Handout 1 (Security Engineering)

Much of the material and inspiration in this module is taken from the works of Bruce Schneier, Ross Anderson and Alex Halderman. I think they are the world experts in the area of security engineering. I especially like that they argue that a security engineer requires a certain *security mindset*. Bruce Schneier for example writes:

> *"Security engineers — at least the good ones — see the world differently. They can't walk into a store without noticing how they might shoplift. They can't use a computer without wondering about the security vulnerabilities. They can't vote without trying to figure out how to vote twice. They just can't help it."*
>
> *"Security engineering…requires you to think differently. You need to figure out not how something works, but how something can be made to not work. You have to imagine an intelligent and malicious adversary inside your system …, constantly trying new ways to subvert it. You have to consider all the ways your system can fail, most of them having nothing to do with the design itself. You have to look at everything backwards, upside down, and sideways. You have to think like an alien."*

In this module I like to teach you this security mindset. This might be a mindset that you think is very foreign to you—after all we are all good citizens and not hack into things. I beg to differ: You have this mindset already when in school you were thinking, at least hypothetically, about ways in which you can cheat in an exam (whether it is about hiding notes or looking over the shoulders of your fellow pupils). Right? To defend a system, you need to have this kind mindset and be able to think like an attacker. This will include understanding techniques that can be used to compromise security and privacy in systems. This will many times result in insights where well-intended security mechanisms made a system actually less secure.

Warning! However, don't be evil! Using those techniques in the real world may violate the law or King's rules, and it may be unethical. Under some circumstances, even probing for weaknesses of a system may result in severe penalties, up to and including expulsion, fines and jail time. Acting lawfully and ethically is your responsibility. Ethics requires you to refrain from doing harm. Always respect privacy and rights of others. Do not tamper with any of King's systems. If you try out a technique, always make doubly sure you are working in a safe environment so that you cannot cause any harm, not even accidentally. Don't be evil. Be an ethical hacker.

In this lecture I want to make you familiar with the security mindset and dispel the myth that encryption is the answer to all security problems (it is certainly often part of an answer, but almost always never a sufficient one). This is actually an important thread going through the whole course: We will assume that encryption works perfectly, but still attack "things". By "works perfectly" we mean that we will assume encryption is a black box and, for example, will not look at the underlying mathematics and break the algorithms.[1]

For a secure system, it seems, four requirements need to come together: First a security policy (what is supposed to be achieved?); second a mechanism (cipher, access controls, tamper resistance etc); third the assurance we obtain from the mechanism (the amount of reliance we can put on the mechanism) and finally the incentives (the motive that the people guarding and maintaining the system have to do their job properly, and also the motive that the attackers have to try to defeat your policy). The last point is often overlooked, but plays an important role. To illustrate this lets look at an example.

The questions is whether the Chip-and-PIN system with credit cards is more secure than the older method of signing receipts at the till. On first glance, Chip-and PIN seems obviously more secure and this was also the central plank in the "marketing speak" of the banks behind Chip-and-PIN. The earlier system was based on a magnetic stripe or a mechanical imprint on the card and required customers to sign receipts at the till whenever they bought something. This signature authorises the transactions. Although in use for a long time, this system had some crucial security flaws, including making clones of credit cards and forging signatures. Chip-and-PIN, as the name suggests, relies on data being stored on a chip on the card and a PIN number for authorisation.

Although the banks involved trumpeted their system as being secure and indeed fraud rates initially went down, security researchers were not convinced (especially the group around Ross Anderson). To begin with, the Chip-and-PIN system introduced a "new player" that needed to be trusted: the PIN terminals and their manufacturers. Of course it was claimed that these terminals are tamper-resistant, but needless to say this was a weak link in the system, which criminals successfully attacked. Some terminals were even so skilfully manipulated that they transmitted PIN numbers via a built-in mobile phone connection. To mitigate this security flaw, you need to vet quite closely the supply chain of such terminals—something that also needs to be done in other industries.

Later on, Ross Anderson and his group managed to launch man-in-the-middle attacks against Chip-and-PIN. Essentially they made the terminal think the correct PIN was entered and the card think that a signature was used. This flaw was mitigated by requiring that a link between the card and the bank is established at every time the card is used. Even later this group found another problem with Chip-and-PIN and ATMs which do not generate random enough numbers (nonces) on which the security of the underlying protocols relies.

