Appendix C

CompuRacer toolset – Manual

The manual contains a guide on how to use the toolset for exploiting race conditions. It covers using the extensions to add requests to the Core, the composing of batches of requests in different modes, sending the batches and interpreting the results. Throughout this process, some powerful built-in view and compare tools are also used to support the process. The manual assumes both the toolset and the test web app have been installed successfully according to the README.[1]

Command formatting In the manual, every command is shown on a new line and can be used in the Command Line Interface (CLI) of the CompuRacer Core. As already mentioned before, the terminal of a MacBook Pro running macOS High Sierra is used. It is configured to look similar to the JetBrains PyCharm terminal using the 'Homebrew' theme with the colour of `Text` and `Bold Text` set to white. If a command argument of a string-like type contains a space, for instance when it is a name, the argument must be enclosed in double quotes (""). When an argument is a boolean, the true and false values can be abbreviated to 't' and 'f'. Commands used in this manual are formatted as follows:

```
racer> command (abbreviation) <required argument> [optional
    ↪ argument]
```

In this manual, we will not explain every part of the functionality exhaustively. Instead, the manual will go through most functionality just like a tester would when he is using the tool. For this to be exactly reproducible, we take the included web

[1] As already indicated in the README enclosed with the toolset, the Chrome extension cannot send some headers anymore, and this breaks most security testing activities. That is why will only use Firefox in this manual.

app for voucher redeeming as a concrete example for all detection and exploitation related functionality. Some commands have different behaviours defined when optional arguments are provided or omitted. The help command can be used to view the details of commands and their behaviour or to search for a command:

```
racer> help [search term]
```

C.1 How to add HTTP requests of interest to the tool?

In this section, we will show how to add HTTP requests to the tool from the extensions, or by manually adding a batch to the tool by creating a JSON file. Last, we will explain how to use the Core action modes that help make the testing process more efficient. In order to view stored requests, the tester can use the first command below. If the tester wants to remove one or more requests (all ids between the first and second id) from the complete list, use the second command below:

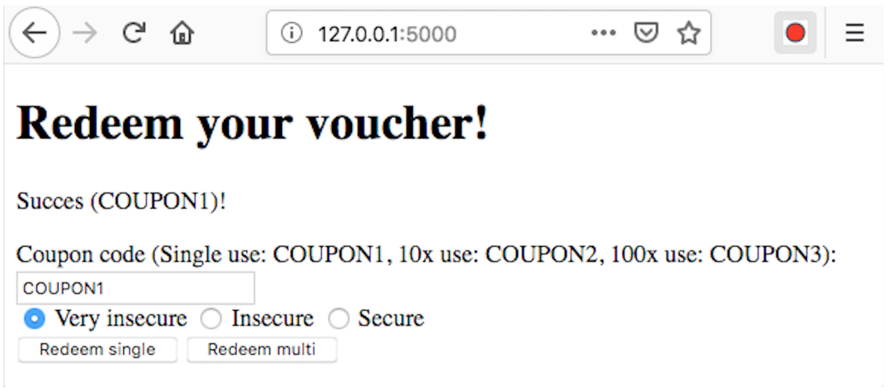
```
racer> reqs
racer> rm reqs [first request id] [last request id]
```

C.1.1 Send it from the browser using the Firefox extension

One option for adding requests to the Core is by using the Firefox extension. When the Firefox extension is loaded, it will show a grey circle as its icon as long as it is not connected to the CompuRacer Core. You can click the icon to try to reconnect. When it shows a white circle, the connection is made successfully. If we click it again, it will show a red circle for three seconds. During this time, any request of interest that is sent to the domain of the current tab is forwarded to the CompuRacer Core. When the tester hovers over the button, the extension state is also listed in a tool-tip.

Example We want to add the POST request of redeeming `COUPON1` in the very insecure way to the CompuRacer Core. In figure C.1, you can see the web app just after activating the extension (see red circle).

After this, we click the `Redeem single` button. This sends the request to the CompuRacer Core. When it is received successfully, we will get output similar to the cyan



Redeem your voucher!

Succes (COUPON1)!

