CHALMERS UNIVERSITY OF TECHNOLOGY
Dept of Computing Sciences

Home assignment 1, Cryptography course

The assignment consists of two largely independent parts. In the first (and main) part you will study a well-known attack on an SSL channel and answer some questions. In the second part, you will encrypt your solution using `gpg` before submitting it.

Part A. An attack against SSL.

0. Introduction

In this part we will explore a relatively recent (2003) attack on widely used cryptographic software, discovered by Serge Vaudenay and co-workers at EPFL in Switzerland and reported in [1]. The attack is against an SSL/TLS channel; one example instantiation could be to find the password used by an email client to get email from an IMAP mail server. In reported experiments, the attack could recover the password of a user in less than an hour.

The attack is trivial to prevent by a simple change in the SSL implementation and the OpenSSL software implements this change from version 0.9.7a, February 2003 (see [2]). This fix was implemented before [1] was announced (thanks to communication between the discoverers and the OpenSSL developers), so the attack was never significant in practice, but it is anyhow interesting.

In this document we will describe the attack in some detail; at some points you will find questions that you are supposed to answer in your submission. The questions are generally simple when you have understood the explanations up to that point. Thus they serve mainly to check your understanding of what is presented.

Note added October 21, 2010: Variants of this attack has been subject to a lot of publicity in the last few months, under the catchy name of *Padding Oracle Attacks.* This is due to the discovery by Rizzo and Duong that many web frameworks (ASP.NET, JavaServer Faces, Ruby on Rails, OWASP ESAPI) are vulnerable to the attack. We describe this in more detail in the Appendix to this assignment. The appendix is independent of the assignment as such, but we hope that it provides an interesting case study.

1. Overview of SSL

SSL (Secure Socket Layer) is a security protocol that runs below application-layer protocols like HTTP or IMAP, but on top of transport-layer protocols such as TCP. It provides both confidentiality and data integrity, thus providing a secure channel to the communicating applications. An overview of how the protocol operates is given in the course textbook.

For the purposes of this assignment it is only necessary to know that when a connection is established, secret keys are exchanged between the parties using public-key techniques. The parties also agree on which algorithms to use for secret-key encryption and Message Authentication Code (MAC) computations.

A Message Authentication Code (MAC) is a short piece of information used to authenticate a message; in other words, to confirm that the message came from the stated sender (its authenticity) and has not been changed in transit (its integrity). When the connection

has been established, subsequent communication during the session is encrypted using the agreed methods and keys.
When a message MES is to be sent, first its MAC is computed, using the agreed method. Then the MAC is appended to MES and padding PAD added so that the full message MES || MAC || PAD makes up a whole number of blocks. The padded message is then encrypted using a block cipher in CBC mode. The IV for encryption is not sent as part of the ciphertext but is agreed on in other ways, not detailed here.
The protocol is typically used in a client/server setting. On the server side we will find e.g. a web server or a mail server. The attack will be performed against the server by an active adversary, who intercepts client messages, modifies them and sends the modified message to the server. It is thus a chosen ciphertext attack.

The server, on receipt of a ciphertext, performs the following steps:

1. The message is decrypted, using the CBC decryption algorithm.
2. The padding is checked to be correct and removed.
3. The MAC is checked to be correct and removed.
4. The remaining message MES is handed over to the application.

If either of the checks in steps 2 or 3 fails, an error message is sent (over the secure channel, i.e. MACed, PADded and encrypted) and the session is aborted. This behaviour is typical for cryptographic protocols: any failed check is an indication of an attack and the protocol should be immediately aborted.

