Technische Universität München
Lehrstuhl Informatik VIII
Prof. Dr.-Ing. Georg Carle
Dr. Heiko Niedermayer
Dipl.-Inform. Ralph Holz

Network Security WS13/14
Assignment 3

Submission by **Wed, 15 January 2014, 23:00 UTC (24:00 CET)**.

General remark:

- This is a considerably more intensive assignment. We have thus increased the credits you can get.
- Task 2 is relatively complex and relatively-new. We may have to put additional information, hints or even corrections on the homepage, so please watch your mail and the homepage!

Important directions:

- Submit by SVN. **Always** indicate your team: place a file `team.exercise02.txt` in **both** user directories under `exercise02/`. E.g. in `s_gu27jaf/exercise02/team.exercise02.txt` **and** in `s_guab38/exercise02/team.exercise02.txt`.
- **Always** show how you arrived at your answer to get full credits. I.e. show the computation, refer to the output, the man page, the RFC, etc. – whatever it takes to make your answer understandable.
- Be sure you do not violate our plagiarism guidelines.
- Submit solutions for T-credits in a **PDF**. Submit **well-documented code**. We accept code in Python and Java. For details, see below.

Task 1 Breaking a crypto protocol (7 T-credits)

The goal of this task is to have you analyse and extend a cryptographic protocol.

Alice, Bob and Mallory are students of Computer Science at the prestigious Terrific University of Madagascar (TUM). At the beginning of the semester, they have all securely pair-wise exchanged their public keys. While Alice and Bob have become good friends, Mallory is secretly very jealous of Alice and only pretends to be friends with the two.
One day, Alice and Bob meet for a coffee at the end of class. Bob is really happy and tells Alice: *I have designed a new authentication protocol. It's really good, look!*

1. B chooses a nonce N_B and sends it to A, explicitly indicating sender and receiver:
   $B \rightarrow A : B, A, N_B$
2. A responds with a nonce N_A and a signature:
   $A \rightarrow B : A, B, N_A, N_B, \; Sig_A(N_A, N_B, B)$

3. B accepts and replies with a new nonce N'_B:
   $B \rightarrow A : B, A, N'_B, N_A, Sig_B(N'_B, N_A, A)$

Bob continues: *This ensures the following. When the protocol is complete,*

1. B can be sure that A created message 2 specifically as a response to B's first message. Thus, it must be A with whom B has executed the protocol!

2. The other way around, A can be sure that she is communicating with B because only B can create the signature in the third message!

Alice knows that authentication protocols can be vulnerable in very subtle ways. She takes a good long look at the protocol and then declares: *I am afraid it's broken. An attacker can inject messages such that A would falsely assume she has run the protocol with B, while in fact she was talking to the attacker.*
Bob is down-hearted, so Alice takes pity and explains to him why the protocol is vulnerable. Can you do the same?
Use the following attacker model: Assume that Mallory (M) can control all messages in the network, i. e. read, delete, modify etc. She is only limited by the cryptographic functions, which we assume to be perfect. She does not know any party's private keys (except her own).

a) **(3 T-credits)** Show that the **authentication** is broken, as Alice claims. Do this by giving a sequence of message exchanges that conform to the protocol specification yet constitute a violation of Bob's second claim. (Note: **write down the full message exchange, not just your changes!**)

b) **(1 T-credit)** State precisely which field in which protocol message causes the vulnerability, and why. Change the thus identified field so the authentication property is not violated anymore. Give the new protocol flow.

c) **(1 T-credit)** The following is a variant of Bob's protocol that adds a weak kind of key establishment:

1. $B \rightarrow A : B, A, N_B, K_{pub,A}(K_B)$

2. $A \rightarrow B : A, B, N_A, N_B, Sig_A(N_A, N_B, B), K_{pub,B}(K_A)$

3. $B \rightarrow A : B, A, N'_B, N_A, Sig_B(N'_B, N_A, A)$

The shared key is then derived as $(K_A||K_B)$ (i.e. concatenation). Explain why the key establishment does not meet the criteria for Perfect Forward Secrecy (PFS).

d) **(1 T-credit)** Show how to enable PFS. Write down the new message flow.

e) **(1 T-credit)** We said the key establishment is weak in Bob's version (we do not mean the lack of PFS here). That is because there is a hidden vulnerability in there. Which one? (Say why!)