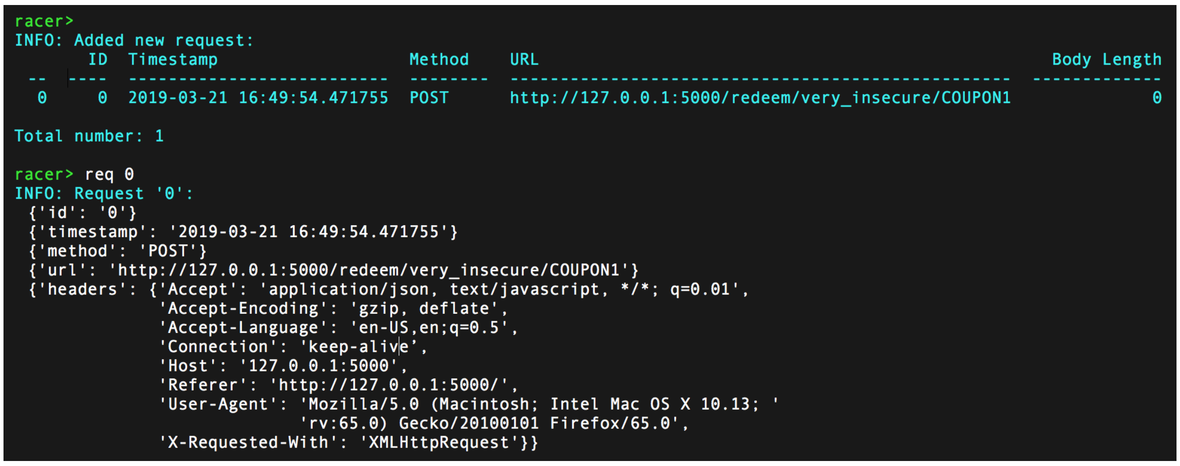
Coupon code (Single use: COUPON1, 10x use: COUPON2, 100x use: COUPON3):

COUPON1

[✓ Very insecure] Insecure Secure

Figure C.1: *The figure shows the activated Firefox extension in the test web app just before redeeming a voucher.*

text in figure C.2 that shows the summary of the request. Then, in the same figure, the command `req 0` is executed which shows the details of the request with id '0'.



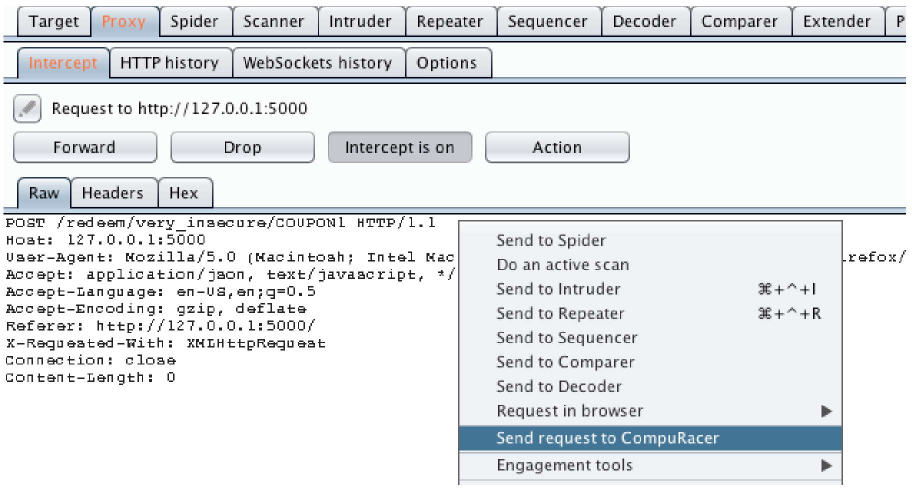
```
racer>
INFO: Added new request:
        ID  Timestamp                   Method    URL                                                 Body Length
  --  ----  --------------------------  --------  --------------------------------------------------  -------------
   0     0  2019-03-21 16:49:54.471755  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1              0

Total number: 1

racer> req 0
INFO: Request '0':
 {'id': '0'}
 {'timestamp': '2019-03-21 16:49:54.471755'}
 {'method': 'POST'}
 {'url': 'http://127.0.0.1:5000/redeem/very_insecure/COUPON1'}
 {'headers': {'Accept': 'application/json, text/javascript, */*; q=0.01',
              'Accept-Encoding': 'gzip, deflate',
              'Accept-Language': 'en-US,en;q=0.5',
              'Connection': 'keep-alive',
              'Host': '127.0.0.1:5000',
              'Referer': 'http://127.0.0.1:5000/',
              'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; '
                            'rv:65.0) Gecko/20100101 Firefox/65.0',
              'X-Requested-With': 'XMLHttpRequest'}}
```

Figure C.2: *The figure shows the output of the CompuRacer core when a new request is added.*

C.1.2 Send it from the Burp Suite using the Burp extension

The second option for adding requests is by using the Burp extension. We first need to configure the Burp proxy to capture all requests as we often do not want the original request to arrive at the target server. When the request of interest is sent, it will first appear in the Proxy tab, and then we can forward it to the CompuRacer Core using a button in the context menu. This button will be greyed-out when it is busy sending requests or when the Core not available. This is indicated clearly.

Example We want to add the same POST request as earlier to the CompuRacer Core, but now using the Burp extension. We set the proxy in capturing mode and in the web app, we click the button to redeem the voucher. After this, it pops up at the proxy tab. By selecting it, we can then click the button in the context menu to forward it to the CompuRacer Core as figure C.3 shows. The tester can also select multiple requests from any part of the Burp Suite interface and send these via the extension to the Core.

