## 14. Introduction to Web Secure Coding

In this Learning Module, we will cover the following Learning Units:

- Trust Boundaries
- Untrusted Input Handling
- Output Encoding and Character Escaping
- File Handling
- Parameterized Queries

Security should not be an afterthought when developing applications. As an application grows, it becomes increasingly more difficult to "add security later". Instead, we should try to write code in such a way that reduces the chances of introducing vulnerabilities in the first place, a practice known as secure coding parctice.

There are various practices that can improve the overall security posture of an application. However, since modern frameworks often include a baseline set of security controls. In this module, we will focus on controls that can prevent or mitigate the most common vulnerabilities in web applications.

We will start by discussing trust boundaries and how they factor into application security. Then, we'll focus on input validation and output encoding. Finally, we will discuss parameterized queries. While there are many other topics related to secure coding, these will form a basic foundation for the process of securely coding web applications.

### 14.1. Trust Boundaries

This Learning Unit covers the following Learning Objectives:

- Understand trust boundaries
- Understand how to identify trust boundaries in applications
- Understand subresource integrity
- Calculate a hash using openssl
- Understand the concept of defense-in-depth

#### 14.1.1. Intro to Trust Boundaries

A trust boundary in an application is a point at which data or commands change permission levels. For example, data submitted by a user without validation is at a different trust level than data validated by the application server-side. Likewise, unauthenticated users operate at a different trust level than authenticated administrative users. Security controls must be applied consistently at all trust boundaries to protect the application from untrusted input.

Developers must understand trust boundaries between disparate systems to avoid "second order" attacks wherein a payload is submitted to one system and exploited in another. This concept is becoming more important as developers trend toward smaller applications with collections of microservices.

Let's review an example and practice identifying trust boundaries in a web application.

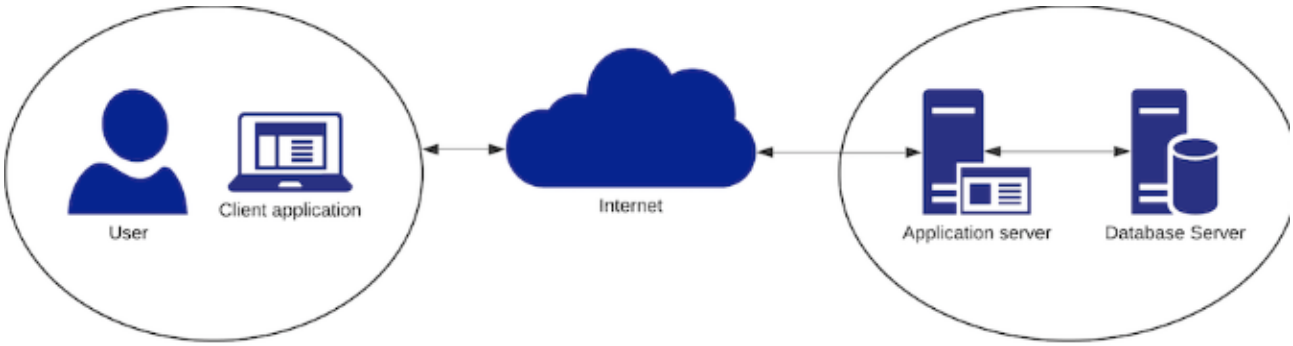
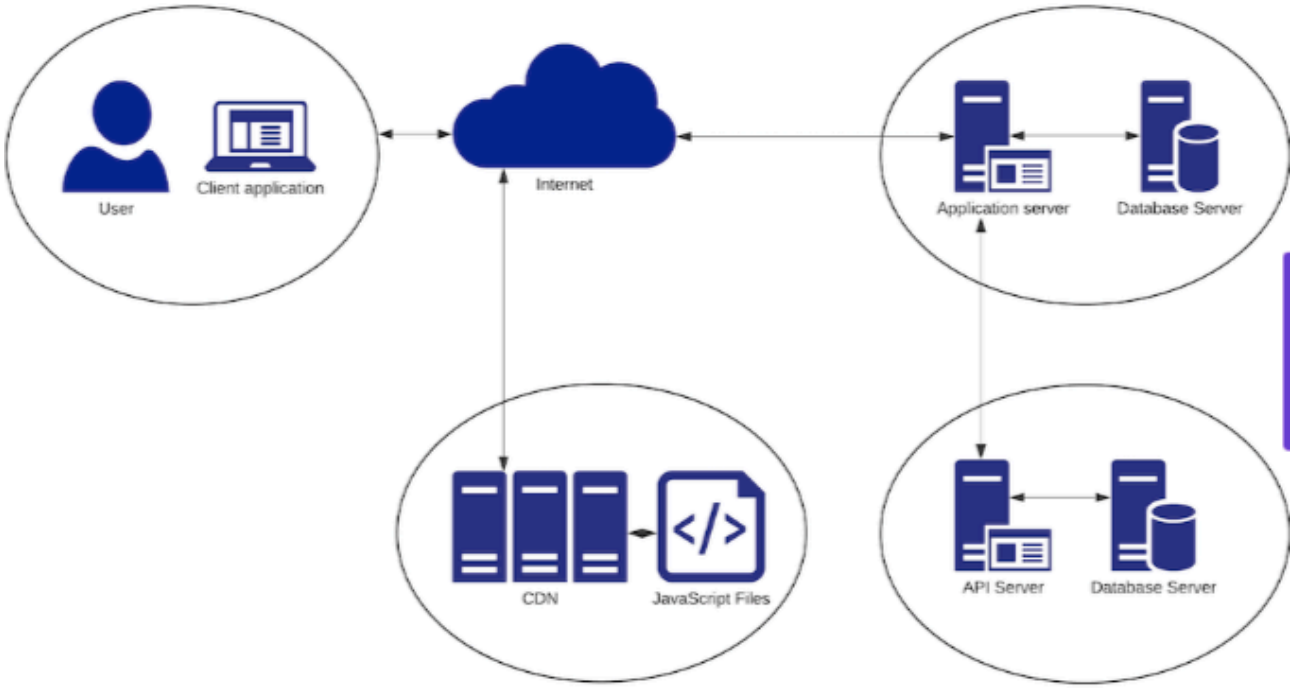