Task 2 Writing miniSSL (30 P-credits)

We will implement a simplified version of SSL/TLS plus a simple application-layer protocol in this task. The goal is to get a deeper understanding of the SSL/TLS protocol. Solutions are accepted in both Python (must run under Python 2.7) and Java (must run under JDK 7). Both of them must of course run under our provided virtual machine.

miniSSL and miniGET

miniSSL is a bare-bone version of SSL. We disregard the complete SSL/TLS record layer. Instead, miniSSL conducts a simplified SSL/TLS handshake, which leads to authentication (optionally: mutual) and establishment of a session key for encryption and a session key for the HMAC. We do not negotiate ciphers nor MAC algorithms: we always use AES-CBC-128 and HMAC-SHA1. We do not implement Diffie-Hellman key exchange, nor compression. However, we do use RSA and implement optional client authentication (i.e., mutual authentication). Our derivation of the session keys is simplified compared to the real SSL/TLS.
miniGET is simply a `GET filename` operation sent from client to server after the miniSSL handshake. The server sends the requested file using the session keys for encryption and MAC protection. Once the server has stopped sending, both client and server terminate without further signalling.

Protocol flow

The simplified protocol you are going to implement is shown in the following. C is the client, S is the server. There is one valid CA, which we call miniSSL-CA.
We begin with the miniSSL handshake. Note that we group the different message types of SSL/TLS in new, custom ones. In the following, the comma operator indicates the border between message fields, and the `|` operator indicates concatenation of two (bit) strings.

1. C chooses a nonce n_c of length 28B. It chooses the cipher suite to be AES-CBC-128 and HMAC-SHA1 as the MAC function, represented as a string `AES-CBC-128-HMAC-SHA1`. The type of the first message is `ClientInit`. C sends this to S:

   $C \rightarrow S$: `ClientInit`, $\texttt{n}_\texttt{c}$, `AES-CBC-128-HMAC-SHA1`

2. Upon receiving this message, S chooses a random nonce n_s of length 28B. It acknowledges the client's cipher choice and also sends a certificate. It can optionally add a request for the client to authenticate with a certificate by sending the `CertRequest` string:
   $S \rightarrow C$: `ServerInit`, $\texttt{n}_\texttt{s}$, `AES-CBC-128-HMAC-SHA1`, $\texttt{Cert}_\texttt{s}$ [, `CertReq`]

3. C verifies that the server certificate has been issued by the miniSSL-CA (checking the signature, using certificates in its root store), that the certificate is not expired, and that the Common Name is the expected one. It extracts S's public key. It generates a pre-master secret p as a random value of length 46B. From this, it derives two session keys k_1 and k_2 as k_1=`HMAC-SHA1`$(p, n_c|n_s|\texttt{enc})$ and k_2=`HMAC-SHA1`$(p, n_c|n_s|\texttt{mac})$, with `enc,mac` being the binary strings `00000000` and `11111111`. Finally, it computes a HMAC over all message contents up to this point in the following way:
   m_c=`HMAC-SHA1`(k_2,`ClientInit`$|\ \texttt{n}_\texttt{c}|$`AES-CBC-128-HMAC-SHA1`$|$`ServerInit`$|\texttt{n}_\texttt{s}|\texttt{Cert}_\texttt{s}$ [$|$`CertReq`])

   C sends the following to the server (note the optional client certificate). E means encryption with the respective public key:
   $C \rightarrow S$: `ClientKex`, $E_S(p), m_c$[, $\texttt{Cert}_\texttt{C}, \texttt{Sig}_\texttt{C}(\texttt{n}_\texttt{s}|\texttt{E}_\texttt{S}(\texttt{p}))$]

   $Sig_C(n_s)$ is C's signature on the server's nonce.

4. The server, upon receiving this message, also verifies that $Cert_C$ was issued by miniSSL-CA (see above) and that the certificate is not expired. There is no need to check the Common Name. It computes k_1, k_2 and verifies that m_c has the correct value. It computes a HMAC over all message contents up to this point in the following way:
   m_s=`HMAC-SHA1`(k_2,`ClientKex`$|E_S(p)|m_c|[,$`Cert`$_C$]).