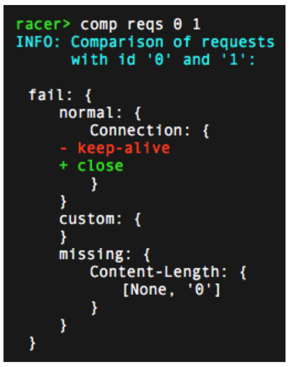


```
POST /redeem/very_insecure/COUPON1 HTTP/1.1
Host: 127.0.0.1:5000
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac                          refox/
Accept: application/json, text/javascript, */
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://127.0.0.1:5000/
X-Requested-With: XMLHttpRequest
Connection: close
Content-Length: 0
```

Figure C.3: *The figure shows the Burp Suite after capturing a request. From the context menu,we can decide to forward it.*

After this, we then receive the request in the CompuRacer Core. We expected to receive a duplicate request, and this should be rejected, but it was not the case, and we got a message equivalent to the one in figure C.2. When two requests appear to be the same, but the tester does not want to check the details line-by-line, the built-in compare function can be used. It compares every header field and the body of two requests and displays the changes. As you can see in figure C.4, we executed the command `comp reqs 0 1` and got a precise indication of the differences.

The command works as follows. When a header is present in both requests but the values differ based on a string-comparison, it is added to the 'normal' differences. When instead, a custom compare function is used on header values, and this fails, it is added to the 'custom' differences. Finally, when the header is missing altogether from one of the requests, it is added to the 'missing' differences. Appar-

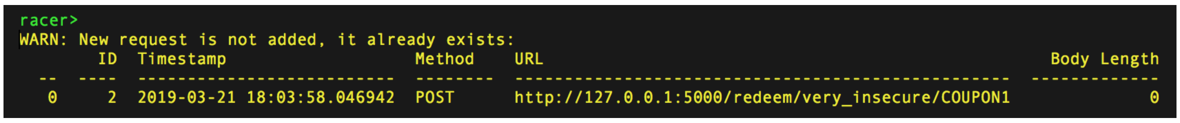


```
racer> comp reqs 0 1
INFO: Comparison of requests
      with id '0' and '1':

 fail: {
     normal: {
         Connection: {
        - keep-alive
        + close
         }
     }
     custom: {
     }
     missing: {
         Content-Length: {
             [None, '0']
         }
     }
 }
```

Figure C.4: *The figure shows the output of the CompuRacer core when comparing the requests from Firefox (id=0) and Burp (id=1). Only the 'Connection' and 'Content-Length' headers differ between them.*

ently, the Burp Suite adds a `Content-Length` header with a value of zero to all POST requests even when they do not have a body. Also, it changes the `Connection` header value from 'keep-alive' to 'close'.

Finally, to show what would happen when a duplicate request would arrive, we send the request again from Burp to the Core. The resulting message is shown in figure C.5.



```
racer>
WARN: New request is not added, it already exists:
       ID  Timestamp                   Method    URL                                                Body Length
  --  ----  --------------------------  --------  -------------------------------------------------  -------------
   0     2  2019-03-21 18:03:58.046942  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1              0
```

Figure C.5: *The figure shows the output of the CompuRacer core when a duplicate request is added.*

C.1.3 Add it manually using the correct JSON format

The third option is to add a request manually. This is not officially supported, but still possible. As it alters the `state.json` JSON file, the CompuRacer Core should not be running when adding a request in this way. In this file, the current settings of the Core are stored alongside the complete list of requests. When we would like to add a request, the request headers and content should be added to the dictionary-value of the `requests` key with a request key that is unique.

Example For this final example, we want to add almost the same HTTP request as in the examples earlier. The only changes are in the following HTTP headers: `Connection` will have the value `close` instead of `keep-alive`, and `Accept-Language` is changed from English to Dutch. It will get id '1' as '0' is already taken by the request added earlier. All of this can be done by adding the following entry to this JSON file as shown in listing 2.

C.2 How to compose a batch of HTTP requests?

In this section, we will show how to add stored HTTP requests to a batch. A batch is a collection of requests that are sent with several individually defined settings. The request can be sent after a specific delay (in ms) from the start of sending the batch, and it can be sent multiple times in parallel and in sequence. This provides for almost unlimited options to prepare combinations of requests to trigger race conditions.

```
"1": {
    "body": "",
    "headers": {
        "Accept": "application/json, text/javascript, */*; q=0.01",
        "Accept-Encoding": "gzip, deflate",
        "Accept-Language": "nl-NL,nl;q=0.5",
        "Connection": "close",
        "Content-Length": "0",
        "Host": "127.0.0.1:5000",
        "Referer": "http://127.0.0.1:5000/",
        "User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13;
         ↪  rv:65.0) Gecko/20100101 Firefox/65.0",
        "X-Requested-With": "XMLHttpRequest"
    },
    "method": "POST",
    "timestamp": "2019-03-21 18:01:43.019910",
    "url": "http://127.0.0.1:5000/redeem/very_insecure/COUPON1",
    "id": "1"
},
```

Listing 2: *Adding a request manually to the CompuRacer Core state file: 'state.json'*

