11

MASS ASSIGNMENT

An API is vulnerable to mass assignment if the consumer is able to send a request that updates or overwrites server-side variables. If an API accepts client input without filtering or sanitizing it, an attacker can update objects with which they shouldn't be able to interact. For example, a banking API might allow users to update the email address associated with their account, but a mass assignment vulnerability might let the user send a request that updates their account balance as well.

In this chapter, we'll discuss strategies for finding mass assignment targets and figuring out which variables the API uses to identify sensitive data. Then we'll discuss automating your mass assignment attacks with Arjun and Burp Suite Intruder.

Finding Mass Assignment Targets

One of the most common places to discover and exploit mass assignment vulnerabilities is in API requests that accept and process client input. Account registration, profile editing, user management, and client management are all common functions that allow clients to submit input using the API.

Account Registration

Likely the most frequent place you'll look for mass assignment is in account registration processes, as these might allow you to register as an administrative user. If the registration process relies on a web application, the end user would fill in standard fields with information such as their desired username, email address, phone number, and account password. Once the user clicks the submit button, an API request like the following would be sent:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"password":"Password1!"
}
```

For most end users, this request takes place in the background, leaving them none the wiser. However, since you're an expert at intercepting web application traffic, you can easily capture and manipulate it. Once you've intercepted a registration request, check whether you can submit additional values in the request. A common version of this attack is to upgrade an account to an administrator role by adding a variable that the API provider likely uses to identify admins:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"admin": true,
"password":"Password1!"
}
```

If the API provider uses this variable to update account privileges on the backend and accepts additional input from the client, this request will turn the account being registered into an admin-level account.

Unauthorized Access to Organizations

Mass assignment attacks go beyond making attempts to become an administrator. You could also use mass assignment to gain unauthorized access to other organizations, for instance. If your user objects include an organizational group that allows access to company secrets or other sensitive information, you can attempt to gain access to that group. In this example, we've added an `"org"` variable to our request and turned its value into an attack position we could then fuzz in Burp Suite:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"org": "§CompanyA§",
"password":"Password1!"
}
```

If you can assign yourself to other organizations, you will likely be able to gain unauthorized access to the other group's resources. To perform such an attack, you'll need to know the names or IDs used to identify the companies in requests. If the `"org"` value was a number, you could brute-force its value, like when testing for BOLA, to see how the API responds.

Do not limit your search for mass assignment vulnerabilities to the account registration process. Other API functions are capable of being vulnerable. Test other endpoints used for resetting passwords; updating account, group, or company

profiles; and any other plays where you may be able to assign yourself additional access.

Finding Mass Assignment Variables

The challenge with mass assignment attacks is that there is very little consistency in the variables used between APIs. That being said, if the API provider has some method for, say, designating accounts as administrator, you can be sure that they also have some convention for creating or updating variables to make a user an administrator. Fuzzing can speed up your search for mass assignment vulnerabilities, but unless you understand your target's variables, this technique can be a shot in the dark.

Finding Variables in Documentation

Begin by looking for sensitive variables in the API documentation, especially in sections focused on privileged actions. In particular, the documentation can give you a good indication of what parameters are included within JSON objects.

For example, you might search for how a low-privileged user is created compared to how an administrator account is created. Submitting a request to create a standard user account might look something like this:

```
POST /api/create/user
Token: LowPriv-User
--snip--
{
"username": "hapi_hacker",
"pass": "ff7ftw"
}
```

Creating an admin account might look something like the following:

```
POST /api/admin/create/user
Token: AdminToken
--snip--
{
"username": "adminthegreat",
"pass": "bestadminpw",
"admin": true
}
```

Notice that the admin request is submitted to an admin endpoint, uses an admin token, and includes the parameter `"admin": true`. There are many fields related to admin account creation, but if the application doesn't handle the requests properly, we might be able to make an administrator account by simply adding the parameter `"admin"=true` to our user account request, as shown here:

```
POST /create/user
Token: LowPriv-User
--snip--
{
"username": "hapi_hacker",
"pass": "ff7ftw",
"admin": true
}
```

Fuzzing Unknown Variables

Another common scenario is that you'll perform an action in a web application, intercept the request, and locate several bonus headers or parameters within it, like so:

```
POST /create/user
--snip--
{
```

```
"username": "hapi_hacker"
"pass": "ff7ftw",
"uam": 1,
"mfa": true,
"account": 101
}
```

Parameters used in one part of an endpoint might be useful for exploiting mass assignment using a different endpoint. When you don't understand the purpose of a certain parameter, it's time to put on your lab coat and experiment. Try fuzzing by setting `uam` to zero, `mfa` to false, and `account` to every number between 0 and 101, and then watch how the provider responds. Better yet, try a variety of inputs, such as those discussed in the previous chapter. Build up your wordlist with the parameters you collect from an endpoint and then flex your fuzzing skills by submitting requests with those parameters included. Account creation is a great place to do this, but don't limit yourself to it.

Blind Mass Assignment Attacks

If you cannot find variable names in the locations discussed, you could perform a blind mass assignment attack. In such an attack, you'll attempt to brute-force possible variable names through fuzzing. Send a single request with many possible variables, like the following, and see what sticks:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
"admin": true,
"admin":1,
"isadmin": true,
"role":"admin",
"role":"administrator",
```

```
"user_priv": "admin",
"password":"Password1!"
}
```

If an API is vulnerable, it might ignore the irrelevant variables and accept the variable that matches the expected name and format.

Automating Mass Assignment Attacks with Arjun and Burp Suite Intruder

As with many other API attacks, you can discover mass assignment by manually altering an API request or by using a tool such as Arjun for parameter fuzzing. As you can see in the following Arjun request, we've included an authorization token with the `–headers` option, specified JSON as the format for the request body, and identified the exact attack spot that Arjun should test with `$arjun$`:

```
$ arjun --headers "Content-Type: application/json]" -u http://vulnhost.com/api/register -m JSON --include=

[~] Analysing the content of the webpage
[~] Analysing behaviour for a non-existent parameter
[!] Reflections: 0
[!] Response Code: 200
[~] Parsing webpage for potential parameters
[+] Heuristic found a potential post parameter: admin
[!] Prioritizing it
[~] Performing heuristic level checks
[!] Scan Completed
[+] Valid parameter found: user
[+] Valid parameter found: pass
[+] Valid parameter found: admin
```

As a result, Arjun will send a series of requests with various parameters from a wordlist to the target host. Arjun will then narrow down likely parameters based

on deviations of response lengths and response codes and provide you with a list of valid parameters.

Remember that if you run into issues with rate limiting, you can use the Arjun `--stable` option to slow down the scans. This sample scan completed and discovered three valid parameters: `user`, `pass`, and `admin`.

Many APIs prevent you from sending too many parameters in a single request. As a result, you might receive one of several HTTP status codes in the 400 range, such as 400 Bad Request, 401 Unauthorized, or 413 Payload Too Large. In that case, instead of sending a single large request, you could cycle through possible mass assignment variables over many requests. This can be done by setting up the request in Burp Suite's Intruder with the possible mass assignment values as the payload, like so:

```
POST /api/v1/register
--snip--
{
"username":"hAPI_hacker",
"email":"hapi@hacker.com",
§"admin": true§,
"password":"Password1!"
}
```

Combining BFLA and Mass Assignment

If you've discovered a BFLA vulnerability that allows you to update other users' accounts, try combining this ability with a mass assignment attack. For example, let's say a user named Ash has discovered a BFLA vulnerability, but the vulnerability only allows him to edit basic profile information such as usernames, addresses, cities, and regions:

```
PUT /api/v1/account/update
Token:UserA-Token
--snip--
{
"username": "Ash",
"address": "123 C St",
"city": "Pallet Town"
"region": "Kanto",
}
```

At this point, Ash could deface other user accounts, but not much more. However, performing a mass assignment attack with this request could make the BFLA finding much more significant. Let's say that Ash analyzes other GET requests in the API and notices that other requests include parameters for email and multifactor authentication (MFA) settings. Ash knows that there is another user, named Brock, whose account he would like to access.

