Java Security

LORENZO FALLANI

25/03/2024

Sicurezza Generale in Java

1. Sicurezza del Linguaggio e Bytecode Verifier
2. Security Provider
3. Java Cryptography Architecture
4. Management di certificati e chiavi
5. Java Secure Socket Extension
6. Java Authentication

Sicurezza del Linguaggio Java e Bytecode Verifier

Il linguaggio Java è stato concepito per essere type-safe ovvero riesce a prevenire errori di tipo.
Riesce a gestire la memoria automaticamente facendo utilizzo di garbage collection e effettua il controllo sui range degli array, per prevenire un possibile buffer overflow.

Un Bytecode Verifier viene invocato per assicurarsi che solo i bytecode legittimi siano eseguiti nella JVM.
Esso controlla anche eventuali violazioni di memoria e typecast illegali.

Infine Java mette a disposizione delle keyword di accesso che possono essere assegnate alle classi, ai metodi o ai campi per permettere agli sviluppatori di filtrare l'accesso alle varie implementazioni.
Essi sono:

1. private → Nessun accesso è possibile fuori dalla classe
2. protected → E' permesso l' accesso solo dalle sottoclassi o alle classi dello stesso package
3. public → E' sempre permesso l' accesso

Security Provider

La JDK definisce un insieme di API per accedere alle più conosciute funzioni di sicurezza.

Queste API vengono implementate secondo i principi di indipendenza, interoperabilità e estensibilità, ed esse prendono forma attraverso i Security Provider.

Ogni Security Provider incapsula una lista di servizi di sicurezza già implementati e permette di accedervi attraverso delle interfacce standard.

Attraverso il metodo getInstance sulla classe Provider è possibile richiedere un determinato servizio di sicurezza di un determinato Provider.

Ad esempio:

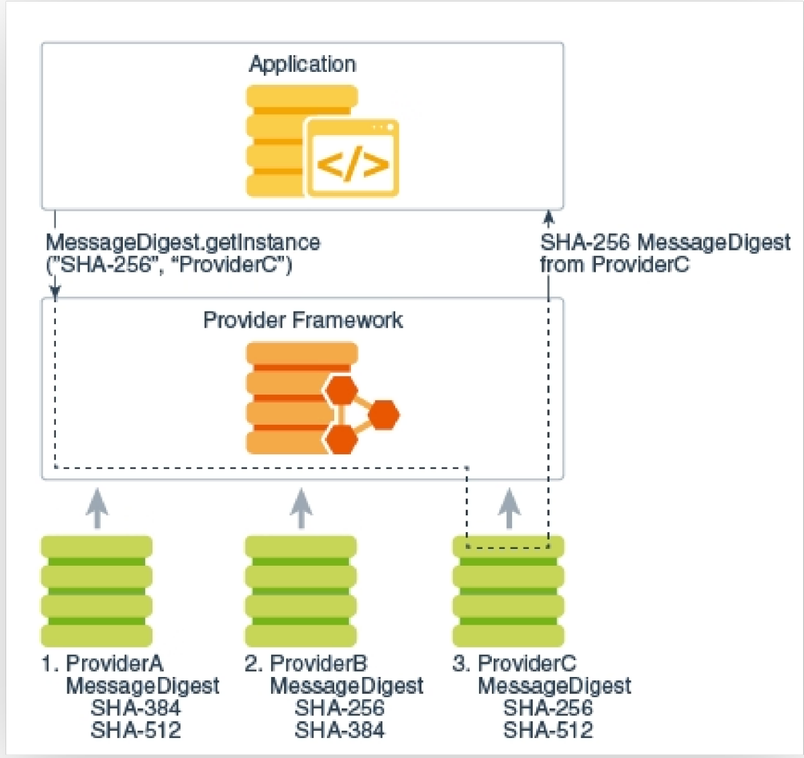
```
MessageDigest md = MessageDigest.getInstance("SHA-256","SunJSSE");
```

In questo caso si richiede il Provider MessageDigest che usa come algoritmo SHA-256 implementato da SunJSSE.

