Chapter 2: Identification protocols, One-way functions, and PRFs

Suggested reading:

- Security Engineering Chapters 3.1, 3.2, 3.3
- Security Engineering Chapters 5.1, 5.2.4, 5.3

Advanced reading. Note: In this class we will skip a lot of mathematical formalism around crypto. If you want to read about these concepts with full mathematical rigor, check out *A Graduate Course in Applied Cryptography* by Dan Boneh and Victor Shoup.

- GCAC 18.6 (challenge-response protocols)
- GCAC 4.4 (PRFs)

Bonus reading:

- Anatomy of a Subway Hack
- On the (In)Security of Automotive Remote Keyless Entry Systems
- Flipper Zero, a device for interacting with RFID cards

Consider one of the most essential security mechanisms: a door lock. Mechanical locks are a fascinating area with a long history, but we'll consider a contactless electronic lock for now, as widely installed at NYU and many other organizations. Your NYU ID card contains an RFID transponder that receives a signal (and electric power) from the reader (the door) and automatically responds. The door should then open.

Consider the **security policy for an RFID-powered door lock**:

1. The door should open whenever a valid card is present.
2. The door should not open without a valid card present.
    a. The above should be true even for an attacker who has observed and recorded valid cards opening the door.

Designing a security mechanism to enforce this policy is already non-trivial and requires the use of cryptography. There are actually many more elements to the security policy which we won't discuss today:

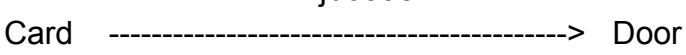
3. Given a valid card, it should not be easy to *clone* the card and produce a second working card.
4. The door should be reasonably resistant to physical force.
5. Emergency services must be able to open the door without a card.
6. It should always be possible to exit through the door in case of emergency.
7. It should not be easy to disable the door's function.
8. A card must be present to open the door. It should not be possible to open the door by relaying the signal from a valid card which is far away (a **relay attack**).
9. Administrators should be able to quickly enroll a new card.

10. Administrators should be able to quickly **revoke** a card (for example, when a student graduates).

The NYU system aims for most, but not all of these. Another possible policy is that a card should only be usable by the human being it was issued to, but this requires the use of **biometrics** which we will address much later. NYU only attempts this in certain buildings by having human guards look at the photo on the card, or having the photo appear on screen when a card is used.

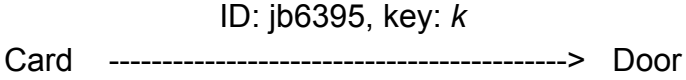
Simple protocols for opening a door

Let's consider how to achieve even the first 2 elements of the security policy above. The simplest design is for the card to transmit its NYU ID (in my case jb6395) to the door:



This is obviously not secure, because anybody who knows my ID (which is on my website) can make a fake card that transmits it and can open any door my card can.

A better design is to have my card transmit a secret value. Call it *k* (for key):



The key *k* should be unique for each user, otherwise any card can fairly easily impersonate any other card. This may seem obvious but it's been screwed up in practice: The same key was used by Volkswagen for every car manufactured for nearly 20 years from the mid-90s to mid-2010s.

In this design, the door will have to store a large table mapping each user to their key *k*. Later we'll see how we could build such a protocol with only a constant amount of storage at the door regardless of the number of users.

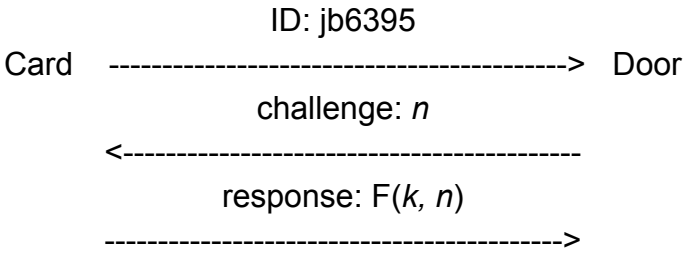
This protocol is slightly better because my key is not on my website, but it is still subject to a **replay attack**. Any attacker who records the data my card sends to the door can replay it exactly to open the door themselves. Since the data is transmitted via radio waves, it is not hard to capture.

You might think this system is “good enough” in practice because it’s not simple for most people to capture the radio transmission from a card or create a new one. But this will only get easier over time. For example, the **Flipper Zero** **device** costs less than $200 and makes it easy for anybody to record and replay communication from an RFID card.

Preventing replay attacks with challenge-response protocols

The simple protocols above are **static protocols** in that the same information is always transmitted by the card. Any static protocol will be vulnerable to replay attacks.

To prevent replay attacks, we’ll need a **dynamic protocol** where the card doesn’t always send the same information to the door. The most common approach is a **challenge-response** protocol. The door will now send the card a one-time challenge value. This number is called a **nonce**,[1] which we’ll call n:



The card will respond to this challenge by computing some function F of the nonce and its secret key. We should still assume that an **eavesdropper** has recorded the whole protocol transcript (potentially many times). What properties does F need to have to prevent this eavesdropper from later opening the door?