Figure 1 describes the data flow in a basic web application. Users interact with an application server over the Internet. The application server connects to a database server. In older, monolithic applications, the database might even run on the same server as the web application. For the sake of simplicity, we have omitted appliances like routers and firewalls from the diagram.

The first trust boundary exists between the application server and the Internet. While our application provides content (such as HTML, JavaScript, images, etc.) to the user and their browser, we have limited ways to guarantee outside parties have not manipulated our content in transit or in the browser. For example, some ISPs inject advertisements into HTML responses and some users attempt to block them with various browser plugins such as ad blockers. Each of these manipulate data in transit to the browser.

A second trust boundary occurs between the application server and the database server. While an application might be responsible for writing most of the data contained within the database, there may be situations in which unintended data ends up in the database. For example, database administrators (DBAs) might interact with the database directly. Attackers might be able to bypass input validation and inject malicious data into the database.

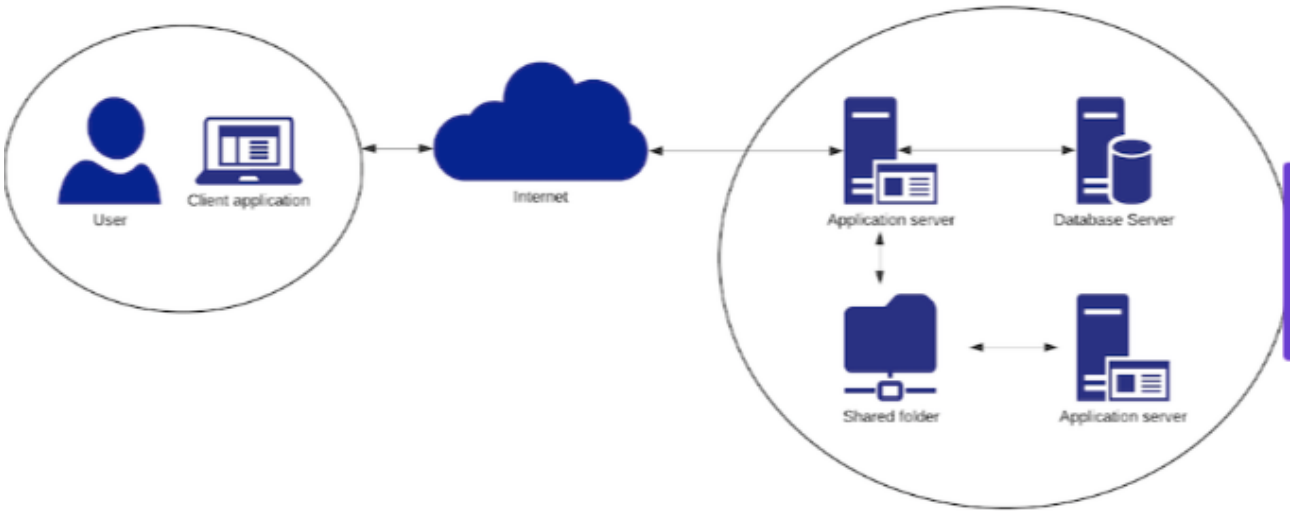
Next, let's consider a web application using a basic interpretation of microservices architecture.