The problem with all this is that the banks who introduced Chip-and-PIN managed to shift the liability for any fraud and the burden of proof onto the customer with the new system. In the old system, the banks had to prove that the customer used the card, which they often did not bother about. In effect if fraud occurred the customers were either refunded fully or lost only a small amount of money. This taking-responsibility-of-potential-fraud was part of the

[1] Though fascinating this might be.

“business plan” of the banks and did not reduce their profits too much. Since they successfully claimed that their Chip-and-PIN system is secure, banks were able to point the finger at the customer when fraud occurred: it must have been the fault of the customer, who must have been negligent loosing the PIN. The customer had almost no means to defend themselves in such situations. That is why the work of *ethical* hackers like Ross Anderson’s group was so important, because they and others established that the bank’s claim, their system is secure and it must have been the customer’s fault, was bogus. In 2009 for example the law changed the burden of proof back to the banks whether it was really the customer who used a card or not.

It is a classic example where a security design principle was violated: The one who is in the position to improve security, also needs to bear the financial losses if things go wrong. Otherwise, you end up with an insecure system. In case of the Chip-and-PIN system, no good security engineer would actually think that it is secure: the specification of the EMV protocol (underlying Chip-and-PIN) is some 700 pages long, but still leaves out many things (like how to implement a good random number generator). Moreover, banks can add their own sub-protocols to it. With all the experience we already have, it is as clear as day that criminals were able to poke holes into it. With how the system was set up, the banks had no incentive to come up with a system that is really secure. Getting the incentives right in favour of security is often a tricky business.

Of Cookies and Salts

Lets look at another example which helps us to understand how passwords should be verified and stored. Imagine you need to develop a web-application that has the feature of recording how many times a customer visits a page. For example to give a discount whenever the customer visited a webpage some x number of times (say x equal 5). For a number of years the webpage of the New York Times operated in this way: it allowed you to read ten articles per months for free; if you wanted to read more you had to pay. There is one more constraint: we want to store the information about the number of times a customer has visited inside a cookie.

A typical web-application works as follows: The browser sends a GET request for a particular page to a server. The server answers is request. A simple JavaScript program that realises a “hello world” webpage is as follows:

```
var express = require('express');
var app = express();

app.get('/', function(request, response){
    response.write('Hello World');
    response.end()
});

```

```
 9  // starting the server
10  app.listen(8000);
```

The interesting lines are 4 to 7 where the answer to the GET request is generated…in this case it is just a simple string. This program is run on the server and will be run whenever a browser initiates such a GET request.

For our web-application of interest is the feature that the server when answering the request can store some information on the client. This information is called a *cookie*. The next time the browser makes another GET request to the same webpage, this cookie can be read by the browser. Therefore we can use a cookie in order to store a counter recording the number of times a webpage has been visited. This can be realised with the following small program

```
 1  var express = require('express');
 2  var cookie  = require('cookie-parser')
 3
 4  var app = express();
 5  app.use(cookie());
 6
 7  app.get('/', function(req, res){
 8      var counter =  parseInt(req.cookies.counter) || 0;
 9      res.cookie('counter', counter + 1);
10      if (counter >= 5) {
11          res.write('You are a valued customer ' +
12                    'visting the site ' + counter + ' times.');
13      } else {
14          res.write('This is visit number '+ counter +'!');
15      }
16      res.end();
17  });
18
19  app.listen(8000);
```

The overall structure of this code is the same as the earlier program: Lines 7 to 17 generate the answer to a GET-request. The new part is in Line 8 where we read the cookie called `counter`. If present, this cookie will be send together with the GET-request from the client. The value of this counter will come in form of a string, therefore we use the function `parseInt` in order to transform it into a string. In case the cookie is not present, or has been deleted, we default the counter to zero. The odd looking construction `...|| 0` is realising this in JavaScript. In Line 9 we increase the counter by one and store it back to the client (under the name `counter`, since potentially more than one value could be stored). In Lines 10 to 15 we test whether this counter is greater or equal than 5 and send accordingly a message back to the client.