1. Given observations of F(k, n) and n for many values of n, it should not be feasible to compute k.
2. Given observations of F(k, n) for many values of n, it should not be feasible to compute F(k, n_*) for an unobserved nonce n_*

If our function F satisfies these two properties, then the protocol will be secure as long as the door never sends the same challenge n. This can be achieved either by choosing n randomly from a sufficiently large space (say, 128-bit values) that it will never repeat, or using a counter value for n that is always increasing. Either choice has a trade-off:

---

[1] The word “nonce” in cryptography is a shortening of n_{once}, or “number used once” Unfortunately it also has a vulgar meaning in British English, so British speakers often pronounce it as two syllables: “n-once.”

the random approach requires larger challenges and a secure source of randomness, while the counter approach requires persistent storage at the door. Both approaches are common in practice.

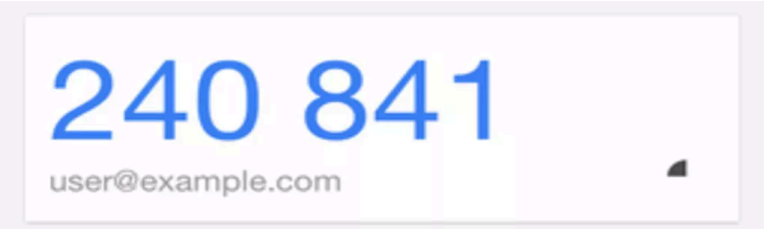
An alternate design: time-varying codes

Notice that our challenge-response protocol above has three steps instead of one. This might be reduced to two, but at least two are needed as the random challenge must be sent from the door to the card before the card's response can be sent. It's possible to design a one-step dynamic protocol if the card uses the current time *t* as the challenge:

ID: jb6395, code: F(*k*, *t*)

Card ----------> Door

This approach is sometimes called a **rolling code protocol**. It's not common for use with RFID cards like those used at NYU since they don't have their own source of power and hence cannot maintain a clock. Even if they could, you have to worry about **synchronization**, ensuring that both sides have the same clock value (or at least don't drift too far apart). But you probably have used this approach before. It's common in 2-factor authentication apps like Google Authenticator or Duo:



This approach makes more sense in this setting: mobile devices have an accurate clock and it's simpler for users to only type one code (rather than having to enter a challenge as well). Of course, synchronization is not perfect. These apps typically round the current time to the nearest 30 seconds and allow for some delay for users to type. The exact details commonly used for second-factor authentication apps are standardized in the TOTP protocol (Time-based One-time Password protocol).

It's also possible to build a similar protocol without relying on a synchronized clock, but storing a **counter** which is incremented after every successful authentication. This requires persistent storage, but no running clock. Duo and other authenticator apps implement this as well. The downside is, again, synchronization: what happens if the card attempts to authenticate, causing the counter to be incremented, but the door

never hears it due to radio interference? To provide some resiliency against lost values, usually the authentication will allow a range of values (a sliding window) to be used.

One-way functions

If you think about what properties the function F needs for a rolling code protocol to be secure, you should be able to convince yourself it's exactly the same set of properties needed for a challenge-response protocol! This is a good situation in cryptography (and computer science in general): multiple possible protocols are possible using a common shared function. Cryptographers usually start with desired properties for a function like those outlined above and then try to define a mathematical function that meets them. This is similar to starting with security policy then designing mechanisms to realize it. These functions are often called **primitives** in cryptography: the basic building blocks needed for a variety of secure protocols.

Cryptographic primitives are usually defined by an interactive **security game** between a **challenger** and an **adversary**. The challenger always follows a defined strategy, and can be thought of as a referee or judge testing if an adversary is capable of breaking the security of the primitive. The adversary, on the other hand, can follow any strategy within the rules of the game to try and violate a security assumption.

We can define our first security game to test the essential security property of a cryptographic **one-way function**, namely that an adversary cannot learn the key even after seeing many outputs of the function computed using that key:

1. The challenger randomly chooses a λ-bit key *k* for the duration of the game.
2. The adversary repeatedly chooses a nonce *n* and sends it to the challenger. For each nonce *n*, the challenger responds with F(*k, n*).
3. The adversary outputs a guess at the key *k'*
4. If *k'*=*k*, the adversary wins.

This definition has some limitations. For example, it is satisfied for a function F that outputs 0 on any input. We will give better definitions in future chapters, but this is a sufficient definition for the current example.

Key sizes, exponential and negligible functions

There are a lot of subtleties to the security game introduced above. First, how many queries can the adversary make? How much computation can they perform? Typically we assume an adversary that is **computationally bounded**. More specifically, we usually require the adversary to run in **probabilistic polynomial time (PPT)**. The

probabilistic part means the adversary can be a randomized algorithm, and polynomial time means the adversary's running time is a polynomial function of the **security parameter** which is traditionally denoted λ. In many cases it's the length of a secret key in bits. This allows us to set security arbitrarily: if we want higher security, choose a larger value of λ.

