
        }
        else
            flag = 2;
    }else
        flag = 3;

    return flag;
}
```

Figura 3.1: Funzione unban

In base al valore restituito dalla funzione *unBan*, il server restituisce al client tre tipologie di stato HTTP differenti. Nello specifico:

- Stato 270: Tale valore indica al client che è momentaneamente bloccato.
- Stato 271: Esso indica al client che ha ancora dei tentativi a disposizione.
- Stato 200: In caso di corretto login, il server restituirà al client tale valore.

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

    String user = request.getParameter("user");
    String pwd = request.getParameter("pwd");
    Cookie loginCookie = new Cookie("user", user);
    loginCookie.setMaxAge(30 * 60);
    HttpSession session = request.getSession();
    String sessionID = session.getId();
    System.out.println("ID SESSIONE: "+sessionID);

    addUserACL(sessionID,user);
    if( unBan(sessionID) == 2 ) {
        RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");
        PrintWriter out = response.getWriter();
        out.println("<font color=red>You are banned Wait and try later again.</font>");
        response.setStatus(270);
        rd.include(request, response);
    }else if (userID.equals(user) && password.equals(pwd)) {
        ACL.put(sessionID, new Object[] { getTimeinMills(), RESET_TENTATIVI, NOTBANNED, user});
        session.setAttribute("user", user);
        response.addCookie(loginCookie);
        response.sendRedirect("LoginSuccess.jsp");
    }else {
        increaseAttempts(sessionID);
        System.out.println("TENTATIVI =" + getAttemptsACL(sessionID));
        RequestDispatcher rd = getServletContext().getRequestDispatcher("/login.html");
        PrintWriter out = response.getWriter();
        out.println("<font color=red>"
                + "Either user name or password is wrong IncreaseAttempts."
                + "</font>");
        response.setStatus(271);
        rd.include(request, response);
    }
    doGet(request, response);
```

Figura 3.2: Funzione doPost

3.2 Aspect LogServer

Nella figura sottostante, è implementato un pointcut che verrà eseguito nel momento in cui viene lanciato il metodo addUserACL. Tale pointcut stamperà l'ID (cookie) dell'utente che ha provato a connettersi.

```
public aspect LogServer {
    pointcut callDoPost(String session): execution(* LoginServlet.addUserACL(..)) && args(session, ..);

    after(String session) : callDoPost(session)  {
        System.out.println("ID: "+session+ " Ha provato a connettersi");
    }
}
```

Figura 3.3: Aspect logserver

Nella figura [3.4] si può notare l'effetto del pointcut sopracitato, sulla console di Eclipse.

Figura 3.4: Output pointcut server

Capitolo 4

Implementazione classe Attacker

In questo capitolo viene spiega l'implementazione della classe Attacker, che si occuperà di trovare la password per violare il sistema. Le funzioni principali utilizzate all'interno della classe sono: *run*, *sendMessage*, *receivedMessage*.

4.1 Funzione run

Una volta invocata questa funzione, il client setta i parametri fondamentali per poter attaccare il sistema. Inizialmente, legge da un file il dizionario contenente tutte le password da provare. Successivamente, attraverso una richiesta GET al server, il client estrae le informazioni essenziali dal form che gli permetterà di costruire la richiesta POST in maniera corretta. Dopo di che verranno scelte le prime tre password con indice random che saranno inviate al server.

```
while (getPasswFound() == false) {
    for (int i = 0; i < 3; i++) {
        String postParams = "";
        try {
            postParams = this.getFormParams(page, "prova", passwToSend[i]);
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }
        try {
            response[i] = this.sendPost(login, postParams);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    for (int i = 0; i < response.length; i++) {
        if (response[i] == 200) {
            try {
                sendMessage("OKAY;" + passwToSend[i]);
            } catch (IOException e) {
                e.printStackTrace();
            }
            setPasswFound(true);
        } else if (response[i] == 271) {
            try {
                sendMessage(getName() + ";" + passwToSend[i]);
            } catch (IOException e) {
                e.printStackTrace();
            }

            passwordRecieved.add(passwToSend[i]);
            passwToSend[i] = getNewPasswToSend(passwordRecieved, data);
        } else if (response[i] == 270) {
            passwToSend[i] = getNewPasswToSend(passwordRecieved, data);
            ban = true;
        }
```

Figura 4.1: Funzione Run

Nella figura 4.1 vengono costruiti tutti i parametri che successivamente verranno inviati al server attraverso la funzione *sendPost*. Un esempio di questi parametri è:

“user=prova&pwd=prova”

Tale funzione restituirà, all’interno di un array(response), i valori dello status http che il server ha restituito per quella coppia username-password. Ogni risposta, all’interno dell’array response, verrà gestita con il costrutto “if annidato” che eseguirà differenti istruzioni a seconda del valore della risposta.

- Se il valore della response è pari a *200*: l’attaccante ha trovato la password e comunicherà a tutti gli altri attaccanti tale informazione.

- Se il valore della response è pari a *271*: l'attaccante ha ancora a disposizione dei tentativi, invierà la password errata agli altri attaccanti per informarli di non utilizzarla.

- Se il valore della response è pari a *270*: l'attaccante entrerà nello stato di ban. Quando l'attaccante è bloccato, entra in una fase di attesa che potrà finire in due casi: quando un altro attaccante ha trovato la password oppure allo scadere del tempo di ban.

Tutto il procedimento descritto dalla figura 4.1 avviene fino a quando la password non verrà trovata.

