Security & Privacy

Objectives

Because software is everywhere, it has become a prime target for attack: compromising the software behind a system is a good way to take control of the entire system. Unlike hardware and humans, software can be attacked remotely and automatically. Security is thus an important aspect of software engineering, even though it is often considered a hindrance by users. As a software engineer, you will need to keep security in mind when writing code. Unfortunately, there is no easy way to tell whether a piece of code is secure; rather, one can go through a list of well-known security problems and check whether the code is vulnerable to any of them. One common category of security problems is privacy problems: anything that can allow someone to view user data when they shouldn’t be able to, or infer facts about users based on the code’s behavior.

In this module, you will learn:

- How to keep security in mind when writing code
- How to avoid common security pitfalls

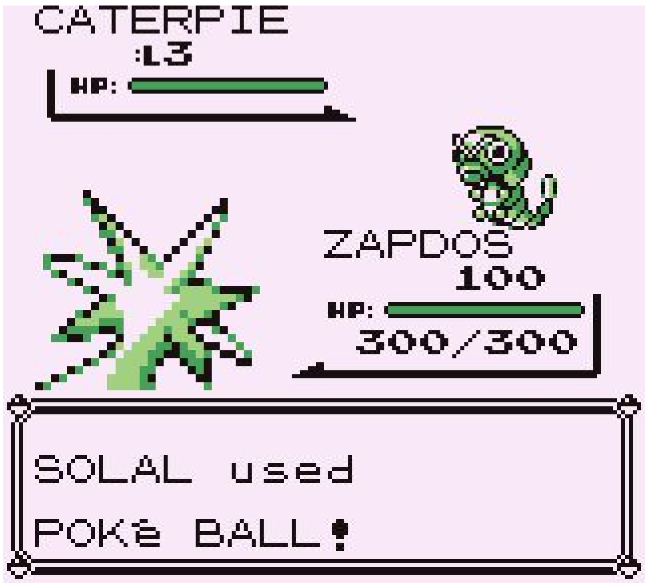
- How to preserve users' privacy

Introduction with Pokémon Red and Blue

In this section, we'll see a real-world example of a security vulnerability in two parts, from the games Pokémon Red and Blue. (This bug works in the original American releases; it was patched in most other versions)

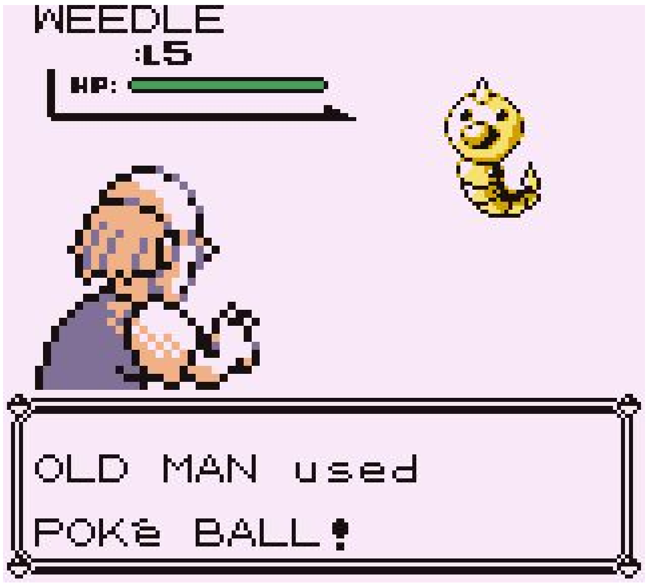
Writing to an unexpected memory location

One of the main goals in the Pokémon game is to catch all Pokémons.



SOLAL used
POKé BALL!

However, given that not everyone knows that, the game helpfully includes a tutorial. In Viridian City, the first town the player reaches after leaving their hometown, there is an old man who demonstrates Pokémon catching:



OLD MAN used
POKé BALL!

This is a nice feature to show new players what to do, especially given that Pokémon Red and Blue were the first Pokémon games (except for the Japan-only Pokémon Green).

Now let's think about how the tutorial is implemented. Pokémon Red and Blue only have one playable character, so how can the game show an old man instead? (The female playable character was added in Pokémon Crystal)

Replacing the player image is the easiest part: since there is only one possible player image in the rest of the game, the tutorial code switches the pointer to the image to point to the old man instead, then switches it back, without having to explicitly remember which image was used before since there can be only one.

But the name is another matter: it's customizable! In the first screenshot above, the player is named “SOLAL”. The tutorial code must store it somewhere. The Game Boy only has 8 KB of RAM, so declaring a variable just for that one time in the entire game seems suboptimal. So, where could the developers store it?

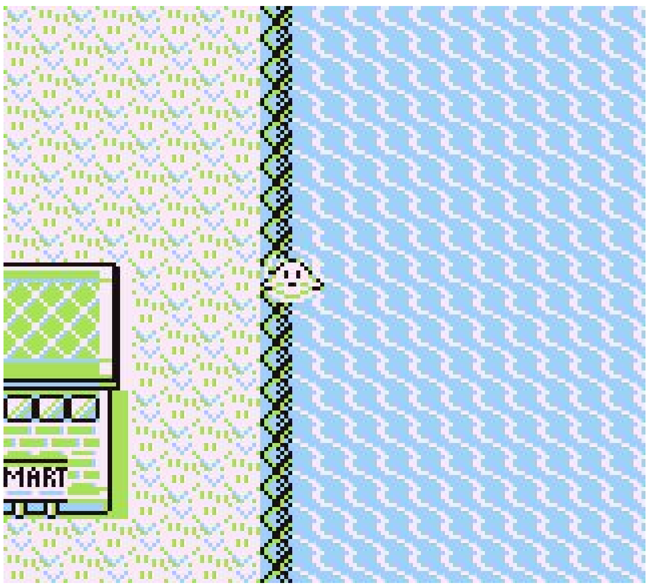
It turns out there is a convenient memory location that is otherwise unused during this part of the game: the list of wild Pokémons that can be encountered in tall grass in the current location. Every location in Pokémon has an associated list of Pokémons that can be encountered when the player is in tall grass, and similar lists for fishing and surfing on water. Whenever the player moves to a different area, the game copies the corresponding lists from the cartridge into memory. But Viridian City has no tall grass, so the corresponding Pokémon list goes unused. Why not use it? After all, its size in memory is more than enough to accommodate any player name.