Security Provider

La JDK di default mette in ordine di preferenza le varie implementazioni fornite dai vari provider. Infatti nel caso in cui non venga specificato un Provider la JDK ci fornisce il servizio implementato dal Provider con priorità più alta.
Nella figura accanto si vede come si comporta la JVM nel caso in cui si richiede l' implementazione relativa ad un determinato Provider di SHA-256.

NOTA: Esistono dei Provider di default all' interno della JDK ed è possibile implementarli con una extend della classe Provider. Oppure è possibile scaricare le classi Provider da terze parti ed aggiungerle nel progetto.



Java Criptography Architecture

Il Framework che mette a disposizione le funzionalità crittografiche più comuni all' interno di Java è la JCA.

Questo framework è un insieme di API Provider-based, ed alcune di esse sono:

- Algoritmi di Hashing (ex. SHA-256)
- Algoritmi di Firma Digitale (ex. DSA)
- Cifratura Simmetrica e Cifratura di stream (ex. 3DES)
- Cifratura Asimmetrica (ex. RSA)
- Cifratura Password-Based (ex. PBKDF2)
- Algoritmi basati sulle curve ellittiche (ex. ECDHA)
- Scambio di chiavi (ex. DH)
- Generatore di chiavi
- Message Authentication Codes (ex. HMAC)
- Generatore di numeri casuali

Management di certificati e chiavi

Per implementare le funzioni relative alla Public Key Infrastructure Java mette a disposizione una struttura su file, chiamata keystore, che permette lo storage di coppie di chiavi pubbliche/private e relativi certificati e una struttura chiamata truststore che permette il salvataggio di certificati fidati.
Di default il truststore di sistema è il file cacerts che contiene tutti i certificati dei top level CA globali ed è presente in ogni sistema operativo.
Queste strutture sono manipolabili con il tool di Java keytool e sono convertibili in formati compatibili ad openssl.

Questi file verranno utilizzati quando è necessaria l' autenticazione dei peers tramite lo scambio dei certificati, ad esempio in una comunicazione TLS.

Il formato dei keystore e dei truststore è definito nello standard PKCS#12 che è quello di Default, ma è possibile richiedere l'istanza di strutture di diverso formato come JKS o JCEKS, ormai deprecati.

Java Secure Socket Extension

Invece di lasciare allo sviluppatore lo sviluppo di Socket sicuri tramite le funzioni fornite dalla JCA, Java ha creato una API che fornisce una interfaccia semplice per accedere a Socket che implementano i protocolli di sicurezza più comuni come SSL, TLS e DTLS chiamati SSLSocket.

SSLSocket è una sottoclasse di Socket e permette di utilizzare le stesse interfacce semplici di Socket.

E' possibile configurare la SSLSocket utilizzando dei KeyManager e dei TrustManager, le classi che permettono di leggere keystore e truststore, per dichiarare con quali chiavi avviene la firma e con quali certificati avviene la verifica della firma.

La JDK include Provider che implementano i seguenti protocolli:

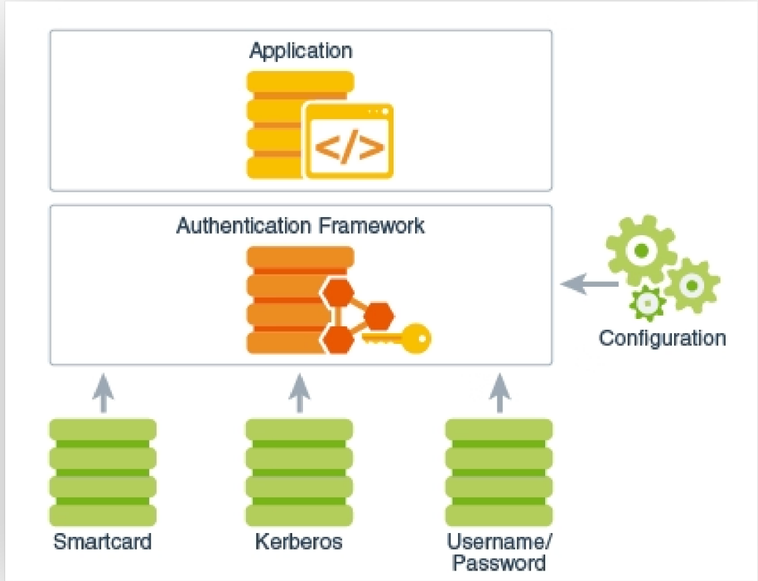
- SSL 3.0
- TLSv1.2
- TLSv1.3
- DTLSv1.2
- ecc….