The web application in Figure 2 uses an API to handle some logic and a *content delivery network* (CDN) to host and deliver JavaScript files. The trust boundaries we identified last time still apply here. The addition of an API creates a new trust boundary between the web application and the API. Another trust boundary exists between the API server and its database, but from the perspective of the developer responsible for the web application, that boundary is non-essential. If we do not automatically trust any data from the API, downstream integrations of the API do not concern us.

Using a CDN to handle resources, such as JavaScript, introduces an interesting delegation of trust. If our application serves HTML that includes resources from an external domain, we don't have direct control over those contents. If attackers compromise the CDN, it could start serving malicious content that applications may load. We'll learn how to prevent this type of attack in the next section.

For the following exercise, consult the data flow diagram below.




**Exercise: True or false: A trust boundary exists between the two application servers because they connect to the same shared folder.**

### 14.1.2. Subresource Integrity

*Subresource Integrity* (SRI) instructs the browser to not load a resource if the hash of the resource does not match a pre-provided value. We can set an *integrity* attribute with the pre-defined hash value as part of a *script* or *link* element.
Let's review an example of a *script* element with SRI enabled.

```
<script src="scripts/bootstrap.min.js" integrity="sha384-
B4gt1jrGC7Jh4AgTPSdUtOBvf08shuf57BaghqFfPlYxofvL8/KUEfYiJOMMV+rV"
crossorigin="anonymous"></script>
```

The *script* element in Listing 1 includes an *integrity* attribute. Notice how the *integrity* value begins with "sha384". This indicates the base64-encoded hash value after the dash was calculated with the SHA384 algorithm. The currently supported algorithms are SHA256, SHA384, and SHA512. We must include the *crossorigin* attribute when using SRI for resources hosted on external domains. This also means the site hosting the resources must support *Cross-Origin Resource Sharing* (CORS) to enable SRI. The "anonymous" value used in this example means the browser will only send credentials, such as HTTP cookies, if the request is to the same origin.
In Kali Linux, we can use *openssl* to calculate and base64-encode a hash value of a file. We need to set **dgst** to create a message digest, specify the algorithm with **-sha384**, set **-binary** to generate binary output, and finally the file we want hashed. Since we need the result base64-encoded, we'll pipe the binary output to openssl (with the **base64** argument) to base64-encode the output and use **-A** to write the base64 output as a single line.

```
kali@kali:~$ openssl dgst -sha384 -binary test.js | openssl
base64 -A
IMR5mPD7vl9kNTutndXHoFmH1NmdJzbzv1hmrKNN8UR+98beWaNSRJ9d9Ymkcs+c
```

Even a minor change to the file will change the file's hash value. For this reason, it is common to link to a specific version of a resource when using SRI to prevent browsers from rejecting a resource due to updates.

**Labs**

**What is the SHA256 hash of the provided sri_exercise.js file?**

### 14.1.3. Defense in Depth

Software should have multiple layers of security embedded throughout the application. This is known as *defense in depth*. Our goal is to leverage centralized controls, while avoiding a single point of failure.

For example, it's a good idea to use a set of functions to handle input sanitization consistently across the application. However, if we only apply input sanitization to unauthenticated users while exempting authenticated users, we put the application at risk. An application should consistently perform output encoding when displaying user-supplied data, even if it has already validated the data. Consistently applying multiple controls increases the security posture of an application and may prevent a single bug in one security control from allowing attackers to compromise the application.

Additionally, we cannot rely on client-side controls as a sole security mechanism. We previously mentioned how attackers can bypass these controls.

JavaScript in-browser validation can reduce erroneous traffic from the user to the server from normal users but client-side validation is not a secure control. Malicious users have control over their browser and can modify the DOM in their browser or send requests with non-browser tools such as curl or Burp Suite.

One of the best ways to prevent common web application vulnerabilities is to apply a set of consistent security controls throughout an application's data flows. This may include validating user-supplied input, using parameterized queries when accessing a database, and performing output encoding when returning data to users. Using this approach, an attack payload must make it through multiple layers of controls to be effective. We'll discuss these types of controls in subsequent Learning Units.