Batches can be created from the CLI, and be filled with requests manually or automatically based on newly added requests. Batches that have been exported can also be imported back into the tool. Finally, batches can be added manually using JSON files similar to how requests were added. The tester can list all stored batches using the first command below. The second command can be used to remove one or more batches (all ids between the first and second id) from the complete list of batches. The third command is used to remove a request from the current batch by id and delay:

```
racer> batches (bss)
racer> rm batches (bss) [first batch id] [last batch id]
racer> rm [request id] [delay]
```

C.2.1 Creating it manually and adding requests

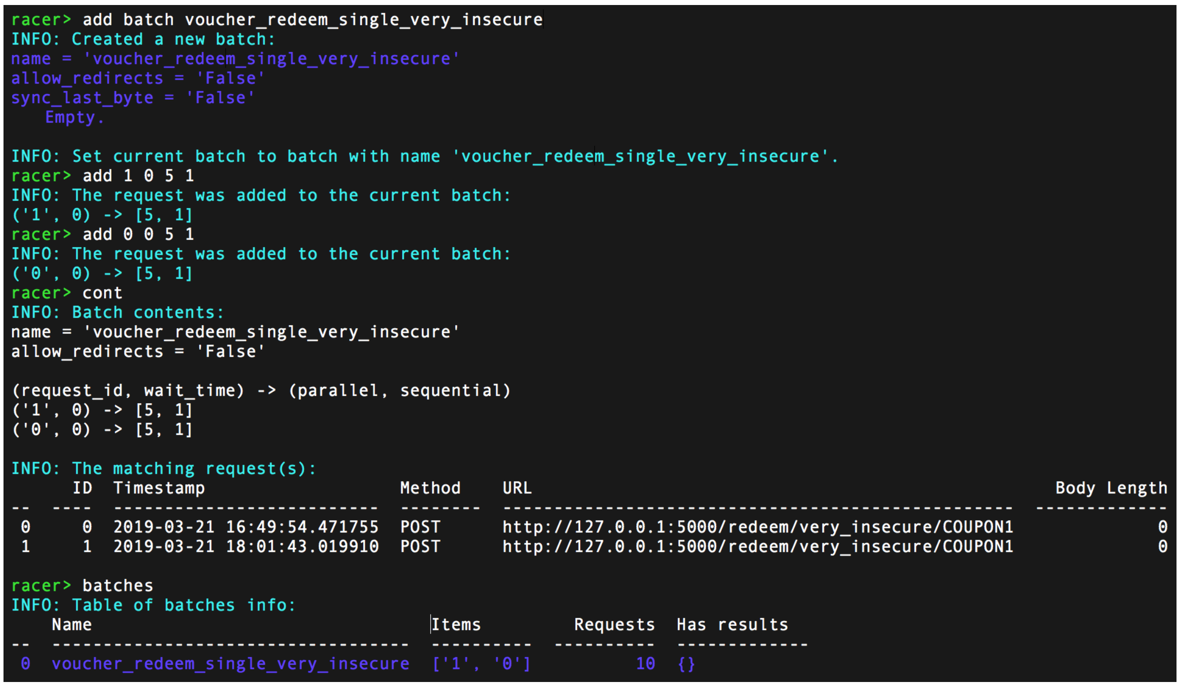
The first option is to add a batch and its requests via the CLI. We create a batch with the name `voucher_redeem_single_very_insecure` using the following command:

```
racer> add batch (bss) <batch name>
```

This does not only create a batch, but it also makes it our 'current batch'. This is the one specific batch that is selected to be edited, viewed and sent easily. Access to other batches requires more effort (longer command, and requires you to provide the batch id) and these batches cannot be edited. The current batch is marked in the listing of all batches. The current batch can be changed using the first command below. The new batch does not hold any requests yet. We can add the two requests from the previous section with ids '0' and '1', no delay, 5x parallel and no sequential duplication using the second command. Finally, the contents of the current batch can be shown using the third command:

```
racer> set curr <batch id>
racer>
    add <request id> [delay] [num parallel] [num sequential]
racer> cont
```

Figure C.6 shows the whole process of creating a batch, adding requests to the batch and listing the contents.



```
racer> add batch voucher_redeem_single_very_insecure
INFO: Created a new batch:
name = 'voucher_redeem_single_very_insecure'
allow_redirects = 'False'
sync_last_byte = 'False'
    Empty.

INFO: Set current batch to batch with name 'voucher_redeem_single_very_insecure'.
racer> add 1 0 5 1
INFO: The request was added to the current batch:
('1', 0) -> [5, 1]
racer> add 0 0 5 1
INFO: The request was added to the current batch:
('0', 0) -> [5, 1]
racer> cont
INFO: Batch contents:
name = 'voucher_redeem_single_very_insecure'
allow_redirects = 'False'

(request_id, wait_time) -> (parallel, sequential)
('1', 0) -> [5, 1]
('0', 0) -> [5, 1]

INFO: The matching request(s):
      ID  Timestamp                   Method    URL                                                  Body Length
--  ----  --------------------------  --------  ---------------------------------------------------  -------------
 0     0  2019-03-21 16:49:54.471755  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1               0
 1     1  2019-03-21 18:01:43.019910  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1               0

racer> batches
INFO: Table of batches info:
    Name                                 Items        Requests  Has results
--  -----------------------------------  ----------  ----------  -------------
 0  voucher_redeem_single_very_insecure  ['1', '0']          10  {}
```