2. Padding

In this assignment, for concreteness we make the assumption that the block size is 64 bits = 8 bytes (as is the case e.g. for 3DES).
We will need to be specific about how padding is done. Let the length in bytes of MES || MAC be n. The padding then consists of $8 - (n \bmod 8)$ bytes, each with the value $7 - (n \bmod 8)$ (as an eight-bit integer). So if n is a multiple of 8, padding consists of eight bytes, each with value 7 ($= 00000111_2$).
We now introduce some notation: With capital letter variables (S,C,...) we will always refer to blocks. We will need to describe blocks also as sequences of eight bytes, for which we will use the notation $< b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8 >$.
The last block of a full message, i.e. the one that contains the padding, we will call the *pad block*. The pad block thus has one of the following eight forms, where the ?:s represent the last bytes of the unpadded message:

$<?, ?, ?, ?, ?, ?, ?, 0 >$
$<?, ?, ?, ?, ?, ?, 1, 1 >$
$<?, ?, ?, ?, ?, 2, 2, 2 >$
$<?, ?, ?, ?, 3, 3, 3, 3 >$
$<?, ?, ?, 4, 4, 4, 4, 4 >$
$<?, ?, 5, 5, 5, 5, 5, 5 >$
$<?, 6, 6, 6, 6, 6, 6, 6 >$
$< 7, 7, 7, 7, 7, 7, 7, 7 >$

(Remark: SSL actually allows longer padding than necessary, in order to hide message length, but we will ignore that.)

We will later construct random blocks and need to have an idea about the probability that a random block is actually a valid pad block. So, let's for a moment just consider randomly constructed blocks of 64 bits. Note that in the next three questions we look just at randomly constructed blocks and some probabilities when such blocks are chosen; you do not need to think about the wider context to answer these questions.

Question 1: What is the probability that such a random block (uniform distribution over all 64 bit blocks) is a pad block of type 0 (i.e., of the form in the first line above)?

Question 2: What is the probability that a random block is a valid pad block (of any of the above forms)?

Question 3: What is the *conditional probability* that a random block is of type 0, given that it is a valid pad block? To avoid unnecessary resubmissions, we give the answer: around 99.6 %. You must anyhow answer, showing how this is computed.
Hint: check Bayes Theorem (Wikipedia)

We will later make use of these facts; when a random block is a pad block, we will assume that it is of type 0, since the probability for this is very high.

3. A side channel attack

In the attack to be described, the attacker will intercept a ciphertext in transit and modify it, adding some random data before forwarding it to the server.
When the modified message is received, it is processed as described above. It is almost certain that this will lead to abortion; even if the padding check succeeds, it is extremely unlikely that the MAC check will succeed. The attacker will not be able to comprehend the error message, since it is encrypted. Instead the attack is based on measuring the time elapsed before the session aborts. Checking the padding is a quick process, while recomputing a MAC for a lengthy message takes noticeably longer time. The attacker thus uses a *side channel*; he cannot break encryption but gets useful information by other means. Other forms of side channels that have been exploited in attacks are e.g. variations in power consumption for small devices.
We can now see the simple fix that prevents the attack: just perform the MAC check even if the padding check fails! A better solution, which requires a change to the protocol, would be to apply the MAC *after* padding and encryption. Then all the attacker's modified messages will have an invalid MAC and decryption will never be done and thus padding never checked.

4. The IMAP protocol

As mentioned above, one special case of the attack could be to find the password of a user, checking mail from an email server running IMAP (Internet Mail Access Protocol). We do not need to discuss the IMAP protocol in detail. It is enough to know that a common set-up is that the client that wants to check mail opens an SSL channel to the server. After channel establishment at the SSL level, the first application-level command from the client is

XXXX `LOGIN "`*username*`" "`*password*`"`

where *XXXX* is a message sequence number and *username* and *password* are replaced by actual data. Of course, the message is MAC'ed, padded and encrypted as above by the SSL channel.
The important properties here are

- A session starting with this command is initiated several times per hour by the client, sometimes as frequent as once per minute. In some cases multiple mailboxes in the client lead to independent logins, resulting in dozens or even hundreds of sessions per hour.
- This login message is the same every time for a given user except for the sequence number. In particular, the password is at a fixed position in the message.

