**Exploitation Remarks & Prevention**

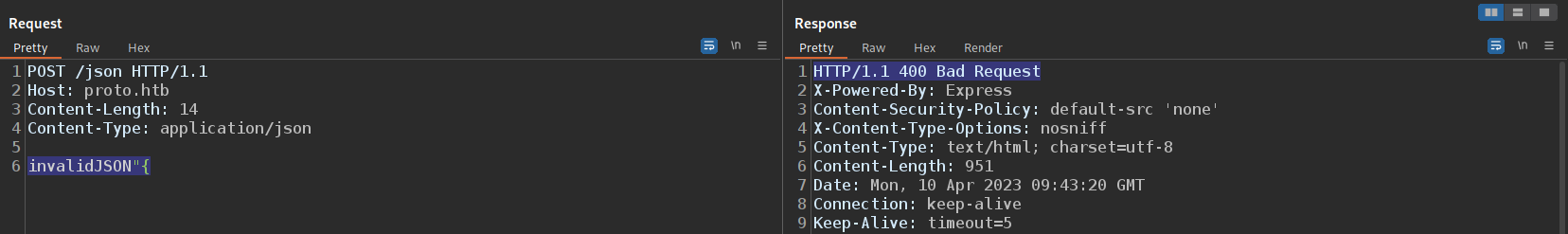
Now that we have examined and exploited various prototype pollution vulnerabilities, let us discuss some remarks regarding identifying and exploiting them in real-world engagements. Additionally, this section will end with how to prevent prototype pollution vulnerabilities.

**Exploitation Remarks**

As discussed previously, polluting prototypes can result in unforeseen and undesired side effects that potentially break the entire web application. Therefore, it is ill-advised to throw prototype pollution payloads on a production web application and hope for the best. Testing and fine-tuning the prototype pollution payload is recommended on a local copy of the target web application. While we focused on detecting and exploiting prototype pollution vulnerabilities from a whitebox approach in this module, we do not always have access to the source code of the target web application and need to be able to identify them black-box. Fortunately, a few techniques exist to detect prototype pollution using a black-box approach as safely as possible.

**Status Code**

The first and most universal technique is manipulating the status code returned when the web application encounters an issue. First, we need to determine how the web application reacts if we provide an invalid JSON request body:



The web application responds with an HTTP 400 status code. To confirm prototype pollution, we can manipulate the returned status code by polluting the status property of the Object.prototype object using a payload similar to the following:

