Chapter 15. Attacking Data and Objects

The majority of modern programming languages implement logical program design utilizing two distinct capabilities: *data*, which is typically represented in the form of *objects*, and *actions*, which are most often represented in the form of *functions*. Even in programming languages that are not object-oriented programming (OOP), objects are still usually defined as first-class citizens.

The term *first-class citizen* is a programming language design concept used to refer to an entity within a programming language that can be assigned, reassigned, modified, passed as an argument to a function, and returned from a function. Almost all modern programming languages define objects (data) as first-class citizens, but not all modern languages define functions (actions) as first-class citizens. As such, it could be stated that most modern programming languages split the role of *storing data* and *operating on data* into two distinct language features.

This chapter is all about methods of exploiting data while it is being stored in the form of objects and being operated on via functions. These techniques work against a multitude of modern programming languages, and they abuse the powerful side effects of storing data as first-class citizens.

Mass Assignment

The first and most common method of attacking first-class objects is that of the *mass assignment attack*. While typically referred to using the modern standardized terms *mass assignment attack* or *mass assignment vulnerability*, historical attacks also used the terminology *autobinding attack* (in the case of Spring MVC and ASP.NET frameworks) and *object injection*

attacks in the case of legacy PHP vulnerabilities. All of these terms refer to the same system of exploitation.

Mass assignment vulnerabilities allow attackers to change fields that were not intended to be changed, via passing additional fields within an object to a function call that does not validate keys. Although it sounds confusing, it will become very simple to spot after reviewing the examples in this section.

Mass assignment vulnerabilities typically are found within systems that pass around a lot of state-related data (in the form of objects) from one function to another. One good example of the type of application that may be vulnerable to mass assignment is a video game that passes around user state information on a regular basis. Another type of application that may be vulnerable is a web application that implements "dynamic forms," or forms that may have their title, data type, etc. changed by the end user. Either of these applications is likely to be attacked via mass assignment if not programmed with security in mind.

Let's take a minute to evaluate the following code snippets, which describe a common form of 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
 }
});
```

In this example of a mass assignment vulnerability, the video game *MegaGame* implements an API endpoint that is accessible via HTTP POST. This endpoint takes one parameter from the client (web browser) in the

form of a state object. Provided the state object is not null, the API endpoint calls a database helper library to update the current user's game state with the state provided by the client (on behalf of the user playing the game).

The `db.update()` function referenced in the API endpoint is written as follows:

```
const update = function(data) {
 for (const [key, value] of Object.entries(data)) {
   database.upsert({ `${key}`: `${value}` })
 }
}
```

Note that the issue (vulnerability) in this `update()` function is not the fact that it *upserts* (inserts if does not exist, updates if exists) client-provided data. Instead, the mass assignment vulnerability appears because the function does not validate any of the information provided by the client, in particular regarding the keys within the `data` object.

The assumption by the developer of this application is that the client will pass back a payload including the following data:

```
const data = {
 playerId: 123,
 playerPosition: { "x": 125, "y": 346 },
 playerHP: 90
};
```

Because the `update()` function lacks any validation and accepts the data as is from the client, a hacker can modify the payload en route or send a rogue payload including the following data:

```
const data = {
 playerId: 123,
 playerPosition: { "x": 125, "y": 346 },
 playerHP: 90,
```

```
  isAdmin: true // this is the "attack" portion of the payload
};
```

By modifying the payload to include the `isAdmin` key, the `update()` function will write or modify the `isAdmin` entry in the user's database row. Now, the user has elevated their in-game role to `admin` by abusing the fact that sanitization and validation did not occur within the `update()` function. Had the `update()` function validated the object's keys and sanitized the `isAdmin` key, no role update would have occurred as a result of this malicious payload hitting the `updatePlayerData` endpoint.

This is a perfect example of a mass assignment vulnerability, one of the most common forms of an object vulnerability—and one of the easiest to exploit.

Insecure Direct Object Reference

Insecure direct object reference attacks (IDOR) are a common web application vulnerability that was spotlighted in 2007 when it was added to the OWASP top 10 vulnerabilities list. Since then, IDOR vulnerabilities are less common on the web; however, due to their simple nature, they still appear from time to time and take very little skill to exploit.

An IDOR vulnerability occurs when objects on a server are directly accessible via user-supplied parameters, such as URL `query` parameters, HTTP POST BODY, or URL `:id` fields. By trusting the end user to supply the correct parameters to reference a specific object on a server, it's highly likely that other objects become accessible by the end user, which leads to privilege escalation.

Consider the following HTTP POST endpoint:

```
app.get('/files/:id', function(req, res, next) => {
  return res.sendFile(`/filesystem/files/${req.params.id}`);
});
```

This endpoint returns any file under */filesystem/files* given a filename denoted by `:id`. On the client side, the developer's expectation may be that a *specific* file is accessible to the end user. For example, the developer may intend for the end user to be able to download a file named *my-report-card.txt*.

In order to request the file *my-report-card.txt*, the end user's browser would make the following request:

```
HTTP GET https://mywebsite.com/files/my-report-card.txt
```

Attacking this IDOR is simple: just intercept the HTTP GET request in the browser and change the filename. Or make use of an HTTP proxy tool like Burp Suite or ZAP to intercept and change the request before it hits the server.

If the filename is changed to *other-report-card.txt*, the server will attempt to send back the file *other-report-card.txt* even if the current user is not expected to be able to access it. For these reasons, IDOR is a form of privilege escalation in addition to being an attack that targets objects via object references.

Serialization Attacks

When data is passed back and forth between browser clients, servers, and services, it is often impractical, expensive, and difficult to send in its raw format. To resolve this issue, many web applications perform *serialization* on raw data prior to sending it over the network.

Web Serialization Explained

An important key concept in serialization is that the serialized format should be easy to *deserialize*, or revert to its raw data form. In addition, it should be formatted in such a way that it can easily be stored on disk if needed.

An example of serialization would be taking an in-memory object from a browser's JavaScript execution context and converting it to JavaScript Object Notation (JSON). A JSON "blob" representing a serialized `farmer` class would look as follows:

```
{
  "name": "Joe Carrot",
  "age": 62,
  "location": "Montana, USA",
  "crops": {
    "wheat": {
      "measurement": "acres",
      "amount": 100
    }
  }
}
```

The data associated with the farmer "Joe Carrot" in this example can be easily stored on disk or transferred over a network. Whenever required, it can be deserialized and loaded into memory—creating new memory addresses and pointers and allowing a programming language to modify it.

Other popular formats for data serialization used by web applications include XML, YAML, and base64 (a format for serializing text in binary).

Attacking Weak Serialization

The first step toward attacking a web application via a serialization attack is to find a function where data serialization is performed in an application. The next step is to use that function with test data in order to compare and contrast the input and output of a set of serialization functions.

Consider the popular npm library *serialize-javascript*. This library has over 30 million weekly downloads and is used in tens of thousands of JavaScript-based applications. However, versions 3.0.9 and lower were vulnerable to code injection attacks using its `serialization()` function as the delivery mechanism for a payload.

In this case, inputting the payload `{"foo": /1"/, "bar": "a\"@__R-<UID>-0__@"}` into the `serialization()` function would result in the serialized JSON of `{"foo": /1"/, "bar": "a\/1"/}`. The structure of the resulting JSON object escapes quotes (which are an expected part of a proper JSON object) and as a result creates code execution if it ever runs through the `eval()` function.

A proof of concept such as `eval('('+ serialize({"foo": /1" + console.log(1)/i, "bar": '"@__R-<UID>-0__@'}) + ')');` would lead to code execution due to the serializer's inability to format this output string into properly escaped JSON.

All serialization attacks follow the same steps:

1. Find a function that performs serialization.
2. Read the function carefully or test it with common payloads.
3. After finding a failure to properly serialize data, create a payload capable of script execution.
4. Call the function with a payload capable of script execution.
5. Obtain remote code execution (server) or XSS (client).

Since most websites do not implement their own serializers, it is possible to find vulnerable serializers that are open source and scan for vulnerable applications and methods. This is an advanced attack technique, but it is one of the best ways of finding and exploiting serialization vulnerabilities in the real world.

Summary

Even though the web has evolved significantly over the years, web programming languages still make use of programming language design patterns that have existed for a long time. When you get to the point at which you begin to understand the systems underneath any complex application, it becomes obvious how these can be exploited at a fundamental level. The attacks in this chapter that target objects and data exist to show how underlying systems are often viable targets for hackers, even after decades of improvements in programming language design.