The attack proceeds by passive listening to the channel establishment of each session followed by active interception of the login message, which is modified before it is forwarded to the server. This leads to session abortion, but also to more timing information for the adversary. After many sessions (in the order of 1000), the password will be recovered, as detailed in the next section.
For the duration of the attack, the mail client will try to get mail many times, each time with successful channel establishment followed by immediate abortion after the login message. It is interesting that some common mail clients accept this without alerting the user. Of course, if the user is present at her machine, she might notice that no new mails arrive, so an attack undisturbed overnight could be more convenient.

5. An attack against CBC mode

Finally, we are ready to describe the heart of the attack, which is really an attack against block cipher encryption in CBC mode.
We consider the following scenario:

- A sender repeatedly sends messages that contain the same, fixed plaintext block P of critical importance at the same, fixed position in every message. Other parts of the messages may differ.
- Messages are MAC-ed, padded and encrypted using a block cipher with block size 64 bits in CBC mode. Different messages may be encrypted and MAC'ed with different keys (in the mail example, each session has different keys).
- The adversary can intercept and modify messages.
- The adversary can distinguish whether messages he sends have invalid padding or invalid MAC, for example by measuring time differences in server response. The probability of both valid padding and valid MAC is negligible, so that case is ignored.

Most of the discussion so far has tried to convey to the reader that such a scenario is quite realistic. We just note that in the mail example, the password may cross block boundaries, so that two blocks will be of interest, but we ignore that here.
For simplicity, let us assume that the critical block is the third block in the message, so that a cleartext message has the form $??P?\ldots$, where the question-marks stand for uninteresting blocks. The corresponding ciphertext has the form $?C'C?\ldots$, where again ? blocks are uninteresting. Both C' and C are important, however, since CBC encryption means that $C = E(P \oplus C')$, where E is the block cipher encryption function.
In order to gain information about P, the adversary constructs blocks of the form $R_i =< r_1, r_2, r_3, r_4, r_5, r_6, r_7, i >$, where all the r_k are random bytes and i is systematically changed by the adversary. The messages sent to the server are $(R_i \oplus C')||C$.
The server first decrypts the message; the last block (the pad block) will then be decrypted to be $D(C) \oplus (R_i \oplus C')$, which turns out to be $P \oplus R_i$.

Question 4: Show the calculations that verify this fact.

Now, the server checks the padding. If padding is correct, we assume that the block is a pad block of type 0, i.e. $P \oplus R_i$ has last byte 0.

Question 5: What is then the last byte of P?

If padding is not correct, the adversary was unlucky and will have to wait for the next message, where he will try another value of i. After at most 255 messages, he will have determined the last byte of P.

Question 6: The message sent is only two blocks long, so MAC computation does not take noticeable time. In the real attack the message is made considerably longer in a way that does not affect the above reasoning, but will slow down the MAC computation. Suggest (with motivation) a way to do this. There is considerable freedom here, so nothing clever is needed, but you must make a concrete suggestion.

After the last byte of P has been decided to be, say, b_8, the attack proceeds to find the second to last byte. To this end, the adversary constructs blocks of the form $S_i =< r_1, r_2, r_3, r_4, r_5, r_6, i, b_8 \oplus 1 >$, where again the r_i are random. The message sent is $(S_i \oplus C')||C$.

Question 7: If padding is correct, which is byte b_7 of P?

In this way, after at most a further 255 messages, b_7 will be determined and the attack proceeds to find b_6.

Question 8: Describe the form of messages used to discover b_6.

Proceeding similarly, the whole block P can be found in at most $8 \cdot 255 = 2040$ messages, or in ca 1000 messages on average.
Using dictionary lookup techniques, trying values of i in order of frequency, this bound on the number of messages can be considerably tightened. (Just by restricting to printable ASCII characters, disregarding frequencies among these, the average number of messages will be reduced to around 400.)
We end with a voluntary question, which you do not need to answer to pass the assignment.

Question 9: We assumed above, when trying to find the last byte, that a correct pad block had type 0. This is wrong in 0.4 % of the cases. Suggest a way to improve the technique and actually find the type of pad block, at the expense of further messages (but also finding more bytes of P).

