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."*

and

> *"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 do not hack into things. However, 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 by hiding notes or by looking over the shoulders of your fellow pupils). Right? To defend a system, you need to have this kind of 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 a 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 let's look at an example.

Chip-and-PIN is Surely More Secure, No?

The questions is whether the Chip-and-PIN system used with modern credit cards is more secure than the older method of signing receipts at the till? On first glance the answer seems obvious: Chip-and-PIN must be more secure and indeed improved security was 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 cards and required customers to sign receipts at the till whenever they bought something. This signature authorised 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. Even though the banks involved trumpeted their system as being absolutely secure and indeed fraud rates initially went down, security researchers were not convinced (especially not the group around Ross Anderson).[2] To begin with, the Chip-and-PIN system introduced a "new player" into the system that needed to be trusted: the PIN terminals and their manufacturers. It was claimed that these terminals were 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 skimmed PIN numbers via built-in mobile phone connections. To mitigate this flaw in the security of Chip-and-PIN, you need to be able to vet quite closely the supply chain of such terminals. This is something that is mostly beyond the control of customers who need to use these terminals.

To make matters worse for Chip-and-PIN, around 2009 Ross Anderson and his group were able to perform man-in-the-middle attacks against Chip-and-PIN. Essentially they made the terminal think the correct PIN was entered and

[1] Though fascinating this might be.

[2] Actually, historical data about fraud showed that first fraud rates went up (while early problems to do with the introduction of Chip-and-PIN we exploited), then down, but recently up again (because criminals getting more familiar with the technology and how it can be exploited).

the card think that a signature was used. This is a kind of *protocol failure*. After discovery, the 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 did not generate random enough numbers (cryptographic nonces) on which the security of the underlying protocols relies.

The overarching problem with all this is that the banks who introduced Chip-and-PIN managed with the new system to shift the liability for any fraud and the burden of proof onto the customer. In the old system, the banks had to prove that the customer used the card, which they often did not bother with. 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 "business plan" of the banks and did not reduce their profits too much.

Since banks managed to successfully claim that their Chip-and-PIN system is secure, they were under the new system able to point the finger at the customer when fraud occurred: customers must have been negligent losing their PIN and customers had almost no way of defending themselves in such situations. That is why the work of *ethical* hackers like Ross Anderson's group is so important, because they and others established that the banks' claim that their system is secure and it must have been the customer's fault, was bogus. In 2009 the law changed and the burden of proof went back to the banks. They need to prove whether it was really the customer who used a card or not. The current state of affairs, however, is that standing up for your right requires you to be knowledgeable, potentially having to go to court…if not, the banks are happy to take advantage of you.

This is a classic example where a fundamental security design principle was violated: Namely, 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 dare to claim that it is secure beyond reproach: 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). No human being is able to scrutinise such a specification and ensure it contains no flaws. Moreover, banks can add their own sub-protocols to EMV. With all the experience we already have, it is as clear as day that criminals were bound to eventually be able to poke holes into it and measures need to be taken to address them. However, with how the system was set up, the banks had no real incentive to come up with a system that is really secure. Getting the incentives right in favour of security is often a tricky business. From a customer point of view, the Chip-and-PIN system was much less secure than the old signature-based method. The customer could now lose significant amounts of money.

If you want to watch an entertaining talk about attacking Chip-and-PIN cards, then this talk from the 2014 Chaos Computer Club conference is for you:

`https://goo.gl/zuwVHb`

They claim that they are able to clone Chip-and-PINs cards such that they get all data that was on the Magstripe, except for three digits (the CVV number). Remember, Chip-and-PIN cards were introduced exactly for preventing this. Ross Anderson also talked about his research at the BlackHat Conference in 2014:

https://www.youtube.com/watch?v=ET0MFkRorbo

An article about reverse-engineering a PIN-number skimmer is at

https://trustfoundry.net/reverse-engineering-a-discovered-atm-skimmer/

including a scary video of how a PIN-pad overlay is installed by some crooks.