Ash could disable Brock's MFA settings, making it easier to gain access to Brock's account. Moreover, Ash could replace Brock's email with his own. If Ash were to send the following request and get a successful response, he could gain access to Brock's account:

```
PUT /api/v1/account/update
Token:UserA-Token
--snip--
{
"username": "Brock",
"address": "456 Onyx Dr",
"city": "Pewter Town",
"region": "Kanto",
"email": "ash@email.com",
"mfa": false
}
```

Since Ash does not know Brock's current password, Ash should leverage the API's process for performing a password reset, which would likely be a PUT or POST request sent to */api/v1/account/reset*. The password reset process would then send a temporary password to Ash's email. With MFA disabled, Ash would be able to use the temporary password to gain full access to Brock's account.

Always remember to think as an adversary would and take advantage of every opportunity.

Summary

If you encounter a request that accepts client input for sensitive variables and allows you to update those variables, you have a serious finding on your hands. As with other API attacks, sometimes a vulnerability may seem minor until you've combined it with other interesting findings. Finding a mass assignment vulnerability is often just the tip of the iceberg. If this vulnerability is present, chances are that other vulnerabilities are present.

Lab #8: Changing the Price of Items in an Online Store

Armed with our new mass assignment attack techniques, let's return to crAPI. Consider what requests accept client input and how we could leverage a rogue variable to compromise the API. Several of the requests in your crAPI Postman collection appear to allow client input:

```
POST /identity/api/auth/signup
POST /workshop/api/shop/orders
POST /workshop/api/merchant/contact_mechanic
```

It's worth testing each of these once we've decided what variable to add to them.

We can locate a sensitive variable in the GET request to the */workshop/api/shop/products* endpoint, which is responsible for populating the crAPI storefront with products. Using Repeater, notice that the GET request loads a JSON variable called `"credit"` (see *Figure 11-1*). That seems like an interesting variable to manipulate.

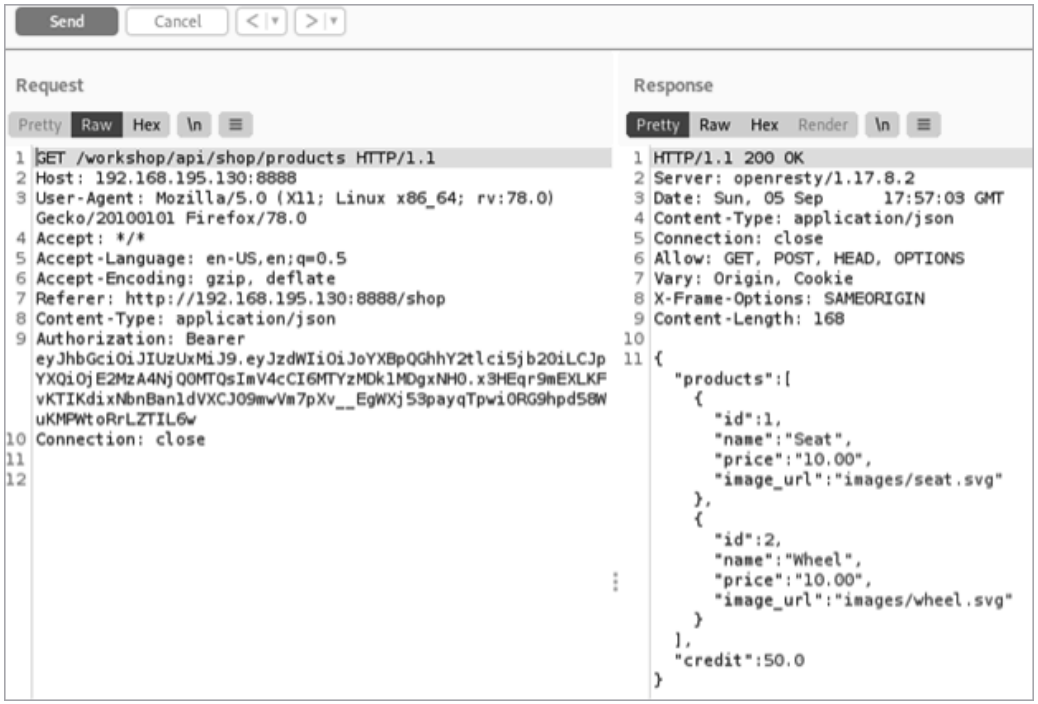

Figure 11-1: Using Burp Suite Repeater to analyze the /workshop/api/shop/products endpoint