Does this refinement apply also to finding the other bytes of P?

Submission

You must answer the questions above and submit your answers. The answers are preferrably written as a text file (use the ordinary addition symbol + instead of $\oplus$).
The resulting file should be encrypted for the course as described in part B below before submission.

References

1. Brice Canvel, Alain Hiltgen, Serge Vaudenay, Martin Vuagnoux: Password Interception in a SSL/TLS Channel, proceedings of CRYPTO'03.

2. `http://www.openssl.org/news/secadv_20030219.txt`.

Part B. Setting up GPG keys.

0. Introduction

To give you some practice in using cryptographic techniques, you will use gpg to encrypt your submissions. Here we will give hands-on instructions on how to do this. Explanations of the concepts involved and what gpg actually does will be given during the rest of the course. gpg is installed on the Linux machines. If you use your home computer, you may download it from http://www.gnupg.org.

1. Using gpg

We describe here only what is necessary to encrypt your solution to the assignment. More information on gpg is available in the Gnu Privacy Handbook, which is linked from the course web site.

- The first step is to download the course public key file from the course homepage. The key is available there in ASCII form and you download it to your account using your web browser. We assume that you store it under the same name, pubkey.asc.
- Next you need to import the course public key to your keyring:

  ```
  > gpg --import pubkey.asc
  ```

  You will get some confirmation information, including the name of the key holder, the cryptography course. You can now delete pubkey.asc; it will be managed by gpg in the .gnupg directory.
- Next you should check the fingerprint of the imported key to make sure that the file has not been tampered with. gpg can do this for you. It is used as a command line tool from a shell window:

  ```
  > gpg --fingerprint crypt
  ```

  Note that you may refer to the key by just the beginning of the course email address (enough characters to uniquely identify the key in your keyring). Among a few other things this command will tell you the fingerprint of the key, namely

  ```
  AA15 2691 AD43 3FF6 0E8F  6ABD 39F6 C391 B54E EEE2
  ```

  We will discuss the meaning of fingerprints later in the course. Any difference is a sure sign that you have downloaded a faulty file, meaning our system is under attack. Of course, such an attack would need also to tamper with this document (changing the output example above), the course website and with the checksum given close to the link to the key file on the same site. If you consider this unlikely, you may just compare your output to the one above.
- As an optional step, you may now sign the course public key, indicating that you believe that this is actually the course key. If you choose to do this, you type

  ```
  > gpg --sign-key crypt
  ```

This will start a dialogue where you have to choose a trust level.

If you choose to omit this step, you will later, when you use the course key for encryption, be asked to confirm that you really want to use an unsigned key.

- When you have completed your answers to part A and have a file, say, `sol2.txt`, you encrypt it using `gpg`:

  ```
  > gpg --encrypt --recipient cryp -a sol2.txt
  ```

  Note that you must tell `gpg` for whom you are encrypting. Note also that if you did not sign the course public key, you will now be asked if you really trust the key.

  The resulting file `sol2.txt.asc` should be uploaded as your submission to this assignment.

If you wish, you may practice by encrypting files for yourself and decrypting them, exchanging keys with fellow students and sending each other encrypted files etc. For this, you need to create a key pair for yourself. For completeness, we give here instructions on how to do this, but emphasize that *this is not a part of the assignment.*
To generate a key pair for yourself, you use the command

```
> gpg --gen-key
```

This will start a dialogue, where you have to select options and give name and email address. Accept the default options (ElGamal encryption and DSA signatures, 2048 bits key length). The meaning of these choices will be explained next week. Choose a suitable lifetime for the key (e.g. 3 months). Give name, email address and a pass phrase (which you MUST remember). Key generation will take some time, since random data may have to be generated.
To send your public key to other users, so that they can encrypt files for you, you must export the key from `gpg` to an ASCII file:

```
> gpg --export -a > pubkey.asc
```