Let us step back and analyse this program from a security perspective. We store a counter in plain text on the client's browser (which is not under our control at all). Depending on this value we want to unlock a resource (like a discount) when it reaches a threshold. If the client deletes the cookie, then the counter will just be reset to zero. This does not bother us, because the purported discount will just be granted later. This does not lose us any (hypothetical) money. What we need to be concerned about is when a client artificially increases this counter without having visited our web-page. This is actually a trivial task for a knowledgeable person, since there are convenient tools that allow us to set a cookie to an arbitrary value, for example above our threshold for the discount.

There is no real way to prevent this kind of tampering with cookies, because the whole purpose of cookies is that they are stored on the client's side, which from the the server's perspective is in a potentially hostile environment. What we need to ensure is the integrity of this counter in this hostile environment. We could think of encrypting the counter. But this has two drawbacks to do with the key for encryption. If you use a 'global' key for all our client's that visit our site, then we risk that our whole "business" might colapse when this key gets known to the outside world. Suddenly all cookies we might have set in the past, can now be manipulated. If on the other hand, we use a "private" key for every client, then we have to solve the problem of having to securely store this key on our server side (obviously we cannot store the key with the client because then the client again has all data to tamper with the counter; and obviously we also cannot encrypt the key, lest we can solve a chicken-and-egg problem). So encryption seems to not solve the problem we face with the integrity of our counter.

Fortunately, *hash function* seem to be more suitable for our purpose. Like encryption, hash functions scrambles data but in such a way that it is easy to calculate the output of a has function from the input. But it is hard (i.e. practically impossible) to calculate the input from knowing the output. Therefore has functions are often called one-way functions. There are several such hashing function. For example SHA-1 would has the string `"hello world"` to

```
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
```

Another handy feature of hash functions is that if the input changes a little bit, the output changes drastically. For example `"iello world"` produces under SHA-1 the output

```
d2b1402d84e8bcef5ae18f828e43e7065b841ff1
```

That means it is not predictable what the output will be from input that is "close by".

We can use hashes and store in the cookie the value of the counter together with its hash. We need to store both pieces of data such we can extract both components (below I will just separate them using a `"-"`). If we now read back the cookie when the client visits our webpage, we can extract the counter, hash it again and compare the result to the stored hash value inside the cookie. If these hashes disagree, then we can deduce that cookie has been tampered with. Unfortunately if they agree, we can still not be entirely sure that not a clever hacker has tampered with the cookie. The reason is that the hacker can see the clear text part of the cookie, say `3`, and its hash. It does not take much trial and error to find out that we used the SHA-1 hashing functions and then graft a cookie accordingly. This is eased by the fact that for SHA-1 many strings and corresponding hashvalues are precalculated. Type into Google for example the hash value for `"hello wolrd"` and you will actually pretty quickly find that it was generated by `"hello wolrd"`. This defeats the purpose of a hashing functions and would not help us for our web-applications. The corresponding attack is called *dictionary attack*…hashes are not reversed by brute force calculations, that is trying out all possible combinations.

There is one ingredient missing, which happens to be called *salt*. The salt is a random key, which is added to the counter before the hash is calculated. In our case we need to keep the salt secrete.

Note ....NYT

```javascript
var express = require('express');
var cookie  = require('cookie-parser')
var crypto = require('crypto');

var app = express();
app.use(cookie());

function mk_hash(s) {
    return crypto.createHash('sha1').update(s).digest('hex')
}

function mk_cookie(c) {
    return c.toString() + "-" + mk_hash(c.toString())
}

function gt_cookie(s) {
    var splits = s.split("-", 2);
    var counter = parseInt(splits[0])
    if (mk_hash(counter.toString()) == splits[1]) {
  return counter
    } else {
  return 0
    }
}


app.get('/', function(req, res){
    var counter = gt_cookie(req.cookies.counter) || 0;
    res.cookie('counter', mk_cookie(counter + 1));
    if (counter >= 5) {
  res.write('You are a valued customer ' +
      'visting the site ' + counter + ' times.');
    } else {
  res.write('This is visit number '+ counter +'!');
    }
    res.end();
});

// starting the server
app.listen(8000);
console.log("Server running at http://127.0.0.1:8000/");
```