This request already provides us with a potential variable to test (`credit`), but we can't actually change the credit value using a GET request. Let's run a quick Intruder scan to see if we can leverage any other request methods with this endpoint. Right-click the request in Repeater and send it to Intruder. Once in Intruder, set the attack position to the request method:

```
§GET§ /workshop/api/shop/products HTTP/1.1
```

Let's update the payloads with the request methods we want to test for: PUT, POST, HEAD, DELETE, CONNECT, PATCH, and OPTIONS (see *Figure 11-2*).

Start the attack and review the results. You'll notice that crAPI will respond to restricted methods with a 405 Method Not Allowed status code, which means the 400 Bad Request response we received in response to the POST request is pretty interesting (see *Figure 11-3*). This 400 Bad Request likely indicates that crAPI is expecting a different payload to be included in the POST request.

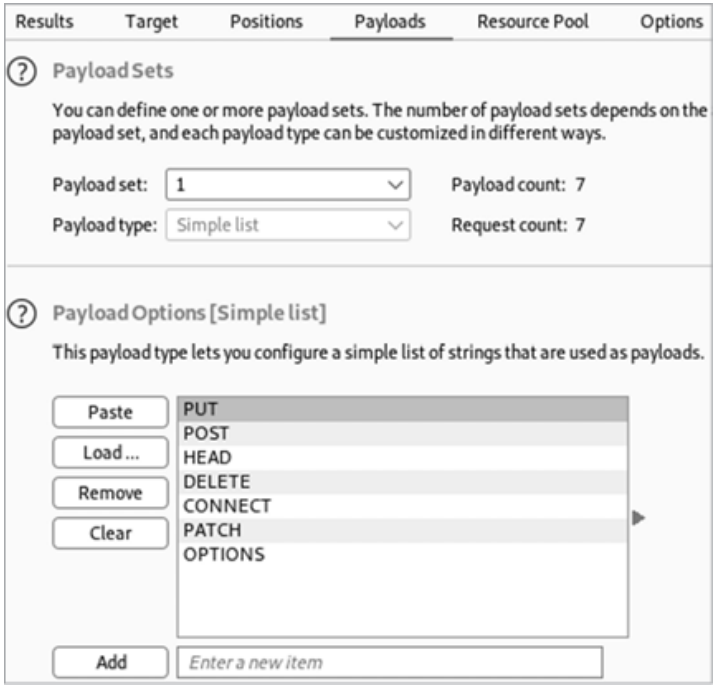

Figure 11-2: Burp Suite Intruder request methods with payloads

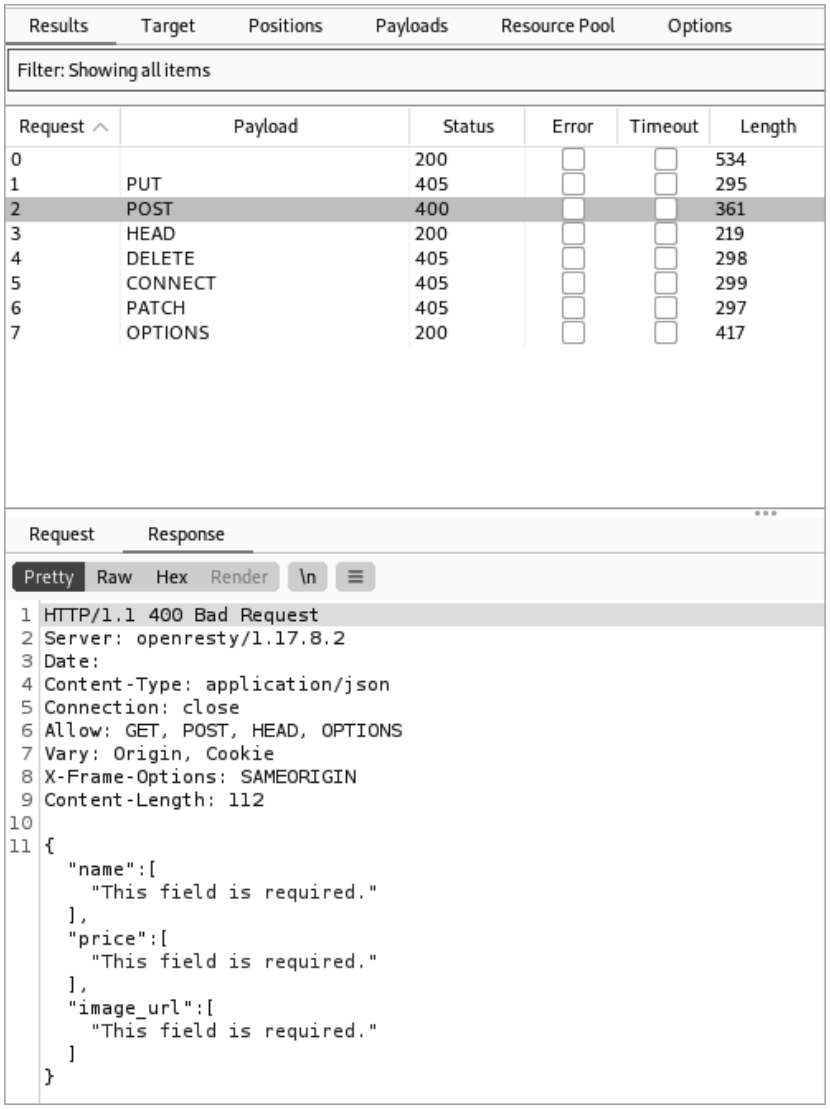

Figure 11-3: Burp Suite Intruder results

The response tells us that we've omitted certain required fields from the POST request. The best part is the API tells us the required parameters. If we think it through, we can guess that the request is likely meant for a crAPI administrator to use in order to update the crAPI store. However, since this request is not restricted to administrators, we have likely stumbled across a combined mass as-