The `-a` option (only one hyphen!) gives an ASCII file. Here we redirect output to the file `pubkey.asc`; the file now contains your public key. You may choose another name, but stick to the conventional `.asc` suffix.
If you use GPG on a regular basis e.g. for encrypting email, you should be careful with signing the public keys of your correspondents; you should only do so after verifying the fingerprint with the key owner. The problem is illustrated in the cartoon *Responsible behavior* from xkcd.com:

HEY, I JUST GOT HOME FROM THE PARTY
THE ONE WITH THE IRC FOLKS?
YEAH.
HOW WAS IT?
GOT TOO DRUNK. I SCREWED UP, BAD.
WHAT HAPPENED?
THERE WAS A GIRL. NO IDEA WHO SHE WAS. DON'T EVEN KNOW HER NAME. I WAS TOO DRUNK TO CARE.
AND WHAT, YOU SLEPT WITH HER?
NO. I SIGNED HER PUBLIC KEY.
SHIT, MAN.

Appendix. Recent developments (added October 2010)

Note: This appendix should be read on screen, since it contains several links to blog posts, YouTube videos etc.

The attack described in this assignment was studied in several papers in the last decade, for example looking at the vulnerability of other padding methods and the development of clever statistical methods to discover also small timing differences over networks. But interest has exploded in the last few months, after Juliano Rizzo and Thai Duong found that the attack is effective against a wide range of web applications.

Web applications, running in a browser, use the HTTP protocol, which is stateless. This means that there is no notion of a session with several interactions between client and server in the protocol. However, typical applications do need such a notion, where some session state is developed in a sequence of interactions. Since the protocol is stateless, such a state must be preserved by other means, either at the server or the client side (or both). Typically, the server lets the client maintain the state. The server thus sends the complete state information to the client side with each response, either in a cookie or in a hidden field in the HTML document. A hidden field (with HTML tag `<hidden>`) is not displayed by the browser; instead the application is set up so that this hidden field is included in the next request from the client. Thus the server gets back the session state in this field, processes the request using the state information, includes the updated state in the hidden field of the response, and so on.

Often, part of the state information is sensitive and should not be revealed to the client or to eavesdroppers; thus a common procedure is that the server encrypts the state and includes the encrypted value in the hidden field. A minor, inessential complication is that the HTML file is a text file, so the encrypted binary value must be converted to ASCII, typically by BASE64 encoding. Note that the client does not decrypt anything; the client just returns the encrypted value in its next request.

Assume now that the state is encrypted with a block cipher in CBC mode. This requires some method of padding, and for simplicity we assume here that padding is done as in SSL. In this scenario, everything is set up for an attacker to do the same attack as in the assignment and decrypt the state, by iteratively modifying the encrypted string in a series of requests. For the attack to succeed, it is vital that the attacker gets to know in which cases the padding is correct. Unlike our previous situation, where this was inferred by the attacker using timing information, the web scenario is often much more favourable for the attacker. It turns out that many web applications give explicit help by replying with an error page saying `Illegal padding` or something similar when padding is invalid. When padding is OK, a corrupt state results after decryption, but typically a response is computed that can be distinguished from the `Illegal padding` case. It is then a simple matter for the attacker to proceed and decrypt the entire state. Further, a typical web server will not be disturbed by receiving a long sequence of requests in a very short time. Rizzo and Duong developed a tool, POET (Padding Oracle Exploit Tool), which assisted in attacking web sites; typical state information could be decrypted in a matter of minutes.

Using the tool, Rizzo and Duong explored the web and found many vulnerable sites. It turns out that many popular web development frameworks produce applications with the necessary faulty behaviour. They presented their work at Black Hat Europe in April 2010 and released POET at about the same time. This was met with considerable interest, but things really exploded when they made a further presentation at the Ekoparty conference in Buenos Aires in September, where they pointed out that many (most?) ASP.NET applications are vulnerable to the same attack.

A brief summary of the development of the events is as follows:

- – September 17: Rumors on various blogs that Rizzo and Duong had found ASP.NET vulnerable to the attack and that they would describe this at Ekoparty.