In this case, notice that the key *k* was chosen to be λ bits long. This is to prevent an attacker from winning the game by **exhaustive search** over all possible values of *k*, also called a **brute-force attack**. This would require 2^{λ} operations, which is an exponential function λ. Recall that every polynomial function of λ (such as $\lambda^4 + \lambda^3$) grows much more slowly than any exponential function of λ (like 2^{λ}). Therefore our definition rules out any attacker performing exhaustive search from "winning" the game. In practice, this also means that if we choose *k* to be big enough (say, a 128-bit random number) then brute force will be infeasible.

Second, we have to ask if the adversary is **adaptive**? That is, can they choose nonces to query based on the answers to earlier queries? Typically we allow an adaptive adversary.

As a final subtlety, won't the adversary always have some chance of guessing the correct key *k* by random chance, even if they can't try all possible keys? Indeed they will, so we can't possibly design a function F such that the attacker can *never* win the game. We instead will require that the adversary has a **negligible** chance of winning. That is, the probability of winning is a negligible function of the length of the key. Practically speaking, this means that if we choose a suitably large key it will be impossible to guess correctly at random.

More technically speaking, a **negligible function** of λ is defined as a function which, as λ tends to infinity, is smaller than the reciprocal of any polynomial function of λ.[2] The probability of exactly guessing a random λ-bit key is $2^{-\lambda}$. This function is exponentially small in λ, which is a negligible function. This notion is also sometimes called **vanishingly small**. We'll sometimes write negl(λ) to denote a negligible function of λ.

The flipside of a negligible function is an **exponential function**.

We'll revisit the notion of security parameters and bit-strength later and have much more to say about key sizes. For now we can assume we choose a standard value of λ=128,

[2] An equivalent way of defining negligible functions, using Big-Oh notation, is to say that F(λ) is negligible if F(λ) = o(1/poly(λ)) for all polynomial functions.

giving an attacker a probability 2^{-128} of correctly guessing the key, which is considered cryptographically secure.

This might seem like overkill for a door-opening protocol. But cryptographers like to build very strong cryptographic primitives so that they can be used as building blocks in higher-level protocols of various types without worry that the crypto will be vulnerable.

Back to the door-though, a one-way function is not enough! Even though it will prevent an adversary from guessing the key, the adversary doesn't need to guess the key to open the door. They only need to guess a correct response to the door's challenge.

Pseudorandom functions (PRFs)

Let's go back to the second property: the adversary shouldn't be able to guess the output of the function. This requires a stronger property than the above. We'll define a new game for it:

1. The challenger generates a random bit *b*. If *b* is 1, they generate a random key *k*.
2. The adversary repeatedly chooses a nonce *n* and sends it to the challenger.
3. If *b*=1, the challenger responds with F(*k, n*), honestly computing the PRF on the adversary's query *n*. Assume this output is λ bits long. Otherwise if *b*=0, The challenger responds with λ randomly generated bits.
4. The adversary outputs a guess *b'*.
5. If *b* is equal to *b'*, the adversary wins.

In this game the challenger picks a random bit *b* to decide if the adversary sees real or random output. We'll still assume the adversary is adaptive and PPT. But now they can win the game half the time by random guessing, so we'll need to change our security definition to say that F is a PRF if no PPT adversary can win the game with probability greater than 0.5 + negl(*λ*). We call this a **negligible advantage** over the baseline probability 0.5 of randomly guessing *b* correctly.

A function F that satisfies this definition is secure enough for the door-opening protocol. In fact, a careful reader might notice it's actually stronger than we need it to be: this definition states that an adversary can't distinguish the function's output from random, but all we need is that they can't predict the function's output exactly. We'll eventually design the precise primitive we need (a MAC function) in coming chapters.

Also note that the one-way property is *necessary*, but not *sufficient*, for building a PRF. If an adversary could win the OWF game, they could win the PRF game as well (first solve for the key, then use that knowledge to distinguish output of the PRF). We'll talk much more about how to reason about the relationship between security properties.

Random oracles and ideal functionalities

We'll talk more about how to actually build a PRF later. For now, just know that we can build PRFs which are suitable for a secure challenge-response protocol.

The name "pseudorandom function" is telling. We want a function that is "almost" a random function. Why not just use a random function? We could design a truly random function by generating and storing a function table (the mapping from every input to every output) completely at random. But storing this function table would require an impractical amount of memory. For example, consider a random 2-bit function:

input	output
00	11
01	10
10	00
11	11

Storing this function requires 8 bits (by storing only the output column, in sorted order). In general, storing an n-bit random function will require storing 2^n rows of n bits each. For a 128-bit random function, the table would have 2^{128} rows of $128=2^7$ bits, requiring a total of 2^{135} bits of storage.

Hence, it's impossible in practice to construct or use a truly random function of this size. As a tool for constructing proofs though, it is convenient to assume a magic **random oracle** exists which is capable of generating and storing the entire random function table and responding to queries for us. It has to be an oracle because we can only query it, but not read out its internal memory (which is too large to read).

Random oracles don't exist, but they serve as a model for ideal cryptographic functions. For example, it is relatively straightforward to prove that if the function F in our challenge-response protocol above is a random oracle, then the protocol is secure.

A related idea is the **ideal functionality** model, in which we write down whatever properties we want for a cryptographic function, then simply assume it exists in oracle form and prove that some protocol using it is secure.