Of Cookies and Salts

Let us look at another example which will help with understanding 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 in order to give a discount whenever the customer has visited a webpage some x number of times (say x equals 5). There is one more constraint: we want to store the information about the number of visits as a cookie on the browser. I think, for a number of years the webpage of the New York Times operated in this way: it allowed you to read ten articles per month for free; if you wanted to read more, you had to pay. My best guess is that it used cookies for recording how many times their pages was visited, because if I switched browsers I could easily circumvent the restriction about ten articles.[3]

To implement our web-application it is good to look under the hood what happens when a webpage is displayed in a browser. A typical web-application works as follows: The browser sends a GET request for a particular page to a server. The server answers this request with a webpage in HTML (for our purposes we can ignore the details about HTML). A simple JavaScript program that realises a server answering with a "Hello World" webpage is as follows:

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function(request, response){
5      response.write('Hello World');
6      response.end()
7  });
8
9  // starting the server
10 app.listen(8000);
```

[3]Another online media that works in this way is the Times Higher Education http://www.timeshighereducation.co.uk. It also seems to use cookies to restrict the number of free articles to five.

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 executed whenever a browser initiates such a GET request. You can run this program on your computer and then direct a browser to the address `localhost:8000` in order to simulate a request over the internet. You are encouraged to try this out…theory is always good, but practice is better.

For our web-application of interest is the feature that the server when answering the request can store some information on the client's side. This information is called a *cookie*. The next time the browser makes another GET request to the same webpage, this cookie can be read again by the server. We can use cookies in order to store a counter that records the number of times our 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 program is the same as the earlier one: 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 an integer. In case the cookie is not present, we default the counter to zero. The odd looking construction `...|| 0` is realising this defaulting 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 specially grafted message back to the client.

Let us step back and analyse this program from a security point of view. We store a counter in plain text on the client's browser (which is not under our con-

trol). 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 not be granted. In this way we do not lose any (hypothetical) money. What we need to be concerned about is, however, 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 one to set a cookie to an arbitrary value, for example above our threshold for the discount.

There seems to be no simple 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 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 keys for encryption. If you use a single, global key for all the clients that visit our site, then we risk that our whole "business" might collapse in the event this key gets known to the outside world. Then all cookies we might have set in the past, can now be decrypted and manipulated. If, on the other hand, we use many "private" keys for the clients, 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 an impossible chicken-and-egg problem). So encryption seems to not solve the problem we face with the integrity of our counter.

Fortunately, *cryptographic hash functions* seem to be more suitable for our purpose. Like encryption, hash functions scramble data in such a way that it is easy to calculate the output of a hash function from the input. But it is hard (i.e. practically impossible) to calculate the input from knowing the output. This is often called *preimage resistance*. Cryptographic hash functions also ensure that given a message and a hash, it is computationally infeasible to find another message with the same hash. This is called *collusion resistance*. Because of these properties, hash functions are often called *one-way functions*: you cannot go back from the output to the input (without some tricks, see below).

There are several such hashing function. For example SHA-1 would hash the string `"hello world"` to produce the hash-value

```
2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
```

Another handy feature of hash functions is that if the input changes only a little, 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 just looking at input that is "close by".

We can use hashes in our web-application and store in the cookie the value of the counter in plain text but together with its hash. We need to store both pieces of data in such a way that we can extract them again later on. In the code below I will just separate them using a `"-"`. For the counter `1` for example

```
1-356a192b7913b04c54574d18c28d46e6395428ab
```

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 the 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 also its hash. It does not take much trial and error to find out that we used the SHA-1 hashing function and then the hacker can graft a cookie accordingly. This is eased by the fact that for SHA-1 many strings and corresponding hash-values are precalculated. Type, for example, into Google the hash value for `"hello world"` and you will actually pretty quickly find that it was generated by input string `"hello world"`. Similarly for the hash-value for `1`. This defeats the purpose of a hashing function and thus would not help us with our web-applications and later also not with how to store passwords properly.

There is one ingredient missing, which happens to be called *salts*. Salts are random keys, which are added to the counter before the hash is calculated. In our case we must keep the salt secret. As can be see in Figure 1, we need to extract from the cookie the counter value and its hash (Lines 19 and 20). But before hashing the counter again (Line 22) we need to add the secret salt. Similarly, when we set the new increased counter, we will need to add the salt before hashing (this is done in Line 15). Our web-application will now store cookies like

