## Attack Prevention

### Injection

Preventing injection attacks requires the interpreter to be able to make a distinction between untrusted data and actual commands and queries, which can be achieved by one of many ways.

The most obvious solution would be to forgo use of the interpreter altogether and instead use a safe API (Application Programming Interface), or at least an API that provides a parameterized interface.

If APIs can’t be used for this purpose, injection can be prevented using an interpreter and making careful use of escaping routines. Injection attacks will usually involve injecting special characters that are normally not found in legitimate queries (such as ‘<’ or ‘>’), so checking for these special characters might help differentiate between an actual user query and a potential script from an attacker. By escaping these special characters (using the escape syntax for that particular interpreter) or sanitizing the input, it should be possible to block those kinds of attacks.

Another possibility is white list validation, which only takes in data structured in a particular way, such as properly formatted dates or e-mail addresses, and disregards the rest of the input.

### Broken Authentication and Session Management

The most important advice for an organization is to make sure that their developers have a single set of strong authentication and session management controls. It’s also important to prevent XSS flaws that could be abused to steal session IDs.

Authentication credentials should always be hashed or encrypted when stored, and should not be sent over unencrypted connections. Special care must be taken with account management functions such as password recovery, for they could be abused to guess or overwrite credentials.

Session IDs are especially vulnerable. They shouldn’t appear in the URL, they should timeout, and they should be rotated after a successful login. Also, it’s important that any used authentication tokens are properly invalidated during logout.

### Cross-Site Scripting (XSS)

Preventing cross-site scripting requires separating legitimate browser content from untrusted data. The simplest method, effective against both reflected and stored XSS, is to filter data according to its HTML context (body, attribute, URL) and escape any data deemed untrusted.

Similar to injection, white list server-side input validation can also be used, allowing only data that follows a specific format, such as length ranges or valid characters, and disregards everything else.

Content Security Policy (CSP) can be used to restrict from which location and what type of resources is the client browser allowed to load, so it could be used to defend against XSS across an entire webpage.

### Insecure Direct Object References

The simplest defense against insecure direct object references is to establish an appropriate check access control. Whenever an object or service is requested, the application should perform an access control check to confirm whether the user is authorized for the requested object.

Another method is to use an indirect reference map, which would be used for mapping from a set of internal object references to a set of indirect references. This way, the user would only have access to these per-user specific indirect references, which would be sent to the database to retrieve the actual request (the internal object reference), such as a filename or database key.

### Cross-Site Request Forgery (CSRF)

The most effective way to protect against CSRF is to include an unpredictable token in each HTTP request, so that the attacker would be unable to execute a CSRF without knowledge of this token. This token should be unique per session and should preferably be included in a hidden field, so that it is sent in the body of the HTTP request and not in the URL.

Another solution is to require the user to confirm all their transactions. This can be achieved by asking them to reauthenticate or requesting proof that they are an actual user (such as by a CAPTCHA).