- Friday, September 17, 16.50 – 17.40 Argentinian time: Conference presentation. To make things more interesting, they presented and demonstrated how to combine this attack with other known vulnerabilities to often gain complete control of the server with administrator rights. The demo was at that time already available on YouTube.

- September 17, unknown time: Microsoft releases Security Advisory 2416728, *Vulnerability in ASP.NET Could Allow Information Disclosure*, which describes the vulnerability. Several Microsoft security related blogs immediately post entries pointing to the Advisory.

- September 17, 19.55 (unknown time zone): As one example of these posts, we pick the one on the Microsoft blog *Security Research & Defense*. The title is Understanding the ASP.NET Vulnerability, and it starts by saying "Our *recent* advisory describes an ASP.NET vulnerability which was *recently* publicly disclosed" (my emphasis). The post goes on to describe a workaround (use the same error page for all errors). A script to apply the workaround is included.

- September 18, early morning: Scott Guthrie, Microsoft VP and head of ASP.NET development, publishes a post on his ScottGu's Blog with title Important: ASP.NET Security Vulnerability. The post starts

  > A few hours ago we released a Microsoft Security Advisory about a security vulnerability in ASP.NET. This vulnerability exists in all versions of ASP.NET.
  > This vulnerability was publically disclosed late Friday at a security conference. We recommend that all customers immediately apply a workaround (described below) to prevent attackers from using this vulnerability against your ASP.NET applications.

  It then goes on to describe various variants of the workaround, with patch code included.

- September 20: A post on the Microsoft SharePoint Team Blog announces that the SharePoint suite of products are vulnerable to the attack and describes workarounds. On the same day, ScottGu publishes a FAQ on the vulnerability and the Microsoft Security Response Center blog informs about an update to the Security Advisory, "as we've begun to see limited attacks with the ASP.NET vulnerability".

- September 24: New ScottGu post explaining that "we are actively working on releasing a security update that fix the issues, and our teams have been working around the clock to develop and test a fix that is ready for broad distribution across all Windows platforms via Windows Update. I'll post details about this once it is available." Further workaround steps are described, to be added to the previous ones.

- September 27: POET developers post a new YouTube video claiming to show that Microsoft's workarounds do not prevent the attack.

- September 28: Microsoft releases Security Bulletin MS10-070 - Important. The bulletin announces an update that "resolves a publicly disclosed vulnerability in ASP.NET. [. . .] This security update is rated Important for all supported editions of ASP.NET except Microsoft .NET Framework 1.0 Service Pack 3".

  The solution is that the server signs the encrypted state. This provides authentication and prevents the attacker from tampering with the ciphertext.

- September 28: ScottGu informs about the availability of the update and adds among the FAQ:s the following:

  Q: Do I Really Need to Apply this Update?

  A: Yes. This update fixes security vulnerabilities that are publically known. You must install this update patch to be safe.

During this period of ten days numerous blogs discuss the vulnerability, but we stop here. By googling, you can also find more videos on YouTube demonstrating attacks and more tools to assist in exploiting the vulnerability.

We finish instead by noting that the different forms of attack help us see the essence of it. The crucial property of the target is that it acts as a padding oracle: it tells the attacker whether his submitted ciphertext becomes correctly padded plaintext after decryption or not. So, the target gives away just one bit of information (a yes/no answer), but this is enough for the attacker when repeated questions are possible. We also see that the presence of the long MAC computation in the original attack just plays the role of creating an observable difference between valid and invalid padding, thus making the target a padding oracle. In the web version this is instead achieved by the distinct error pages. Finally, we point out again that the real problem in the target behaviour is to accept unauthenticated ciphertexts and provide a (partial, one-bit) decryption service.

Final remark (added January 2011): To emphasize even more the impact of repeatedly providing just a single bit of information, we note that a variant of this attack makes it possible for the attacker to *encrypt an arbitrary message* with the server's secret key, without knowing this key. Thus, the attacker can create an arbitrary state. How this is done is explained by solving problem 7 in the exam from December, 2010 (see course web site).