```
1 + salt - 8189effef4d4f7411f4153b13ff72546dd682c69
2 + salt - 1528375d5ceb7d71597053e6877cc570067a738f
3 + salt - d646e213d4f87e3971d9dd6d9f435840eb6a1c06
4 + salt - 5b9e85269e4461de0238a6bf463ed3f25778cbba
...
```

These hashes allow us to read and set the value of the counter, and also give us confidence that the counter has not been tampered with. This of course depends on being able to keep the salt secret. Once the salt is public, we better ignore all cookies and start setting them again with a new salt.

There is an interesting and very subtle point to note with respect to the 'New York Times' way of checking the number visits. Essentially they have their 'resource' unlocked at the beginning and lock it only when the data in the cookie states that the allowed free number of visits are up. As said before, this can be easily circumvented by just deleting the cookie or by switching the browser. This would mean the New York Times will lose revenue whenever this kind of tampering occurs. The 'quick fix' to require that a cookie must always be

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

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

var salt = 'secret key'

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

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

function gt_cookie(s) {
    var splits = s.split("-", 2);
    var counter = parseInt(splits[0])
    var hash = splits[1]
    if (mk_hash(counter.toString() + salt) == hash) {
        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/");
```

Figure 1: A Node.js web-app that sets a cookie in the client's browser for counting the number of visits to a page.

present does not work, because then this newspaper will cut off any new readers, or anyone who gets a new computer. In contrast, our web-application has the resource (discount) locked at the beginning and only unlocks it if the cookie data says so. If the cookie is deleted, well then the resource just does not get unlocked. No major harm will result to us. You can see: the same security mechanism behaves rather differently depending on whether the "resource" needs to be locked or unlocked. Apart from thinking about the difference very carefully, I do not know of any good "theory" that could help with solving such security intricacies in any other way.

How to Store Passwords Properly?

While admittedly quite silly, the simple web-application in the previous section should help with the more important question of how passwords should be verified and stored. It is unbelievable that nowadays systems still do this with passwords in plain text. The idea behind such plain-text passwords is of course that if the user typed in `foobar` as password, we need to verify whether it matches with the password that is already stored for this user in the system. Why not doing this with plain-text passwords? Unfortunately doing this verification in plain text is really a bad idea. Alas, evidence suggests it is still a widespread practice. I leave you to think about why verifying passwords in plain text is a bad idea.

Using hash functions, like in our web-application, we can do better. They allow us to not having to store passwords in plain text for verification whether a password matches or not. We can just hash the password and store the hash-value. And whenever the user types in a new password, well then we hash it again and check whether the hash-values agree. Just like in the web-application before.

Lets analyse what happens when a hacker gets hold of such a hashed password database. That is the scenario we want to defend against.[4] The hacker has then a list of user names and associated hash-values, like

```
urbanc:2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
```

For a beginner-level hacker this information is of no use. It would not work to type in the hash value instead of the password, because it will go through the hashing function again and then the resulting two hash-values will not match. One attack a hacker can try, however, is called a *brute force attack*. Essentially this means trying out exhaustively all strings

```
a, aa, ..., ba, ..., zzz, ...
```

and so on, hash them and check whether they match with the hash-values in the database. Such brute force attacks are surprisingly effective. With modern

[4] If we could assume our servers can never be broken into, then storing passwords in plain text would be no problem. The point, however, is that servers are never absolutely secure.

technology (usually GPU graphic cards), passwords of moderate length only need seconds or hours to be cracked. Well, the only defence we have against such brute force attacks is to make passwords longer and force users to use the whole spectrum of letters and keys for passwords. The hope is that this makes the search space too big for an effective brute force attack.

Unfortunately, clever hackers have another ace up their sleeves. These are called *dictionary attacks*. The idea behind dictionary attack is the observation that only few people are competent enough to use sufficiently strong passwords. Most users (at least too many) use passwords like

`123456`, `password`, `qwerty`, `letmein`, `...`

So an attacker just needs to compile a list as large as possible of such likely candidates of passwords and also compute their hash-values. The difference between a brute force attack, where maybe 2^{80} many strings need to be considered, is that a dictionary attack might get away with checking only 10 Million words (remember the language English "only" contains 600,000 words). This is a drastic simplification for attackers. Now, if the attacker knows the hash-value of a password is