Java Authentication Service

La piattaforma di Java espone delle API che permettono alle applicazioni di autenticare l'utente attraverso dei moduli di login.

Le applicazioni possono chiamare la classe LoginContext che legge il file di configurazione che specifica le classi LoginModule da chiamare (esse contengono l' implementazione del servizio di autenticazione).

Le classi LoginModule sono state concepite con il principio di indipendenza e quindi non è necessario scrivere l' implementazione ma è possibile scaricarle da terze parti o fare riferimento a quelle già messe a disposizione dalla JDK. (ex. JndiLoginModule)



2-way TLS con Autenticazione e Autorizzazione

Nelle prossime slide presento un modo per implementare una connessione TLS dove sia il Client che il Server vengono autenticati (2-way) tramite l'utilizzo di certificati.

I certificati per verificare la firma e le chiavi utilizzate per firmare dovranno essere presenti all' interno dei keystore e truststore relativi ai due peer. Questo verrà fatto utilizzando il tool keytool di Java.

Per stabilire una connessione TLS (v1.2) tra i due peer utilizzerò le API fornite dalla JSSE, che implementano di già le funzioni crittografiche esposte dalla JCA.

Tramite il framework JAAS richiederò al client di fornire username e password per far autenticare l'utente.

In base ai permessi assegnati all' utente autenticato il server invierà al client le risorse che può visualizzare.

Ovviamente i messaggi scambiati saranno cifrati dal canale sicuro instaurato.

Wireshark verrà utilizzato per accertarsi che la cifratura sia stata effettuata e verrà utilizzato a scopo di troubleshooting.

Keytool per generazione chiavi e certificati

Ogni peer dovrà generare una coppia di chiavi da inserire nel keystore, ed esportare il certificato relativo alla chiave pubblica. Infine dovrà inserire nel truststore il certificato dell' altro peer per assicurarsi che il certificato che poi verrà ricevuto matchi quello dei certificati fidati.

1. Generazione coppia di chiavi RSA a 2048bit valide 90 giorni inserite in KEYSTORE protetto da PASS:

   keytool -genkeypair -alias ALIAS -keyalg RSA -keysize 2048 -validity 90 -storepass PASS -keystore KEYSTORE
2. Esportazione certificato da KEYSTORE e salvataggio su file cert.crt

   keytool -exportcert -alias ALIAS -storepass PASS -keystore KEYSTORE -rfc -file cert.crt
3. Importazione certificato cert.crt in TRUSTSTORE

   keytool -importcert -alias ALIAS -storepass PASS -keystore TRUSTSTORE -file cert.crt

SSLContext nel server

SSLContext viene utilizzato per ricavare le SSLServerSocket dopo la accept() una volta che un client si connette.

SSLContext viene inizializzato con dei KeyManagers, con dei TrustManagers e con un numero casuale.

I KeyManagers vengono ricavati dal file s_keystore.p12 e i TrustManagers vengono ricavati dal file s_truststore.p12. Essi permettono di ottenere le chiavi per la generazione della firma ed i certificati per la verifica della firma.

NOTA: Le entry del keystore sono protette da password perchè sono informazioni sensibili, mentre le entry del truststore non sono protette.

```
SSLContext ctx;
KeyManagerFactory kmf;
TrustManagerFactory tmf;
KeyStore ks;
KeyStore ts;
char[] passphrase = "passphrase".toCharArray();
ctx = SSLContext.getInstance( protocol: "TLS");
kmf = KeyManagerFactory.getInstance( algorithm: "SunX509");
tmf = TrustManagerFactory.getInstance( algorithm: "SunX509");
ks = KeyStore.getInstance( type: "PKCS12",  provider: "SunJSSE");
ks.load(new FileInputStream( name: "s_keystore.p12"), passphrase);
kmf.init(ks, passphrase);
ts = KeyStore.getInstance( type: "PKCS12",  provider: "SunJSSE");
ts.load(new FileInputStream( name: "s_truststore.p12"), passphrase);
tmf.init(ts);
ctx.init(kmf.getKeyManagers(), tmf.getTrustManagers(),  random: null);
ssf = ctx.getServerSocketFactory();
```

SSLContext nel client

Nel client la situazione è analoga. Ovviamente i TrustManager e i KeyManager devono essere configurati con i keystore e truststore del client.