4.2 Funzione sendMessage

La funzione descritta nella figura 4.2 permette di inviare la password in broadcast a tutti gli altri attaccanti. Questo avviene grazie alla funzione *basicPublish* messa a disposizione dal middleware RabbitMQ.

```
private void sendMessage(String message) throws UnsupportedEncodingException, IOException {

    this.channel.basicPublish(this.exName, "", null, message.getBytes("UTF-8"));
    System.out.println(name + " MANDA: " + message + "\n");
}
```

Figura 4.2: Funzione sendMessage

4.3 Funzione receivedMessage

La funzione receivedMessage crea una classe anonima di tipo Consumer che effettuerà un Override del metodo *handleDelivery*. Attraverso l'esecuzione della funzione*channel.basicConsume* diremo al server RabbitMQ che la coda "queueName" dovrà essere utilizzata da quel "consumer". Ciò è dovuto grazie alla callback handleDelivery che permette l'esecuzione del programma e non appena un messaggio viene inserito nella coda, il server Rabbit avviserà il client della presenza

del nuovo messaggio. Una volta che il client consuma la coda esegue le istruzioni all'interno del metodo handleDelivery.

```
public void receivedMessage() throws IOException, TimeoutException {

    System.out.println(name + " Waiting for messages. To exit press CTRL+C");

    Consumer consumer = new DefaultConsumer(channel) {
        @Override
        public void handleDelivery(String consumerTag, Envelope envelope, AMQP.BasicProperties properties,
                byte[] body) throws IOException {
            String mexReceived = new String(body, "UTF-8");
            String sender = mexReceived.split(";")[0];
            mexReceived = mexReceived.split(";")[1];

            if (!(sender.equalsIgnoreCase("OKAY"))) {
                if (!(sender.equalsIgnoreCase(getName()))) {
                    passwordRecieved.add(mexReceived);
                    System.out.println(getName() + " HA RICEVUTO DA " + sender + ": " + mexReceived + "\n");
                    System.out.println(
                            getName() + " HA IL SEGUENTE passwordRecieved " + passwordRecieved.toString() + "\n");
                }
            } else {
                setPasswFound(true);
                System.out.println(sender + " ha Trovato la Password: " + mexReceived + "\n");
            }
        }
    };

    this.channel.basicConsume(this.queueName, true, consumer);
}
```

Figura 4.3: Funzione receivedMessage

4.4 Pointcut LogAttacker

Nella figura sottostante, è implementato un pointcut che verrà eseguito nel momento in cui viene lanciato il metodo sendPost. Tale pointcut è diviso in due fasi differenti (before - after). Questo pointcut stamperà nella console: i parametri post che invierà al server e la risposta ricevuta da quest'ultimo.

```
public aspect LogAttacker {
    pointcut callSendPost(Attaccker a, String i): call(* Attaccker.sendPost(..)) && this(a) && args(..,i);

    before(Attaccker a, String i) : callSendPost(a,i) {

        System.out.println(a.getNameAttack()+" provo questa combinazione "+i+"\n");
    }

    after(Attaccker a, String i) returning(int r) : callSendPost(a,i)  {
        System.out.println(a.getNameAttack()+": La coppia "+i+" ha generato la seguente RESPONSE: "+r+"\n");
    }
}
```

Figura 4.4: Pointcut logattacker

Nella figura sottostante si notare tutte le stampe eseguite dall'aspetto LogAttacker.

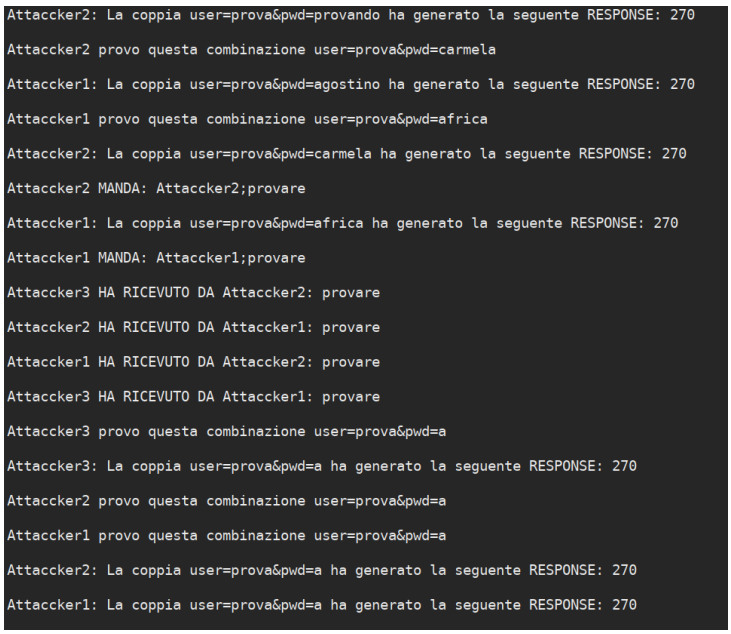

Figura 4.5: Pointcut print console

Bibliografia

[1] Foundation, E.: AspectJ. https://www.eclipse.org/aspectj/ (2018)

[2] Foundation, E.: Eclipse (IDE). https://www.eclipse.org/ (2018)

[3] Pivotal: RabbitMQ. https://www.rabbitmq.com/ (2018)

[4] Wikipedia: Servlet. https://it.wikipedia.org/wiki/Servlet (2018)