This is the first part of the vulnerability: the game copies the player's name, which is user input, into an internal data structure, namely the list of Pokémons that can be found in tall grass. Worse, this input gets silently converted into a different type: from text to a list of Pokémons!

But… that's fine, right? Because this list of Pokémons is going to be overwritten whenever the player goes somewhere that does have tall grass. Right?

Reading from that location

Welcome to sunny Cinnabar Island, the town of volcanoes and Fire-type Pokémons! We hope you'll enjoy your time there, why not try surfing?



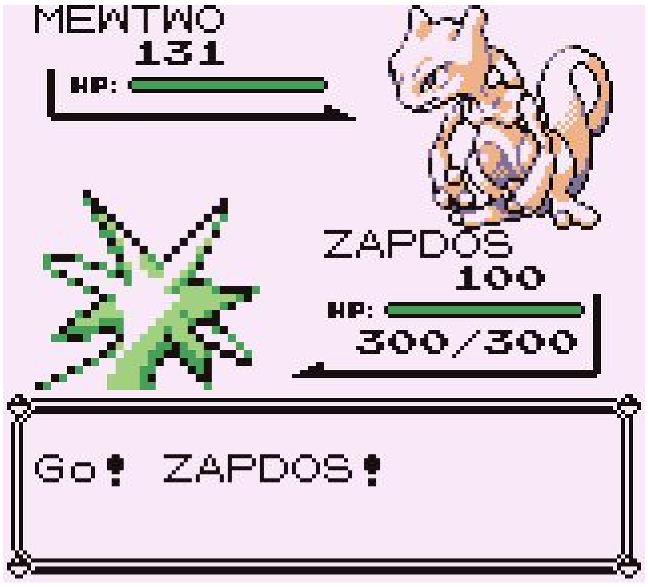
In the screenshot above, the player is surfing on the sea, by riding a Pokémon. Cinnabar Island has no grass, only sea. Why are we even talking about this? Well, it turns out the developers made a mistake: the vertical strip of water just east of the island, where the player is surfing in the screenshot, is not coded as sea into the game. It's coded as tall grass!

This is a problem, because Cinnabar Island isn't supposed to have tall grass. Thus, the game does not configure the list of Pokémons that can be encountered in tall grass when entering the area. Which means the Pokémons encountered on that strip of water are the same as the ones encountered in the tall grass wherever the player was previously.

But that’s fine, right? Because the Pokémons encountered in the strip of water are Pokémons that the player already had access to in the previous area, so the player can’t encounter Pokémons they couldn’t already encounter elsewhere. Right? ...you see where this is going.

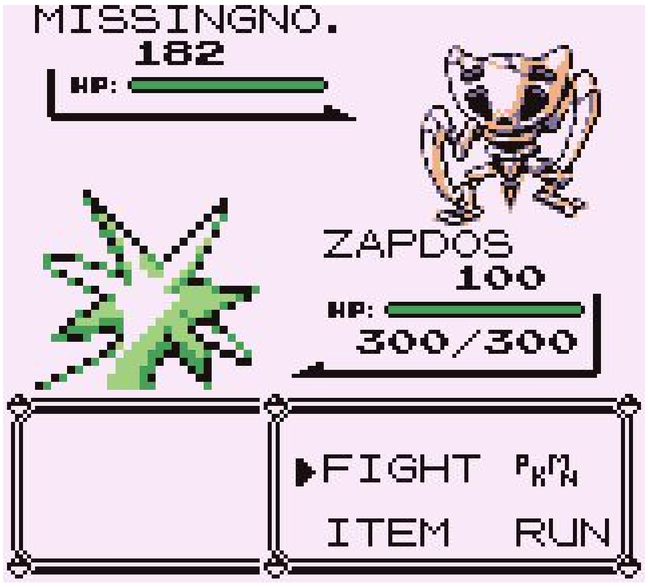
The game is accidentally reading from an uninitialized memory location, and unfortunately for the developers, that location also happens to be the subject of a bug allowing players to write arbitrary data into it.

With this bug, you too can encounter a Mewtwo, an otherwise unique Pokémon not encountered in the wild, at level 131, a full 31 levels above the maximum allowed level! Just name your player “DDDDDDD”, because the game encodes ‘D’ with a binary value that is the same as the one used to encode the ID of Mewtwo and the level 131, and the list of wild Pokémons made up of (<level>, <ID>) tuples. Then go watch the old man’s tutorial, fly directly to Cinnabar Island, surf… and *voilà*!



Go! ZAPDOS!

Since not all characters in the player name correspond to valid Pokémon, one can also get oddities, such as the “MISSINGNO.” Pokémon name, which is used when the game encounters an invalid Pokémon, along with an image that is not a Pokémon but pulled from somewhere else on the cartridge. Pokémon is not usually creepy, but set your name to ‘wwwwwww’, use this glitch, and you get this:



Takeaways

What should you take away from this interesting though mostly harmless bug?