Per forzare la connessione ad utilizzare TLSv1.2 (in quanto di default si utilizza TLSv1.3 per motivi di troubleshooting con wireshark) si usa:

```
ctx = SSLContext.getInstance("TLSv1.2");
```

Per avviare la connessione TLS sul client si usa:

```
SSLSocket socket = (SSLSocket) factory.createSocket(("IP_SERV",PORT_SERV));
socket.startHandshake();
```

E nel server la connessione verrà accettata da:

```
socket = server.accept();
```

A questo punto i socket possono essere utilizzati normalmente come se fossero della classe Socket. Quindi con readLine() e println().

Autenticazione

La classe LoginContext viene creata quando la connessione TLS è già instaurata e prende in input due parametri:

1. Il nome della entry nel file di configurazione che contiene le classi LoginModule da caricare.
2. Una classe che implementa CallbackHandler per recuperare i dati richiesti dai LoginModule dall' utente.

Ad esempio:

```
lc = new LoginContext("Sample", new MyCallBackHandler());
lc.login();
```

Ciò che è necessario che lo sviluppatore implementi è :

- la classe che implementa l'interfaccia CallbackHandler
- la classe che crea il LoginContext
- il file di configurazione

La classe che crea il LoginContext è la classe che è responsabile di comunicare all' utente l' esito dell' autenticazione. Essa può utilizzare ad esempio out.println() sul PrintWriter ricavato dal socket.

Classe che implementa CallbackHandler

La classe che implementa CallbackHandler deve implementare il metodo handle() con parametro: Callback[] callbacks.

Le callback che sono inserite in questo array sono inserite dal LoginModule.

Il compito del programmatore è di inserire all' interno di ogni callback l'informazione richiesta dal LoginModule così da poter autenticare l' utente.

Nella figura accanto si vede come nome e password vengono richieste attraverso la socket ed inserite nella rispettiva callback con i metodi setName() e setPassword().

```
} else if (callback instanceof NameCallback) {
    NameCallback nc = (NameCallback) callback;
    outputsocket.println(nc.getPrompt());
    outputsocket.flush();
    nc.setName(inputsocket.readLine());
} else if (callback instanceof PasswordCallback) {
    PasswordCallback pc = (PasswordCallback) callback;
    outputsocket.println(pc.getPrompt());
    outputsocket.flush();
    pc.setPassword(inputsocket.readLine().toCharArray());
} else {
```

File di configurazione

Il File di configurazione contiene tutte le entry che è possibile specificare quando un LoginContext viene istanziato.

Ad esempio nella figura è possibile vedere che la classe SampleLoginModule viene istanziata con priorità required (è necessario che l'utente passi questo modulo di Login per autenticarsi) nel caso in cui il LoginContext sia istanziato con Entry "Sample".

Ovviamente il LoginModule in questo caso è una classe creata dallo sviluppatore ma è possibile inserire LoginModule presenti nella JDK di default come:

JndiLoginModule

oppure scaricando le classi da terze parti.

```
Sample {
  SampleLoginModule required debug=true;
};
```

Autorizzazione

Una volta che l' utente è riuscito ad autenticarsi (lc.login() non ha alzato la LoginException()) è possibile ottenere una nuova istanza della classe Subject() con i relativi Principals, settati dal LoginContext, con:

Subject subject = lc.getSubject();

Subject è la classe che rappresenta i soggetti in uno spazio di autenticazione. Essi possono rappresentare persone o servizi.

Una volta che un soggetto è autenticato il relativo Subject viene popolato con delle identità associate chiamate Principals, grazie alle quali è possibile scrivere delle Policy che permettono o negano l'accesso alle risorse.

Inoltre ogni Subject ha associati attributi di sicurezza chiamati Credentials, e spaziano dalle chiavi di cifratura fino ad arrivare alle password hashate.