Figure C.6: *The figure shows creating a new batch via the CLI, adding two requests and showing the contents.*

C.2.2 Creating a batch using the automated modes

The CompuRacer Core can be set to three different action modes that control whether to create batches automatically when a request is added to the Core via the REST server. The tester can set the mode using the first command below and change the mode settings using the second command:

```
racer> mode [on/curr/off]
racer> set mode [num parallel] [num sequential]
```

Next, we will explain in detail how every mode influences the behaviour of the Core after adding requests.

- **On** – In this mode, all requests that are received within a 3-second interval together will be added to a special batch with the name 'Imm'. The mode uses its own sequential and parallel duplication settings to add the requests. Then, the batch is sent automatically. If the batch already exists, its contents and results are overwritten. Note that this batch can only be viewed to avoid interference and race conditions ;), but not be altered or sent by the user himself. If more control is required or the user wants to preserve the contents, it must be copied to a new user-controlled batch. It is useful to quickly test forwarded requests without requiring any user interaction.
- **Curr** – In this mode, all requests that are received are added to the current batch. It uses the same sequential, and parallel duplication settings as the 'on' mode does. In contrast to the 'on' mode, will not send the batch automatically. As the normal flow of activities always includes creating a batch and adding stored requests, this mode removes the second step and makes the process more user-friendly.
- **Off** – In this mode, no additional action will be taken when a request is received.

Note: although duplicate requests will not be added to the total request list, they will trigger the mode actions with the current request as an argument.

C.2.3 Add it manually using the correct JSON format

In the `state/batches/` folder, all batches of the Core are stored in their own JSON file. These batches are only loaded at startup, so after adding a batch using this method, the CompuRacer Core must be restarted for it to take effect.

Example Let us say we want to add a batch called 'get_voucher_page' that only contains the GET request for the main voucher page. This GET request has id '1'. Finally, it should be sent after a delay of 100ms for five times in parallel and two times sequentially. This can be done by adding a JSON file with contents as shown in listing 3. The other keys in the batch JSON file are used for the batch-specific settings:

- **allow_redirects** - if this value is true, when the batch is sent, it will follow redirect (302) HTTP responses.
- **custom_comparing** - this would contain a dictionary of HTTP headers that should be ignored or be compared differently when making result groups. Result groups are explained later.
- **results** - this would contain the results when the batch has been sent.

```json
{
    "name": "get_voucher_page",
    "items": [
        {
            "key": ["1", 100],
            "value": [5, 2]
        }
    ],
    "allow_redirects": false,
    "custom_comparing": null,
    "results": null
}
```

Listing 3: *JSON file of a CompuRacer batch: 'get_voucher_page.json'*

C.2.4 Import an exported batch

The final option to get batches into the Core is to import already exported batches. Just like all other storage, these are JSON files as well. By default, they are stored in the `CompuRacer_Core/exp_files/` folder. Contrary to the files that are used to store the internal batches, exported batches contain an extra key `requests` which contains the contents of all requests that are referenced in the batch contents or the results. The requests in the contents and the results might be different when requests are added to a batch after it has been sent. Adding the requests makes it possible to import and use a batch just like it was meant to be, even when the matching requests have been changed or removed from the Core in the meantime. Only unique requests are imported back again. As requests contain a unique id, when a request is imported, it gets a new id that is the highest request id plus one. The tester can export one or more batches (all ids between the first and second id) using the following command:

```
racer> exp batches (bss) [first batch id] [last batch id]
```