Code: json

{

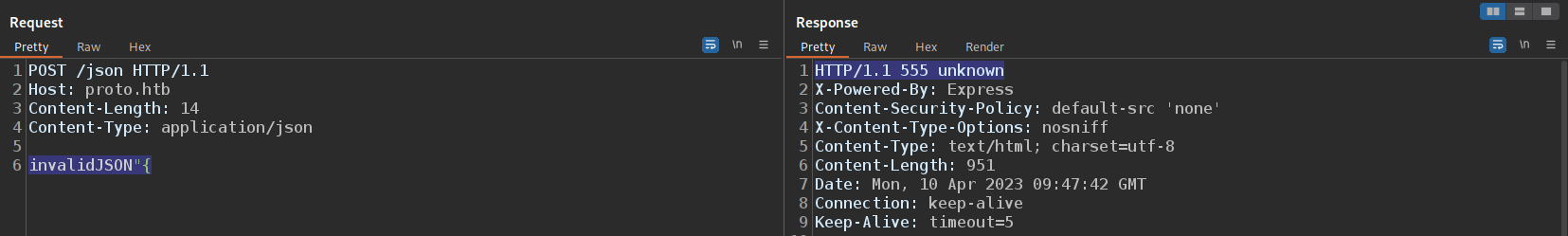
"\_\_proto\_\_":{

"status":555

}

}

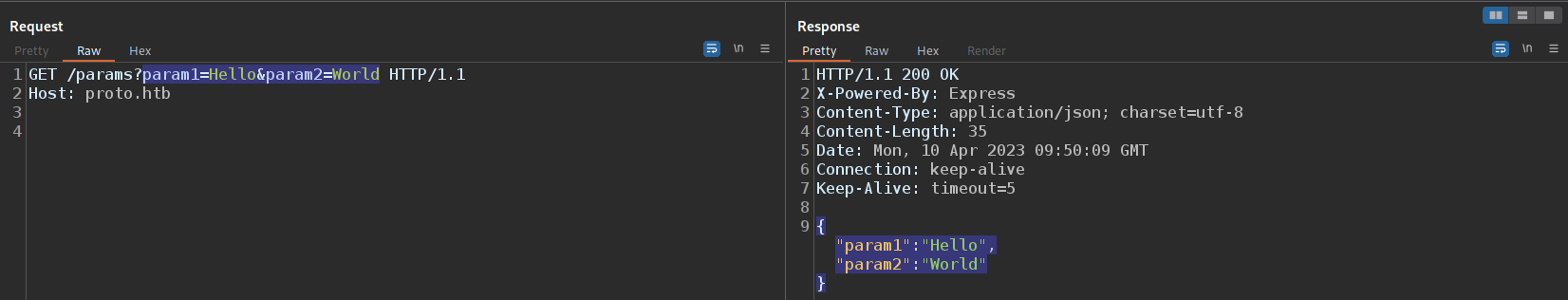
Depending on the web application's implementation, we might need to traverse multiple steps up the prototype chain to reach the Object.prototype object. When we now send the above request again, the server returns the custom-set status code 555:



Thus, we successfully confirmed prototype pollution. We can utilize this technique universally as it does not require any reflection of user input.

**Parameter Limiting**

The second technique requires that the web application contains an endpoint that reflects GET parameters in any way. In our simple example below, the response body reflects the GET parameters in a JSON object:



We can manipulate the number of GET parameters returned by the web application by polluting the parameterLimit property of the Object.prototype object using a payload similar to the following:

Code: json

{

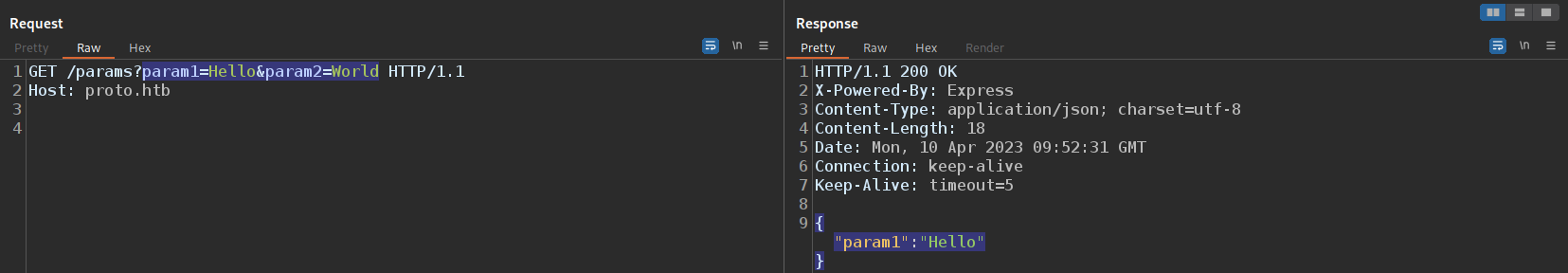
"\_\_proto\_\_":{

"parameterLimit":1

}

}

When we send the above request again, the web application responds with only the first GET parameter since we limited the number of parameters to one. Thus, all parameters after the first one are ignored:



Therefore, we successfully confirmed prototype pollution. We can only utilize this technique if the target web application provides an endpoint that reflects GET parameters.