- Avoid storing data, especially derived from user input, in unintended locations

    - If you must do it, make sure to erase the data from the location immediately after you're done
- Do not store duplicate information, such as a tile's looks (water) and its internal type (tall grass)
    - Instead, have the code derive one kind of information from the other, so they don't get out of sync
- Sanity checks make sense even if you don't see a way the inputs could be corrupted
    - In this example, the game could ignore wild Pokémons whose level is above 100

The Pokémon developers made the same duplication mistake in the fourth generation of games, Diamond and Pearl, in which one can surf through a *door* to reach out-of-bounds locations, and exploit this to go to secret locations in the game. This is problematic because these locations are usually unlocked as a reward for going to a Pokémon event in the real world, such as watching a Pokémon movie, which brings money to the Pokémon developers. Players can thus use this bug to avoid having to pay.

Keeping Security in Mind

Security is an endless cat-and-mouse game between hackers (both good and evil) and the developers and maintainers of a system. Here are some hints for how to write code that will succumb to the incantations of a hacker.

While it may be tempting to write quick and dirty code to put something together and to make it secure and reliable later, this is a dangerous approach. First, when writing code in this mindset, you always come up with another feature to write - which delays even more the necessary security efforts. Second, the more your codebase grows, the harder it is to introduce security notions without rewriting a lot of code. Therefore, it is important to think about security from the start.

Thankfully, the first advice to write secure code is to write *good* code, and this is a topic we have already covered extensively during this course. Indeed, bugs are easier to spot and harder to introduce if:

- The code is well modularized, and interfaces only expose what is strictly necessary to their functionality
- The logic of the code is easy to understand
- There is no code duplication (as it is very easy to patch a bug in one place and forget about the other copies)
- Your code is well tested, using both unit tests, end-to-end testing, and fuzzing techniques

This list is not exhaustive, but it captures the general idea.

Trust boundaries should also be clearly established and enforced. A trust boundary describes a boundary between components that are trusted by the system and components that are not. Any data that crosses this boundary must be handled carefully, as it may originate from a malicious user trying to affect the behaviour of the system. You should always make sure that this data is of the correct format and fail fast if it is not. For example, in an Android application with a web server backend hosted in the cloud, the Application and the Server should be considered as independent: the App should not blindly trust any data received from the server (it may originate from a pirated server, for example), and the Server should not trust any data received from the App (it may come from a modified client or even from hand-crafted HTTP requests). Finally, **users are always outside of your trust boundaries**. You should always validate any input that originates from users.

The principle of least privilege is also a general guideline to keep in mind. It states that each component in your system should be granted exactly the permissions it needs to do its task, and not more. In a Web application, you could for example make sure that the login/registration module can only change the users in the database, and not anything else (their permissions, their messages, their orders…). A simple way to enforce that is to use different users (be it Unix users, Database users, API users, …) for different modules in your app, and make sure that a module cannot access the credentials from other modules. When implementing authorization on your system, it is also better to use credentials that give the least permissions possible, and issue credentials with a very short validity period and more permissions on a per-action basis.

These few hints are not everything, as a lot of security concepts are language or framework specific. In general, most of them can be found in the language or framework documentation. For example, Oracle provides an important list of secure coding guidelines for Java. Other lists of common flaws are very useful and should be carefully reviewed regularly, to make sure you know what are the flaws they describe and how to prevent them. We provide you with some of the most important of these flaws in the next section.

Common Security Pitfalls

Here, we will list some of the common flaws from OWASP Top Ten Web Application Security Flaws as well as the OWASP Mobile Top Ten, which apply specifically to Web and mobile applications respectively. You can also have a look at the Common Weakness Enumeration Top 25 list, that contains more general weaknesses.

In general, you will easily find security lists for your particular application domain online: you should make sure to review your code against each possible vulnerability and actively try to break your software from the point of view of an attacker (e.g. red teaming) to ensure its security. Also, these lists get updated fairly often given new technologies and popular attack vectors, so make sure to check them every so often.

Injection

Injection flaws can be disastrous, but at the same time are relatively simple to prevent. These flaws are characterized by the use of malicious user input in sensitive places. The most common types of injection are SQL Injection and Command Injection. The easiest way to prevent them is always to sanitize input as much as possible. We present here the three most common types of injections: SQL Injections, Command Injections, and Cross Site Scripting (XSS)

SQL Injections refer to SQL, a popular language to query relational databases. In SQL, you write requests to access tables in the database. For example, if you want to get all users with id 42, you could write `SELECT * FROM users WHERE user_id = 42 ;`

The injection happens when you want to set this user id using user provided data. If you write a login form, you can be tempted to write something like this (in Java):

```
connection.createStatement().executeQuery(“SELECT * FROM users WHERE user_email =
‘“ + email + “‘“); // Dangerous, don’t do this!
```

However, this is dangerous - especially if the user email is not previously validated. For example, a malicious user could provide email “`‘ ; DELETE FROM users WHERE 1 = 1 --`”, which would delete all users. More sophisticated attacks are indeed possible, and you can learn about them in multiple courses at EPFL.