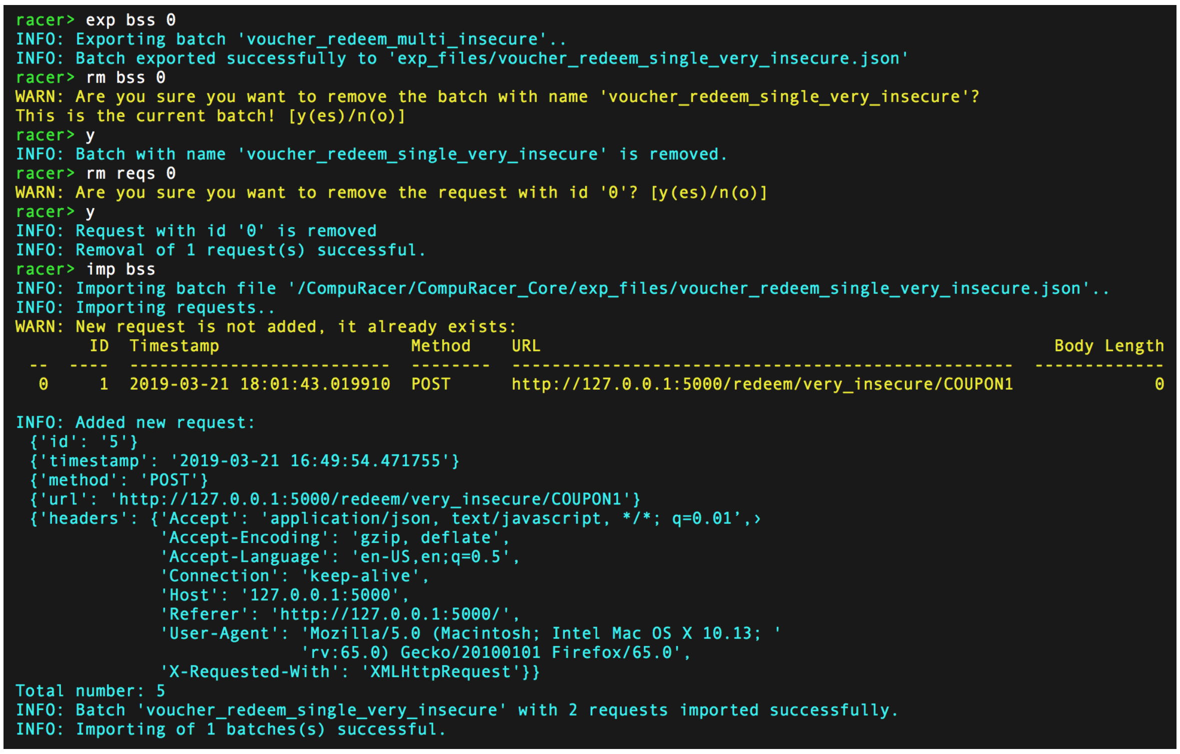
Importing a batch requires a slightly different approach. As exported batches can be stored all over the system, it does not require any arguments, but it opens a file-picker dialog to the default storage location. The tester can select one or more valid exported-batch files and import them back. The tester can import a batch using the following command:

```
racer> imp batches (bss)
```

Example As we have not exported a batch before, we should first do this. We export the only batch that is currently in storage `voucher_redeem_single_very_insecure` with id ’0’. Then, we remove the batch and one of its two requests from the Core itself and then import the exported batch again. It is shown that it only imports one of the requests again. This request gets the id ’5’ as the highest request id used to be ’4’. The process can be seen in figure C.7. The import file-picker dialog is not shown.

C.3 How to send a batch and interpret the results?

In this section, we will show how to send the batches of requests and how to view and evaluate the results. Sending a batch is rather straightforward. There are two ways to do it. We can send a self-created batch, or we can allow the immediate mode to send the automatically created batch. The effects on the target web app should be the same. By using the following command, we can send a batch:



```
racer> exp bss 0
INFO: Exporting batch 'voucher_redeem_multi_insecure'..
INFO: Batch exported successfully to 'exp_files/voucher_redeem_single_very_insecure.json'
racer> rm bss 0
WARN: Are you sure you want to remove the batch with name 'voucher_redeem_single_very_insecure'?
This is the current batch! [y(es)/n(o)]
racer> y
INFO: Batch with name 'voucher_redeem_single_very_insecure' is removed.
racer> rm reqs 0
WARN: Are you sure you want to remove the request with id '0'? [y(es)/n(o)]
racer> y
INFO: Request with id '0' is removed
INFO: Removal of 1 request(s) successful.
racer> imp bss
INFO: Importing batch file '/CompuRacer/CompuRacer_Core/exp_files/voucher_redeem_single_very_insecure.json'..
INFO: Importing requests..
WARN: New request is not added, it already exists:
       ID  Timestamp                   Method    URL                                                  Body Length
  --  ----  --------------------------  --------  ---------------------------------------------------  -------------
   0     1  2019-03-21 18:01:43.019910  POST      http://127.0.0.1:5000/redeem/very_insecure/COUPON1               0

INFO: Added new request:
 {'id': '5'}
 {'timestamp': '2019-03-21 16:49:54.471755'}
 {'method': 'POST'}
 {'url': 'http://127.0.0.1:5000/redeem/very_insecure/COUPON1'}
 {'headers': {'Accept': 'application/json, text/javascript, */*; q=0.01',›
              'Accept-Encoding': 'gzip, deflate',
              'Accept-Language': 'en-US,en;q=0.5',
              'Connection': 'keep-alive',
              'Host': '127.0.0.1:5000',
              'Referer': 'http://127.0.0.1:5000/',
              'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; '
                            'rv:65.0) Gecko/20100101 Firefox/65.0',
              'X-Requested-With': 'XMLHttpRequest'}}
Total number: 5
INFO: Batch 'voucher_redeem_single_very_insecure' with 2 requests imported successfully.
INFO: Importing of 1 batches(s) successful.
```

Figure C.7: *The figure shows exporting the 'voucher_redeem_single_very_insecure' batch, removing the batch and one of its requests and importing it again.*

```
racer> go [batch id]
```