signment and BFLA vulnerability. Perhaps we can create a new item in the store and update our credit at the same time:

```
POST /workshop/api/shop/products HTTP/1.1

Host: 192.168.195.130:8888
Authorization: Bearer UserA-Token

{
"name":"TEST1",
"price":25,
"image_url":"string",
"credit":1337
}
```

This request succeeds with an HTTP 200 OK response! If we visit the crAPI store in a browser, we'll notice that we successfully created a new item in the store with a new price of 25, but, unfortunately, our credit remains unaffected. If we purchase this item, we'll notice that it automatically subtracts that amount from our credit, as any regular store transaction should.

Now it's time to put on our adversarial hat and think through this business logic. As the consumer of crAPI, we shouldn't be able to add products to the store or adjust prices . . . but we can. If the developers programmed the API under the assumption that only trustworthy users would add products to the crAPI store, what could we possibly do to exploit this situation? We could give ourselves an extreme discount on a product—maybe a deal so good that the price is actually a negative number:

```
POST /workshop/api/shop/products HTTP/1.1

Host: 192.168.195.130:8888
Authorization: Bearer UserA-Token
```

```
{
"name":"MassAssignment SPECIAL",
"price":-5000,
"image_url":"https://example.com/chickendinner.jpg"
}
```

The item `MassAssignment SPECIAL` is one of a kind: if you purchase it, the store will pay you 5,000 credits. Sure enough, this request receives an HTTP 200 OK response. As you can see in *Figure 11-4*, we have successfully added the item to the crAPI store.