How to avoid this? In the case of SQL queries, most client libraries provide a way to *prepare* requests. When writing the request, you indicate that some values will be replaced by user values. In some way, this is equivalent to creating a “SQL function” that takes some values. These values are handled in a specific way by the SQL server, meaning they cannot be interpreted as SQL code. Here is how you could write this in Java:

```
PreparedStatement statement = connection.prepareStatement(“SELECT * FROM users
WHERE user_email = ?”);
statement.setString(1, email);
return statement.executeQuery();
```

Command injections refer to system commands. For example, let’s say you want to write an application that takes any domain and runs a *ping* command on this domain and returns the output to the user. In Python, one could write it this way:

```
import os
result = os.system("ping " + domain)
```

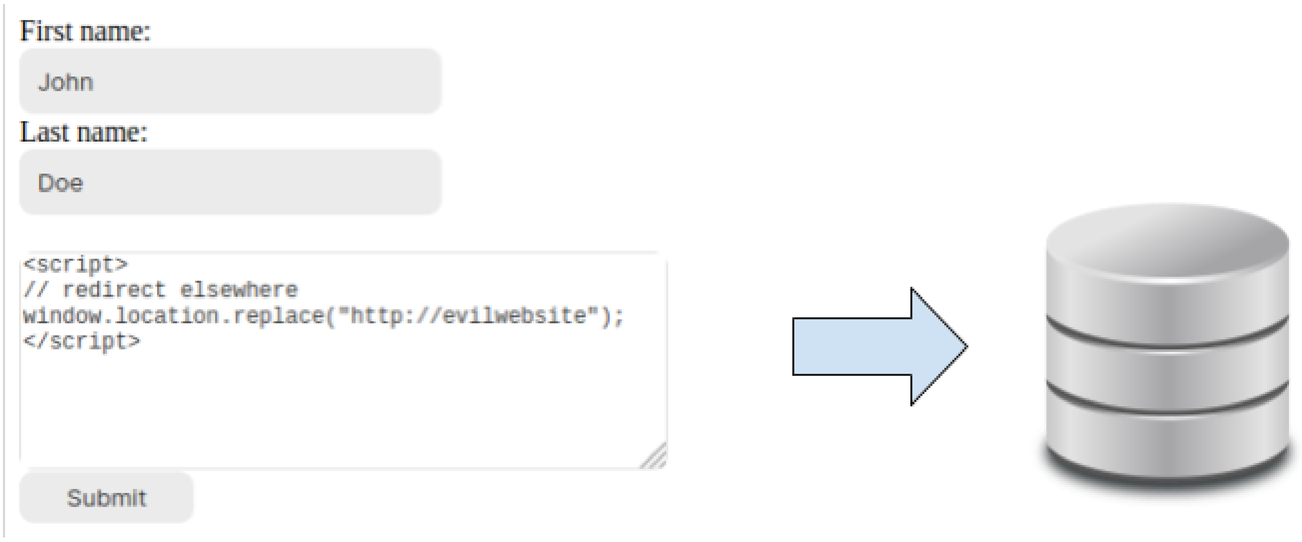
However, an user can provide a domain such as `"epfl.ch ; cat /var/www/passwords.conf"` to simply read all the passwords from your web server configuration.

How to avoid this? In this case, you should use "harder" but "safer" alternatives to os.system, such as the os.exec* family of function. They all take an executable path as an argument and usually consider all other parameters as **arguments** to this executable, instead of passing everything to the shell - which may interpret the arguments as commands.

Cross Site Scripting refers to the ability for an attacker to inject HTML source code in a web page served by your application, targeting the user browser. Depending on the severity, this may give the attacker the possibility to execute any Javascript code in the user browser, giving them full access to the targeted account.

There are two main ways in which an application can be vulnerable to XSS. The first one is if you store some user data and display it on some page without any sanitization. In this case, any person visiting the page can be affected. The second one is if you take some parameters (either in the URL or in response to a form) and display them back without sanitization. Here, the victim must be redirected to this page with a crafted parameter, so they need to click on a malicious link containing the payload.

Example: assume that users are able to leave some comments on a website through an HTML form, which are stored on the website's database. Those comments are stored in plain text, without performing any sanitization or verification whatsoever.


First name: John

Last name: Doe

```
<script>
// redirect elsewhere
window.location.replace("http://evilwebsite");
</script>
```

In this example, the provided input text will be fetched by any subsequent user visiting this page. However, nothing will be displayed, instead the user will be redirected to http://evilwebsite, simply because the script tag will be interpreted by their browser's Javascript engine as being part of the HTML document object model (DOM). The injected code is arbitrary, and can thus be used to steal cookies (with user sessions), run cryptominers, inject advertisements into the page, phish passwords, ...

How to avoid this? As always, it is very important to sanitize user input, preferably through a whitelist. If your input is not supposed to contain HTML, you should make sure that any potential HTML tag (i.e. `<script>`) should be escaped. In HTML, you can simply replace the characters `<` with `&lt;` and `>` with `&gt;`. The browser will display them as the correct characters but will not try to interpret them as HTML tags.

A lot of web templating frameworks automatically escape HTML on user input, and web application frameworks usually provide methods to do this. It is still a good practice to test that the input is correctly sanitized using end-to-end tests.

In general, use a library that automates this for you (e.g. DOMPurify). Escaping HTML perfectly can require a bit of effort, and using a library that does it well is the best way to make sure you don't forget anything.

See OWASP cheat sheets A1:2017-Injection and A7:2017-XSS ; Java Secure Coding Guidelines INJECT

Broken Access Control

This flaw describes the possibility for users to do actions that should be forbidden to them. For example, if a standard user can delete any account on your service using a bug, or if they can see the photos uploaded by any user because the directory has indexing enabled.

There is no easy way to avoid these bugs, but some common patterns can be avoided.

For example, you should always protect *direct object accesses*: if a user account has an id, you should make sure that the URL `/users/delete/:account_id` is protected and can only be accessed by an administrator account. Even if the given ID is supposed to be secret, random and complicated to guess (non sequential, with high entropy), all accesses to the object should still be checked for permission.

