Chapter 33. Defending Data and Objects

Despite being a target for various forms of attack, data and objects within application code are actually quite simple to defend. Outside of data storage in a database, most objects that programming languages interact with are stored either *ephemerally* (in-memory) or *persistently* (in-filesystem).

Because programming languages perform operations primarily in memory, most of the time when dealing with persistent file-stored data, the data is brought into memory for the duration of operations. Because of this, there are many cases where defenses that benefit ephemeral data will also benefit filesystem data.

Defending Against Mass Assignment

Mass assignment attacks are relatively easy to prevent provided consideration is given to security while programming a web application. Consider the following mass assignment vulnerability:

```
/*
 * This is a server-side API endpoint for updating player data
 * for the web-based video game "MegaGame".
*/
app.post("updatePlayerData", function(req, res, next) {
 // if client sent back player state data, update in the database
 if (!!req.body.data) {
   db.update(session.currentUser, req.body.data);
   return res.sendStatus(200); // success
 } else {
   return res.sendStatus(400); // error
 }
});
```

This vulnerability exists because the developer chose to *trust* the data sent by the client and update database fields based on an object that could be tampered via a malicious user.

There are two generally acceptable defenses against this sort of assignment attack.

Validation and Allowlisting

The easiest method of mitigating mass assignment is to simply restrict the fields that are accepted from the client. This is best done in the form of an *allowlist*.

By creating the following allowlist and implementing it in the form of a validation prior to calling the function `db.update()`, we can rest assured that mass assignment of the `admin` field is not possible:

```
const allowlist = ["hp", "location"]; // only allow these two fields to be update
```

Data Transfer Objects

A second, more intensive mitigation is to create an intermediary object called a *Data Transfer Object* (DTO). A DTO is used when passing data between services or function calls.

In our case, the DTO would look as follows:

```
const DTO = function(hp, location) {
 this.hp = hp;
 this.location = location;
};
```

By running all incoming data through the DTO constructor, any additional parameters will be rejected. This prevents malicious client data from ending up in the `db.update()` function.

Defending Against IDOR

Smart application architecture is the first defense against IDOR. Objects and files should never be referenced directly or via any API structure that is easily guessable. In the case where a user needs access to a specific file, a combination of masking the API call (to not show the specific filename) and performing authorization checks prior to each file return is an excellent solution.

Where the preceding is not possible, using randomly generated filenames and object references can provide security—but only if other mechanisms (like rate limits) exist and randomly generated filenames have sufficient entropy and complexity to the point at which millions of guesses would be required to find another file. Again, this is not the ideal solution but typically more of a short-term mitigation until a long-term re-achitecture is possible.

Defending Against Serialization Attacks

At the core of every serialization attack is *weak serialization*. This means that the primary mitigation to prevent serialization-related attacks against your web application is to make use of strong, well-tested serialization and deserialization libraries.

In the case of web applications, it's often best to choose an open source library with millions of users and a history of security audits. Choose a popular format like JSON or YAML and avoid formats with little or no track records.

Consider the type of data you intend to serialize. Data that could easily be interpreted by the server or browser as script, or that contains common escape characters, should be sanitized either by the serializer or another function to ensure such characters are removed prior to serialization and deserialization occurring. It is possible to allowlist input characters and

object types to reduce the probability of such risks if the serializer is not capable of handling such characters and objects.

Summary

While attacks against complex objects and data exist, and appear to be a fault of the programming languages of the web, they are often easy to defend against. Being prepared to design a web application with such data and object attacks in mind, and understanding the risks, should give you peace of mind because mitigations are readily accessible for these types of issues.