After sending the batch, we get several responses back. The CompuRacer Core does not just print all these results as it will often be very much and not very informative. As a tester, we want to be able to quickly spot and investigate anomalies in the responses that indicate a race condition was triggered in the target web app. In order to make this possible, the results of different request are separated. Next to this, for every request, the results that are shown can be divided into three categories: overview tables, a summary of the grouped responses, and the groups themselves. We can view the results of the current batch using the following command:

```
racer> res [show overview tables] [show grouped responses]
```

C.3.1 Overview tables

Every overview table checks just one characteristic of the responses and counts every different value. Currently, the following four characteristics are checked: the status codes, body lengths (bytes), numbers of header and the header lengths (bytes). If the table has just one row, all responses are the same regarding the metric. Otherwise, it might be interesting to investigate it further.

For instance, if the status code is 403 (forbidden) for 18 packets and 302 (redirect) for two packets, then we know that something was allowed for two times. If that something is a function that should only be used once, we might have found a race condition.

C.3.2 Grouped responses

All responses are grouped based on the headers and the body. Some time and tag headers that are likely to change between responses, but are not informative, are ignored. First, it is shown how much groups we have. If this is more than one group (everything is the same) and less than the total number of parallel requests (everything is different), some responses are equal and might be interesting. Next to this, the fields that always match between responses, the fields that never match and the ignored fields are listed.

The always- and never-matched fields are especially interesting when we want to be more/less strict about some differences to create more/fewer groups. The grouping behaviour can be changed for the current batch by ignoring more or fewer fields in the grouping process. This can be altered at runtime. For instance, when we get too many groups, we might want to add a never-matched field to the ignore list. This can be done using the first command below. Viewing the ignored fields for the current batch can be done using the second command. Finally, resetting the ignored fields to the default is one using the third command:

```
racer> add ignore (ign) [field name (case sensitive)]
racer> get ignore (ign)
racer> res ignore (ign)
```

For every change to the ignored fields, the groups will be re-created automatically. When something seems off regarding the grouping after a crash, or randomly, the tester can run the following command to re-create the groups for all batches. For many batches, this could take some time:

```
racer> regroup batches (regr bss)
```

Example of sending a batch

As an example, we will send the batch created earlier called `voucher_redeem_single_very_insecure` to the server of the test app. The batch is supposed to trigger the following TOCTOU race condition and redeem more vouchers than available.

The race would work as follows. When a user tries to redeem a voucher using the 'very insecure' method, it uses two transactions instead of one and also sleeps for 3 seconds between the two transactions. One transaction to check whether the voucher can still be used and one transaction to reduce the available amount by one. In this case, we use a 'single' voucher that can be used only once. When the two transactions are executed in parallel with two other transactions, they might both read a voucher-availability of 1 and reduce this to 0, while this should not be possible. The test app will send the left-over amount back to the client, so a race condition should be easy to spot. If we get more than one success response with an amount of 1, a race condition has occurred.

It is probable that all five parallel requests of the batch will succeed to redeem the single-use voucher as we have a huge race window of 3 seconds. Therefore, for this example, we first change the parallel duplication of the request in the batch to 10 requests using the command below:

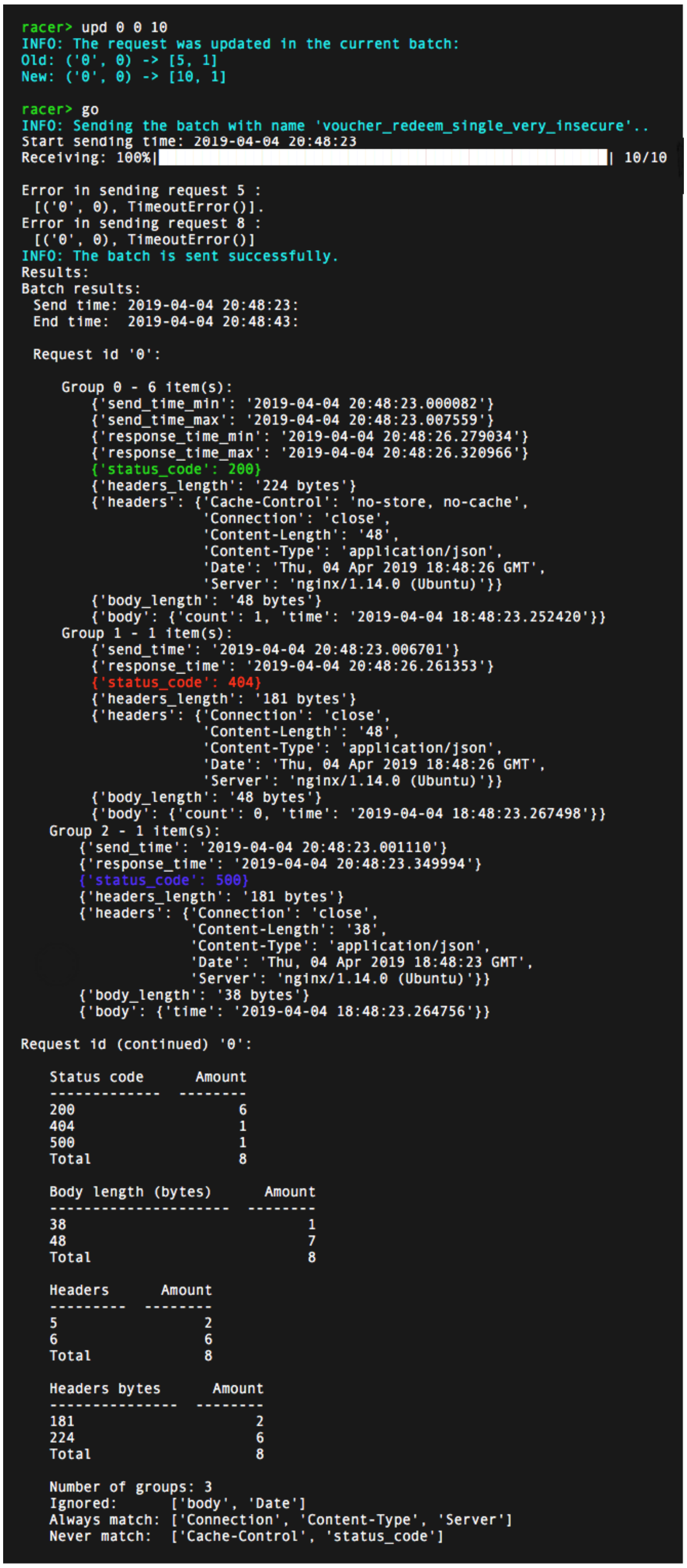
```
racer> update (upd) [request id] [delay] [num parallel]
   ↪ [num sequential]
```