Writing end-to-end tests that try to call each endpoint without the appropriate permissions can also help you gain confidence that you did not entirely forget to put access control in. It can also protect you from later updates that could accidentally disable access control on some endpoints.

See OWASP cheat sheet A5:2017-Broken Access Control ; Java Secure Coding Guidelines ACCESS

Insecure Deserialization

Serialization is the concept of *translating a data-structure or object to a form in which it can be stored.* Deserialization is the opposite operation: translating some stored data to an object or data-structure.

JSON (JavaScript Object Notation) is a very common format for data serialization and deserialization. Libraries to serialize and deserialize objects exist in a lot of languages. For example, the following Scala objects could be serialized in JSON this way:

```
case class Address(street: String, city: String, zipcode: String, country: String)
case class Person(firstname: String, lastname: String, age: Int, address: Address)

val arthur = Person("Arthur", "Dent", 42, Address("Country Lane", "Cottington", "PO16 7GZ", "United Kingdom"))

print(Json.stringify(arthur))

>>>
{
      "firstname": "Arthur",
      "lastname": "Dent",
      "age": 42,
      "address": {
            "street": "Country Lane",
            "city": "Cottington",
            "zipcode": "PO16 7GZ",
            "Country": "United Kingdom"
```

```
        }
}
```

However, there are other ways to serialize and deserialize objects. You can even define your own (for example, if you have a fixed number of fields that always come in the same order, you can simply define that each field should be separated by an arbitrary string such as “`$$SEP$$`”, this would be a serialization scheme). However, when you try to deserialize user data, you must always be very cautious.

Java is natively capable of serialization or deserialization of any object to a string representation. However, there are absolutely no integrity checks on these objects: quoting the Java Secure Coding Guidelines: **“Deserialization of untrusted data is inherently dangerous and should be avoided”.** If you simply deserialize any user input in Java, an attacker could simply write something else in the object - which could lead to a trivial *remote code execution* attack. Therefore, you should never use Java deserialization on untrusted data.

Even when using safer representations such as JSON or custom formats, there are multiple possible attacks.

First, bugs in the deserialization parser can cause infinite loops or very high memory usages, which can lead to a denial of service attacks. This is a common bug for naive parsers in XML, but can also happen in JSON if the size of the input is not limited (for example, an attacker could send a very huge JSON string that would exhaust the system memory when deserializing).

Second, you should always remember that serialized data is not protected against tampering. For example, let’s say you want to authenticate an user and send them a HTTP cookie containing the following data: `{“user_id”: 5, “admin”: false}`. An attacker can simply modify the cookie to read `{“user_id”: 5, “admin”: true}` to become an admin, if no other safeguards are put into place. Therefore, if your object serialization contains sensitive data that must not be changed by the user, it is very important to authenticate it. If you use JSON, JSON Web Tokens (JWT) are a good way - but they also have problems on their own.

A rule of thumb is to always be cautious when deserializing user-data, and to use a battle-tested library to do so, as naive homemade implementations may be subject to known flaws unbeknownst to you.

See OWASP cheat sheet A8:2017-Insecure Deserialization ; Java Secure Coding Guidelines SERIAL

Using Insecure Components

Important applications often depend on dozens of external libraries, which themselves depend on dozens of other libraries. When you want to perform a specific task (for example, JSON serialization or network communication), it is often faster and safer to use specialized libraries that are very good at this specific thing (for example, GSon and Netty respectively). Writing a web-app from scratch in most languages would be re-inventing the wheel: you’d need to develop a whole network stack to handle TCP connections, parse HTTP requests, then dispatch them to your workers and write the reply. Instead of doing that, it is way easier and safer to use a framework (such as Spring or Play, in Java).

Using external libraries is safer, because these libraries are developed by teams of skilled developers that specifically work on them. They are also used widely, which means that discovered flaws are usually patched quickly. However, these observations don’t apply to *all* libraries - some of them are almost unknown and maintained by a single developer, possibly even abandoned. Therefore, it is a good practice to **make sure that a library is well supported and has a wide community** before depending on it in a project.

It is also important to keep these libraries up-to-date. Critical flaws are found every day in tens of open source projects, even the most popular ones. **Running an insecure version of a widely popular library is extremely dangerous**, as easy-to-use exploits are often widely available to attackers, who can then exploit the bugs effortlessly. This affects everyone (e.g., Facebook was affected in September 2020).

See OWASP cheat sheet A8:2017-Using Components With Known Vulnerabilities ; Java Secure Coding Guidelines FUNDAMENTALS-8

Improper Platform Usage and Insecure Data Storage

Most computer systems include secure facilities for the storage and processing of highly sensitive data such as cryptographic keys, database passwords, user credentials, … Examples include the iOS keychain, Android Keystore, or Linux keyrings, as well as hardware-hardened circuitry (Android hardware-backed keystore, Intel Secure Key, or ARM TrustZone, ...) and cryptographic libraries (Android Cryptography, Java / C# Bouncy Castle, ...)

It is your responsibility as a developer to learn about the best security practices of your specific application environment. Failure to do so will result in you reinventing the wheel with great risks of exposing flaws in your implementation (designing truly secure cryptography and storage technologies is extremely hard!). There are many (bad) software systems in the wild that store secrets in plaintext storage (such as files on the disk), or that implement their own (doubtfully secure) encryption mechanisms. Remember that security through obscurity is usually a recipe for disaster, and use Kerckhoffs's principle which states that “a cryptosystem should be secure even if everything about the system, except the key, is public knowledge”.