`5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8`

then just a lookup in the dictionary will reveal that the plain-text password was `password`. What is good about this attack is that the dictionary can be precompiled in the "comfort of the hacker's home" before an actual attack is launched. It just needs sufficient storage space, which nowadays is pretty cheap. A hacker might in this way not be able to crack all passwords in our database, but even being able to crack 50% can be serious damage for a large company (because then you have to think about how to make users to change their old passwords—a major hassle). And hackers are very industrious in compiling these dictionaries: for example they definitely include variations like `passw0rd` and also include rules that cover cases like `passwordpassword` or `drowssap` (password reversed).[5] Historically, compiling a list for a dictionary attack is not as simple as it might seem. At the beginning only "real" dictionaries were available (like the Oxford English Dictionary), but such dictionaries are not optimised for the purpose of cracking passwords. The first real hard data about actually used passwords was obtained when a company called RockYou "lost" at the end of 2009 32 Million plain-text passwords. With this data of real-life passwords, dictionary attacks took off. Compiling such dictionaries is nowadays very easy with the help of off-the-shelf tools.

These dictionary attacks can be prevented by using salts. Remember a hacker needs to use the most likely candidates of passwords and calculate their hash-value. If we add before hashing a password a random salt, like `mPX2aq`, then

[5] Some entertaining rules for creating effective dictionaries are described in the book "Applied Cryptography" by Bruce Schneier (in case you can find it in the library), and also in the original research literature which can be accessed for free from http://www.klein.com/dvk/publications/passwd.pdf.

the string `passwordmPX2aq` will almost certainly not be in the dictionary. Like in the web-application in the previous section, a salt does not prevent us from verifying a password. We just need to add the salt whenever the password is typed in again.

There is a question whether we should use a single random salt for every password in our database. A single salt would already make dictionary attacks considerably more difficult. It turns out, however, that in case of password databases every password should get their own salt. This salt is generated at the time when the password is first set. If you look at a Unix password file you will find entries like

```
urbanc:$6$3WWbKfr1$4vblknvGr6FcDeF92R5xFn3mskfdnEn...$...
```

where the first part is the login-name, followed by a field `$6$` which specifies which hash-function is used. After that follows the salt `3WWbKfr1` and after that the hash-value that is stored for the password (which includes the salt). I leave it to you to figure out how the password verification would need to work based on this data.

There is a non-obvious benefit of using a separate salt for each password. Recall that `123456` is a popular password that is most likely used by several of your users (especially if the database contains millions of entries). If we use no salt or one global salt, all hash-values will be the same for this password. So if a hacker is in the business of cracking as many passwords as possible, then it is a good idea to concentrate on those very popular passwords. This is not possible if each password gets its own salt: since we assume the salt is generated randomly, each version of `123456` will be associated with a different hash-value. This will make the life harder for an attacker.

Note another interesting point. The web-application from the previous section was only secure when the salt was secret. In the password case, this is not needed. The salt can be public as shown above in the Unix password file where it is actually stored as part of the password entry. Knowing the salt does not give the attacker any advantage, but prevents that dictionaries can be pre-compiled. While salts do not solve every problem, they help with protecting against dictionary attacks on password files. It protects people who have the same passwords on multiple machines. But it does not protect against a focused attack against a single password and also does not make poorly chosen passwords any better. Still the moral is that you should never store passwords in plain text. Never ever.

Further Reading

A readable article by Bruce Schneier on "How Security Companies Sucker Us with Lemons"

```
http://archive.wired.com/politics/security/commentary/
    securitymatters/2007/04/securitymatters_0419
```

A recent research paper about surveillance using cookies is

http://randomwalker.info/publications/cookie-surveillance-v2.pdf

A slightly different point of view about the economies of password cracking:

http://xkcd.com/538/

If you want to know more about passwords, the book by Bruce Schneier about Applied Cryptography is recommendable, though quite expensive. There is also another expensive book about penetration testing, but the readable chapter about password attacks (Chapter 9) is free:

http://www.nostarch.com/pentesting

Even the government recently handed out some advice about passwords

http://goo.gl/dIzqMg

Here is an interesting blog-post about how a group "cracked" efficiently millions of bcrypt passwords from the Ashley Madison leak.

http://goo.gl/83Ho0N

Or the passwords from eHarmony

https://goo.gl/W63Xhw

The attack used dictionaries with up to 15 Billion entries.[6] If eHarmony had properly salted their passwords, the attack would have taken 31 years.

Clearly, passwords are a technology that comes to the end of its usefulness, because brute force attacks become more and more powerful and it is unlikely that humans get any better in remembering (securely) longer and longer passwords. The big question is which technology can replace passwords…

[6] Compare this with the full brute-force space of 62^8