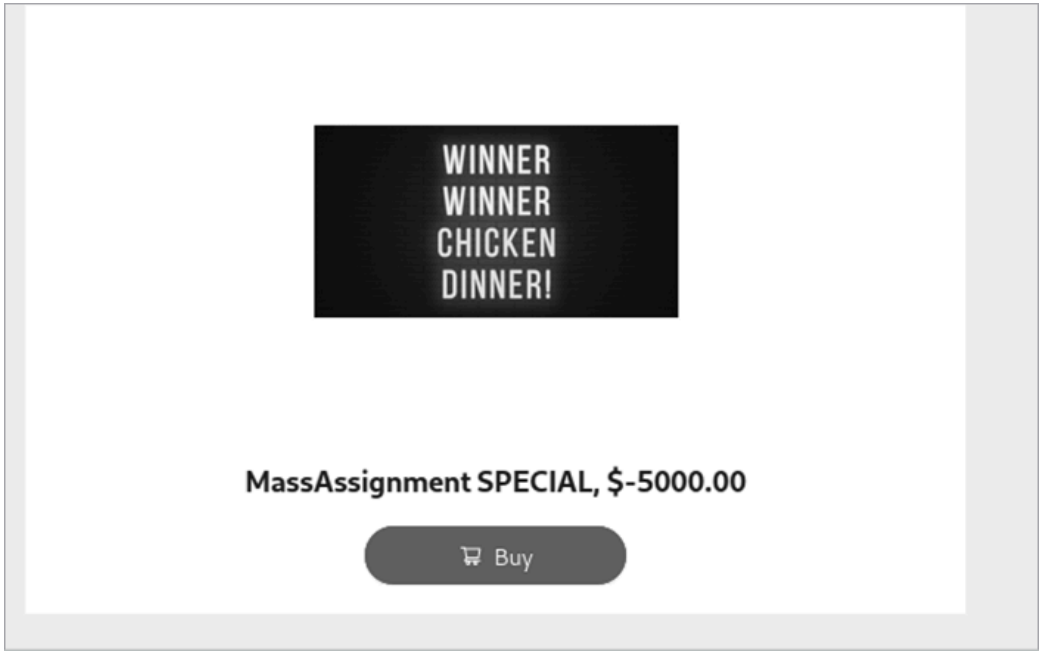

Figure 11-4: The MassAssignment SPECIAL on crAPI

By purchasing this special deal, we add an extra $5,000 to our available balance (see *Figure 11-5*).

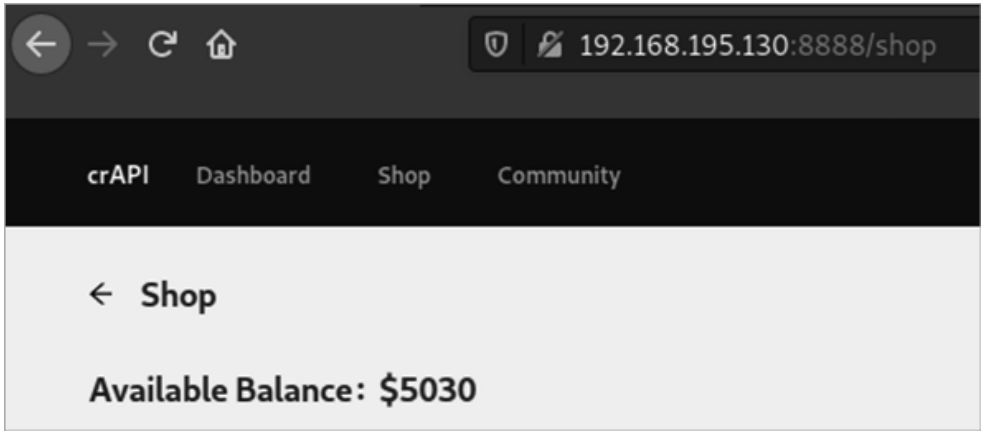

Figure 11-5: Available balance on crAPI

As you can see, our mass assignment exploit would have severe consequences for any business with this vulnerability. I hope your bounty for such a finding greatly outweighs the credit you could add to your account! In the next chapter, we'll begin our journey through the wide variety of potential injection attacks we can leverage against APIs.