See OWASP cheat sheet about M1: Improper Platform Usage, OWASP cheat sheet about M2: Insecure Data Storage

Code Tampering and Reverse Engineering

In these attacks, hackers will typically exploit the application binary and files themselves by decompiling assembly / bytecode executables and retrieving assets (images, configuration files, ...) from the software package. Example scenarios include:

- Changing the application binary or its assets and repackaging it, then distributing it under a different name
- Wrap or redirect API calls from the application to external code (library, OS, remote services) to intercept data in-flight or execute malicious code (stealing credentials, running cryptocurrency miners, ...)
- Extracting constants from the binary code such as API keys, passwords and other hardcoded secrets
- Stealing proprietary algorithms (e.g. trained ML models)

Typically, any code that is accessible to users (mobile apps, javascript frontends, desktop applications) is susceptible to reverse engineering of its source files. By applying security best practices (such as using the aforementioned secure facilities of a system), the execution scope of the end-user software should stay limited. Tools such as code obfuscators (such as Proguard in Android) may also considerably slow down the progress of attackers in understanding the code binary by removing the attached semantics (class names, variables, ...).

See OWASP cheat sheet about M8: Code Tampering, OWASP cheat sheet about M9: Reverse Engineering

Extraneous Functionality

Sometimes hackers don't need anything other than the software itself to find vulnerabilities: debugging tools (such as loggers, privilege backdoors) and WIP code (demo features, test code paths) may find their way into the final product, if developers are not careful about removing them. Attackers can then very easily collect information about the functionalities of the software, or may even use these unwanted features directly. For instance, a mobile app may contain log statements that are observable by simply connecting the phone to a debugging tool (such as adb or Android Studio) and leak implementation details. Another example is an authentication server that should always require users to use 2-factor authentication, but has a hidden backdoor endpoint used for testing purposes that bypasses 2FA: hackers may simply find the corresponding calling code, and try to execute it themselves to gain access to users accounts, with some additional social engineering.

See OWASP cheat sheet about M10: Extraneous functionality

Preserving Users' Privacy

Privacy has become an important concern in modern times. To some degree, awareness has improved as well. We are less likely today to hand out all of our personal data and PII (Personally Identifiable Information: name, address, phone number, …) to unknown websites. In this section, we will briefly discuss some ways in which you can make sure that the data you collect about your users is not misused or leaked.

Strategy 1: Data you don't collect is data you don't need to protect

The first way to protect user data is to refrain from collecting it.

When writing a form, make sure that every field inside that form is needed, because as soon as the data reaches your server, you are responsible for keeping it secret. The same concept applies to any way of collecting data: automated logs, machine learning algorithms, …

Recent regulations require you to explicitly request user consent before collecting certain data. Data collection also comes with multiple legal requirements: you must keep the data safe from unauthorized access, cannot share it with third parties unless explicitly authorized by the user, you must allow the user to download a copy of their data or to remove it from your servers. You may collect some data without requiring any consent, but the scope of this collection is very limited - think username/passwords, for example.

It may sound tempting at first to collect every possible information about your user base, but remember: **all this information needs to be kept private.**

Strategy 2: Hash / encrypt user data

When the data reaches your backend server, you want to make sure that it stays safe even if something bad happens (think: a bad actor logs-in as admin on your server).

Encrypting the whole database is usually a good thing, but depending on the encryption scheme you choose, you won't be protected against the same kind of attacks. For example, encrypting the database on disk won't protect you if the key to decrypt it is on the machine.

Using multiple database accounts with limited permissions can also limit the consequences of bugs in your application code. For example, if your database software allows it, you can make sure that the account you use to create new users is not able to change the permissions of the users or to return a list of users in the database. This way, if a bug in your registration code allows an attacker to write arbitrary requests, they will be constrained by the permissions of the database user.

Keeping passwords safe

An important thing to think about is **hashing passwords.** You may think that if your app is not very important, your users' passwords don't matter. After all, what could possibly go wrong if an attacker gets a user password for your Zoo Management smartphone game? Well, that user has possibly **reused this password elsewhere**. Password reuse is a very common problem because humans are not good at remembering multiple complex passwords. Therefore, you should assume that any user on your app may have used a very important password to register, and you must make sure that this password **never leaks.**

Thankfully, there is an easy way to achieve that: hashing. Contrary to encryption, which is reversible (i.e. if you know the key, you can decrypt and read the original value), good hashing is only one way, meaning it's almost impossible to find the original value given the hash of that value. Using hashes, you don't have to keep an encryption key secret. You also don't have the risk of one of the developers using the key to decrypt the password of an user to access their email account.

Hashing functions are provided in libraries in almost all programming languages. Some of them, as PHP, include them in their standard library. They can be divided in two categories:

- **"Fast" hashing functions**: SHA1, MD5, SHA3…
  These functions are often used to compute hashes for files. They can hash multiple gigabytes in a few seconds, into a short series of bytes written in hexadecimal form (20 bytes for SHA1)
- **"Slow" hashing functions**: Argon2, BCrypt, …
  These functions are slower, and you can adapt their cost when calling them (making them use more or less memory, and more or less CPU time). "Slow" here means that a standard password can take about 500ms-1s to hash.

When storing passwords, **you should only use "slow" hashing functions**. First, they are usually specifically built to handle passwords. Second, the fact that they are slow means it will be very hard for an attacker to *bruteforce* a hash to try to recover the password. Indeed, if an attacker gets access to a list of hashed passwords, they can try a list of common passwords, or simply all permutations of some length. In 2012, using 25 GPUs in parallel allowed attackers to test 60 billion SHA1 hashes per second (https://www.zdnet.com/article/25-gpus-devour-password-hashes-at-up-to-348-billion-per-second/). Argon2 is way harder to parallelize and to run on a GPU, therefore it is not as trivial for attackers to bruteforce passwords hashed using it.