   It sends this to C:
   $S \rightarrow C : m_s$

5. C verifies m_s. The handshake is complete. C will now initiate `miniGET`. This protocol uses k_1 for encryption and k_2 for HMACs.

High-level implementation instructions (MUSTs)

The following are MUSTs for your implementation.

- Your implementation must allow the protocol to run between client and server on different hosts, with different IP addresses. Thus, you must use TCP/IP sockets.
- Your implementation must conduct the above handshake and derive the two session keys, one for encryption, the other HMACing.
- Your implementation must be multi-session capable, i.e. it must be possible to have two clients communicating with the server at the same time.
- Client and server must internally track the state of the current handshake, i.e. do session management. All cryptographic information that pertains to a given protocol (nonces, keys, etc.) run must be tracked and stored internally!
- Your client must verify that the server certificate has been issued from the minissl CA, carries the expected Common Name (it is in the cert), and is not expired. No other checks are necessary. It must use the public key found in the server certificate.
- Your server must support two modes: Simple and ClientAuth. In Simple, the client does not need to authenticate to the server. In ClientAuth, the server verifies that the client presents a certificate from the minissl CA and is not expired.
- After the handshake, the client must download the text file `payload.txt` from the server. Generate this file yourself. This transfer must be encrypted and HMAC-secured with the negotiated keys. In ClientAuth mode, the server must only allow this if the client has presented the correct certificate. You are free to implement your own application-layer protocol for the download signalling. We suggest to implement a simple GET and have both sides simply terminate the connection when the file is received.
- Using the tool `sha1sum`, you must prove that client and server have now identical copies of `payload.txt`.
- When the server or client presents the wrong certificate, the other party must terminate the connection (close the socket). There are rogue certificates in the ZIP file. Use them to verify that you do not accidentally accept the wrong certificates.
- Your code must not crash when a message is unexpected or malformed. Rather, client and server must gracefully terminate the connection in such cases.

Using the implementation (MUSTs)

These are MUSTs. The client must be started like this:

```
./client.py dst_ip dst_port clientcert clientprivkey
or java Client dst_ip dst_port clientcert clientprivkey
```

The server must be started like this:

```
./server.py listen_port servercert serverprivkey {SimpleAuth, ClientAuth} payload.txt
or java Server listen_port servercert serverprivkey {SimpleAuth, ClientAuth} payload.txt
```

I.e. server and client read in their certificates and private keys from file. These are provided.

Implementation help and suggestions

Sockets Here is some reading on how to use sockets in Python: http://docs.python.org/2/howto/sockets.html.
A Java tutorial can be found here: http://docs.oracle.com/javase/tutorial/networking/sockets/index.html

Libraries Under Python, we found that the use of three libraries yields best results (this tells you a lot about crypto for Python):

- pycrypto for simple RSA, AES and HMAC operations, https://www.dlitz.net/software/pycrypto/api/current/
- m2crypto for direct verification of a certificate, http://www.heikkitoivonen.net/m2crypto/api/
- pyopenssl for reading X.509 fields, http://packages.python.org/pyOpenSSL/

On our virtual machine these are already pre-installed. If you use your own machine, you can install them on by doing a `sudo apt-get install python-crypto python-m2crypto python-openssl` (But don't forget to test your code on the VM).
As having to use three crypto libs is awkward, we have coded up some help for you in `keyutils.py`. You'll mostly only need PyCrypto.
Java provides a relatively good crypto API, too – but while you may submit Java code, please note that we are not experts in that area and cannot offer support: http://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html.

Degrees of freedom Apart from the above MUSTs, you are free to choose all implementation details. Refer to RFC4346 for ideas. In particular, you are free to choose:

- Protocol field format and field delimiters
- Appropriate encodings for message types and fixed strings, if needed
- The way you do RSA encryption (e.g., padding). Find a way to deal with the problem of plaintexts that are too long.
- You may choose a higher key length, but then you need to choose the string indicating that.
- Blocking or non-blocking sockets

- Details of the file transfer
- We suggest you use ports > 1024 to avoid needing administrative rights

Grading factors

The following factors will count into the credits for this Task.

- Correctness of code.
- Completion of correct handshake and correct downloads.
- Adherence to MUSTs as outlined above. In particular, complete session tracking.
- Well-commented code.