Java utilizza i metodi doAs() e doAsPrivileged() per far eseguire delle azioni alla JVM impersonificandosi come un soggetto.

Sia soggetto che azione sono passati come parametri ai 2 metodi tramite le classi PrivilegedAction e Subject.

Policy File

Quando usiamo il comando java per caricare le classi compilate si deve passare il policy file che verrà utilizzato per assegnare i permessi ai diversi Principals.

Il comando è:

-Djava.security.policy==sampleazn.policy

Il contenuto del file dovrà:

- specificare l' archivio Jar della classe Action
- specificare la classe che implementa Principal
- specificare il Principal che avrà i permessi inseriti
- specificare la lista di permessi

Nella figura si nota che il Principal "readFileUser" della classe SamplePrincipal ha assegnati i permessi per leggere: java.home, user.home e il file foo.txt quando esegue la classe SampleAction dentro l'archivio jar.

```
grant codebase "file:./SampleAction.jar", Principal SamplePrincipal "readFileUser" {
    permission java.util.PropertyPermission "java.home", "read";
    permission java.util.PropertyPermission "user.home", "read";
    permission java.io.FilePermission "foo.txt", "read";
};
```

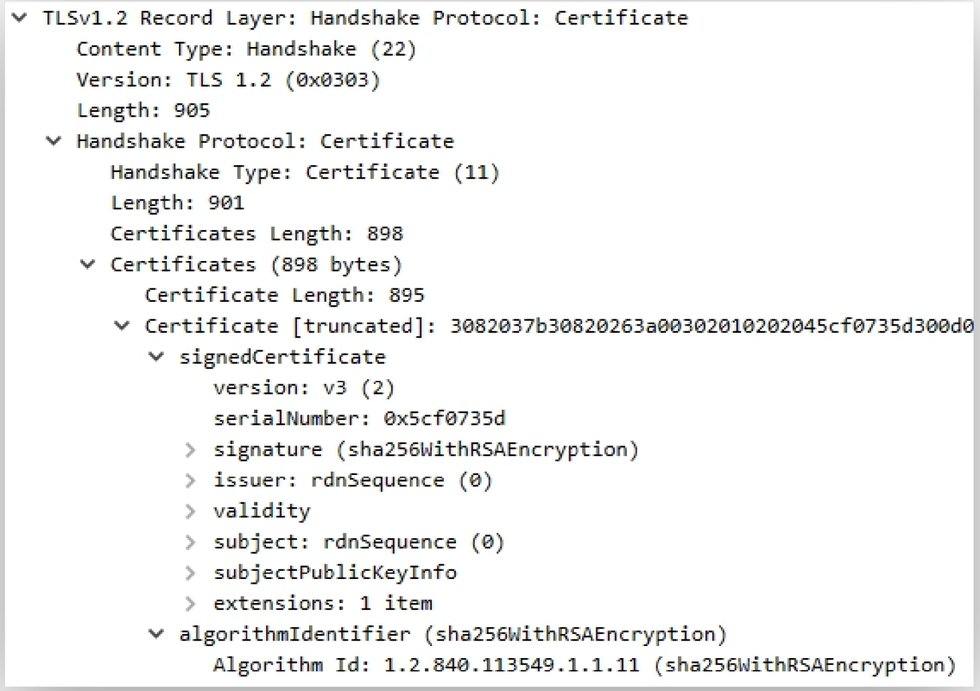
Check con Wireshark

Con Wireshark è possibile controllare se la connessione TLS viene instaurata correttamente e se i certificati scambiati tra client e server appartengono ai keystore ed ai truststore relativi.

Nella figura accanto è possibile controllare che nel pacchetto Certificate inviato dal server al client è presente il campo signedCertificate con un serialNumber.

Confrontiamo il numero seriale con quello presente nel truststore tramite keytool e verifichiamo il match.

Inoltre è importante verificare che tutti i messaggi scambiati dopo l'handshake siano criptati → lo sono perchè i pacchetti vengono marcati come Application Data.



Fine

Pitch

Want to make a presentation like this one?

Start with a fully customizable template, create a beautiful deck in minutes, then easily share it with anyone.

Create a presentation (It’s free)