But how to check a user password if it is impossible to decrypt it? That's simple: because the hash function is deterministic, you can simply apply the hash function again on a provided password and compare it with your stored hash to check if it's correct. You will see a small code sample below.

When hashing passwords, **it is important to use a salt**. A *salt* is a random string added to the password before hashing it. You need to have one random salt for each user, and you obviously need to keep it with the password, otherwise it would be impossible to re-use it when hashing user input. Using a hash makes it harder for an attacker to see if two users used the same password (since hashing is deterministic, if two users choose password "aaaaa", they will get the same sha1 hash "5cafdebe4e78588628681c0fa5fda8a410ccd966"). Moreover, not using salt means that an attacker that wants to bruteforce stored passwords can do it only once: every time they try to hash a guess, they can immediately see all users who used that as a password.

Usually, **the libraries that provide the hash functions also include automatic hashing** (and you should **never** try to write your own hash/cryptography functions libraries to use them in the real world). You have a function to hash the password when the user registers (`` `password_hash` `` in PHP), that will create a salt, hash the password with the salt, and return a string containing both the salt and the hash ; and a function to verify a user password (`` `password_verify` `` in PHP), that takes a user provided password and a previously hashed password and checks that they correspond. For example, Argon2 hashes created using PHP's *password_hash* look like this:

```
$argon2id$v=19$m=65536,t=4,p=1$LkZFZDIwNHZqR2ttMHhRTA$sIy1TMe9frrczfWpOSMrENt1jWk
Ct4vSyzm+i2iGQGo
```

It's a string in which parameters are separated using the dollar sign. The name of the algorithm comes first, then come multiple parameters for the algorithm, and finally the salt `LkZFZDIwNHZqR2ttMHhRTA` and the hashed password `LkZFZDIwNHZqR2ttMHhRTA$sIy1TMe9frrczfWpOSMrENt1jWkCt4vSyzm+i2iGQGo`.

Here is a code example using argon2 in Scala (*reminder: in scala, you don't need to write* return *explicitly*). It implements a custom interface *HashProvider* that defines two methods:

- hash(password): the function that is called when the user registers for the first time, generating the hash using defined complexity parameters

- check(hash, input): the function that is called when the user tries to login using a given *input* password.

```
private class Argon2HashProvider(tpe: Argon2Types, salt: Int, len: Int) extends HashProvider {
  private lazy val argon2 = Argon2Factory.create(tpe, salt, len)
  private val MAX_HASH_TIME_MS = 1000
  private val MEMORY_KBYTES = 65536
  private val PARALLELISM = 4
  private lazy val iterations = Argon2Helper.findIterations(argon2, MAX_HASH_TIME_MS, MEMORY_KBYTES, PARALLELISM)

  override def hash(password: String): String = {
    if (password != null)
      argon2.hash(iterations, MEMORY_KBYTES, PARALLELISM, password.toCharArray)
    else throw new NullPointerException
  }

  override def check(hashed: String, input: String): Boolean = {
    if (hashed != null && input != null)
      argon2.verify(hashed, input.toCharArray)
    else throw new NullPointerException
  }
}
```

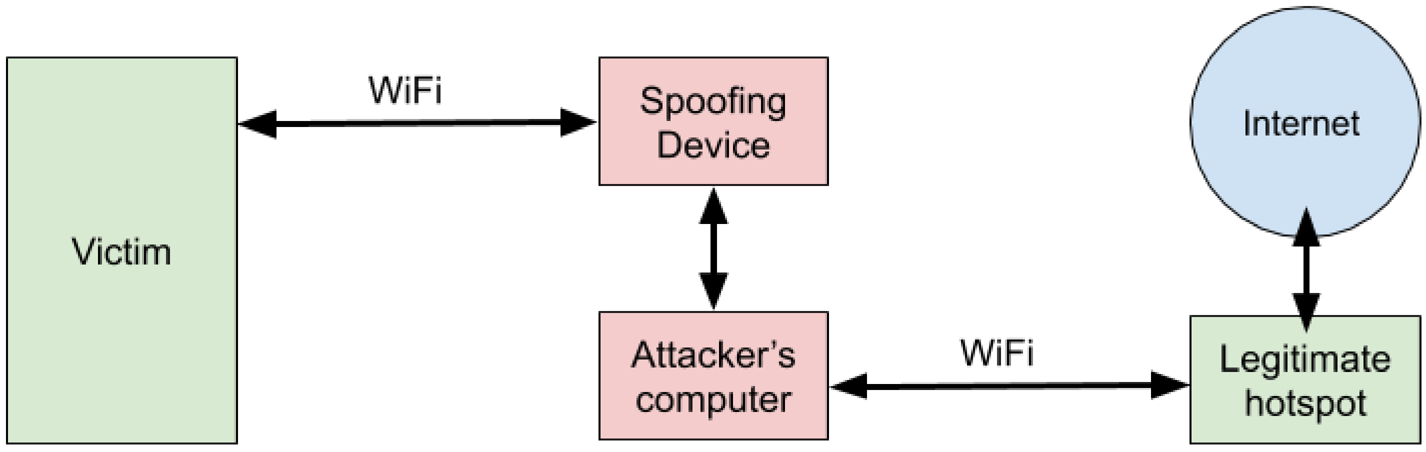
Note: protect your backups as well as your data!