**Content-Type**

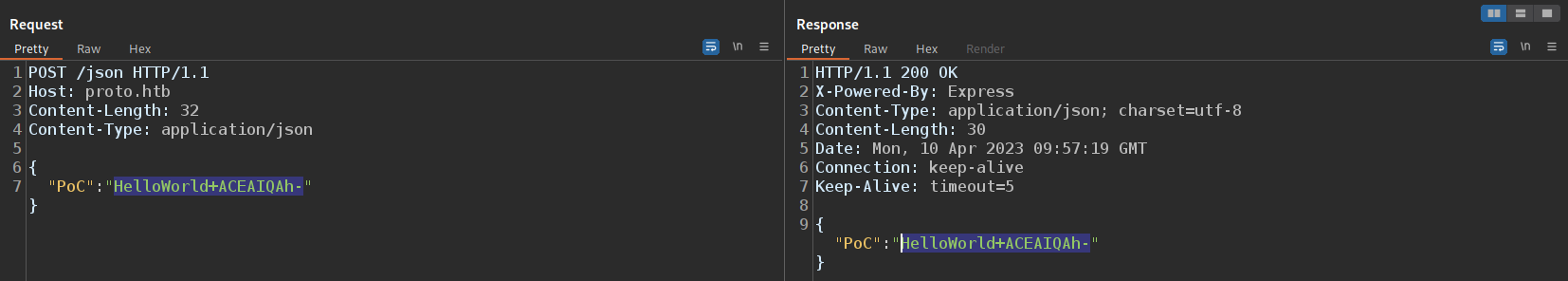
Our last example requires the reflection of a JSON object. We can force the web application to accept other encodings without breaking the web application. We will use the UTF-7 encoding for this since it does not break the web application's default UTF-8 encoding. First, we need to encode a test string in UTF-7, which we can do using iconv:

Exploitation Remarks & Prevention

yovecio@htb[/htb]$ echo -n 'HelloWorld!!!' | iconv -f UTF-8 -t UTF-7

HelloWorld+ACEAIQAh-

If we send the test string to the web application, it is reflected as-is. In particular, it was not UTF-7 decoded:



We can manipulate the value of the Content-Type Header used by the web application by polluting the content-type property of the Object.prototype object using a payload similar to the following:

Code: json

{

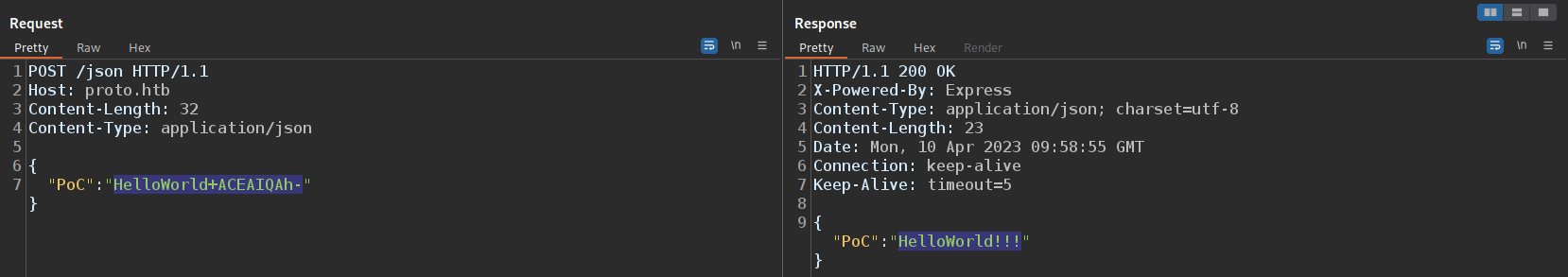
"\_\_proto\_\_":{

"content-type":"application/json; charset=utf-7"

}

}

When we now send the above request again, the web application accepts the UTF-7 encoding as well, such that the test string is decoded to display the exclamation marks in the response:



Thus, we successfully confirmed prototype pollution. We can only utilize this technique if the target web application provides an endpoint that reflects JSON input.

For more details on black-box detection of prototype pollution without breaking the web application, take a look at [this](https://portswigger.net/research/server-side-prototype-pollution) paper.