Even if your main database server is a fortress that no-one can ever enter, it is useless if you regularly backup your database and put it in a freely accessible directory. This is in fact a problem that is quite common and has led to the leak of very personal data, such as with the Red Cross Blood Service in Australia data leak.

Therefore, make sure that your backup strategy doesn't accidentally involve keeping database archives in indexed directories that can be found using any search engine.

Strategy 3: Encrypt data between the server and the client

Users often browse the web from networks that are insecure, such as public wifi hotspots (think Airport WiFi). On such networks, it is particularly easy for an attacker to set-up a device that imitates the hotspot but acts as a Man-in-the-Middle between user devices and the actual network. This means that when a user *thinks* they are connected to *Zurich FH FreeWifi*, they are actually connected to an attacker laptop, that redirects all your requests to a legitimate network but spies on them or modifies them while doing so.



Using this scheme, an attacker can **read** what is exchanged between the victim and the internet, and **modify** that: either by blocking some websites, or inserting malicious links in them.

However, there is a simple protection to these attacks: **TLS** - for Transport Layer Security. As you may recall from your Computer Networks course, the Transport Layer is the network layer in which TCP and UDP reside. **TLS** is a protocol that adds encryption to TCP. Using TLS, the client and server first exchange a **handshake**, in which the server presents a signed certificate to prove that it is legitimate (usually, this certificate simply contains the **domain name** of the server, and enables the client to make sure that the server is the legitimate owner for that domain name), and then both the client and server derive a secret key to encrypt the rest of the communication. There are multiple versions of the TLS standard, and it admits multiple algorithms to derive the signature and the keys.

In HTTP, the use of TLS is denoted by the *S* after *HTTP* in the URL (*https://www.epfl.ch/*). HTTPS uses port 443 instead of 80 and uses TLS instead of simple TCP. Today, using HTTPS is highly recommended, as browsers begin to display insecure warnings on forms sent via an insecure connection and search-engines lower the rank of websites that don't offer HTTPS.

Deploying HTTPS is nowadays very easy and can be done for free. If you rent your own servers, it is very simple to get a certificate and to automate its renewal using Let's Encrypt. Getting the server configuration right can be a bit tricky, but multiple tools exist to help you with that: see for example https://ssl-config.mozilla.org/. Otherwise, you can also use a CDN (Content Delivery Network), such as Cloudflare, as most of them give you a free TLS certificate and easy configuration options.

There is usually no good reason to leave a part of your service reachable using standard HTTP - and be aware that it can be dangerous to do so! Indeed, if your site can be accessed via simple HTTP, a man-in-the-middle can simply block any connection to your HTTPS server and instead force the use of simple HTTP requests. The same goes for "insecure" algorithms: TLS allows a lot of algorithms to be used, and some of them are nowadays considered insecure. Leaving them enabled makes sure that old clients can still connect, but it makes recent clients more vulnerable, as a man-in-the-middle can use downgrade attacks to force the client to use the insecure algorithms.

Strategy 4: Beware of side-channels

Even when your data is not accessible directly, it can be possible to access it via *side channels*, by observing the result of some actions to infer what the value is.

The website Ashley Madison promoted itself as a website for extra-conjugal encounters. It is not hard to understand why their customers wanted to stay relatively anonymous, and why they would have been very annoyed if it was trivial to check if their email address was registered on the site… which was actually the case.

When poorly implemented, a password reset form can simply leak if the user already exists. In fact, a naive way of implementing password reset would be to ask the user its email, then send them a recovery link if the email exists, and otherwise display an error. However, this makes it very easy to check if an email address exists in the database. If Ashley Madison displayed the same message in both cases, *they removed the email field and button if the email existed*! This means that no user on their service was really anonymous, and it was trivial for anyone to check if someone had an account there.

How to get this feature right? The easiest way is to send an email regardless of whether the account exists or not. If you have an account, the email contains a recovery link. Otherwise, it contains a generic message such as "You tried to recover your password for this email but your email was not registered on our service". The page shown to the user must indeed be exactly identical in both cases.

Even if you get the password reset form right, another feature that can expose the existence or not of a user in a database is the registration form. On most websites, the email address is the login, therefore it has to be unique among all users. And therefore, if a user attempts to register using an email address that already exists, you need to tell them that they can't register, as the email is already used… Here, you can use the same strategy as before: when registering, send a confirmation email to the user. If the email already exists, simply ask them to login instead.

Finally, a sneakier way to guess if a user exists or not is to use the login feature and to time how long it takes to return "Wrong credentials". Indeed, a naive way to handle login would be the following:

```
login(email, password):
    account = database.get_user(email)

    if account is None:
        return WrongCredentials
    else:
        return password_verify(password, account.password_hash)
```

Now think about what happens if your password hash function is slow - like about 500ms. If the account exists, then you need to verify the password - which takes about 500ms. However, if the account doesn't exist, the function returns immediately. Assuming the round-trip time is not high (for example, 50ms), it is very easy for the attacker to distinguish both cases, leaking if the email exists in the database.

Here, the mitigation can be a bit tricky. One simple way to mitigate this problem is to **hash something regardless**. For example, you can have a constant somewhere containing a hashed random string. Whenever someone tries to login using an account that does not exist, simply use *password_verify* on that hash. This will take the usual 500ms and make it impossible for the attacker to distinguish both cases. Indeed, you don't want to ruin your efforts by returning a different error message depending on whether the email exists or not in the database.

Not all applications need to hide their list of users this way - however, these advices can apply to any kind of sensitive information. Always try to reason about how an attacker could use innocent features to gain knowledge about information that should be kept private.