Now, we send the batch. The required commands and the results of sending the batch are shown in figure C.8. If we would want to view these results again, we use the command as shown before where both boolean arguments are set to 'true':

```
racer> res t t
```

In figure C.8, it shows that the tool has sent the ten requests and 20 seconds later (indicated by the send and end time), the results are back. These 20 seconds are not coincidental. This is the timeout that is used by default. Two responses were not back in time to meet this requirement and have been ignored in the results.

The results show three groups. One with six success responses (200), one not-found response (404) and two internal server errors (500). From this, we can conclude that we were successful in triggering the voucher-redeem race: the single-use voucher was redeemed six times.



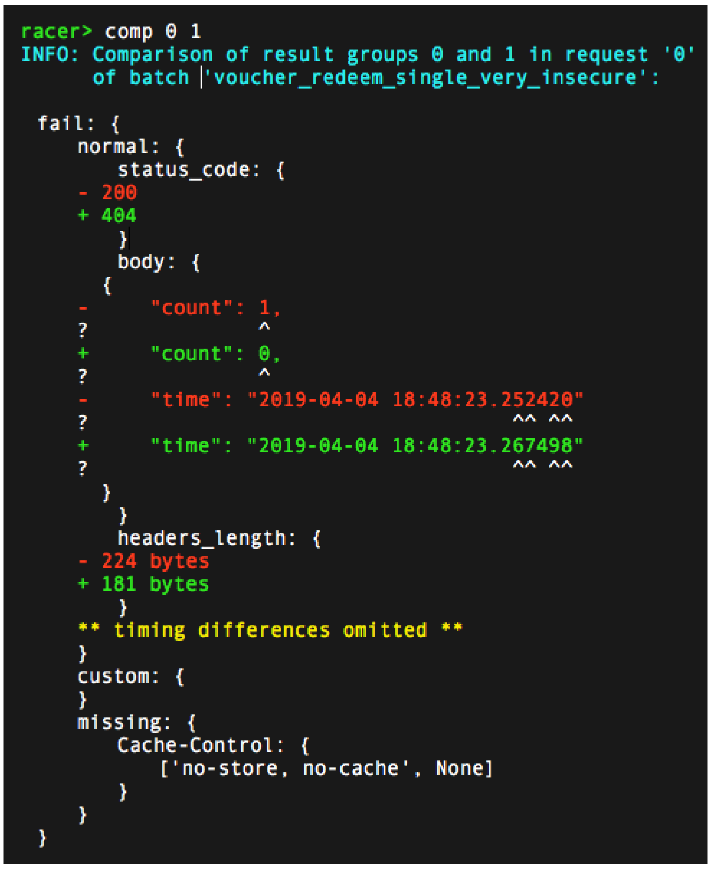
```
racer> upd 0 0 10
INFO: The request was updated in the current batch:
Old: ('0', 0) -> [5, 1]
New: ('0', 0) -> [10, 1]

racer> go
INFO: Sending the batch with name 'voucher_redeem_single_very_insecure'..
Start sending time: 2019-04-04 20:48:23
Receiving: 100%|██████████████████████████████████████████| 10/10

Error in sending request 5 :
 [('0', 0), TimeoutError()].
Error in sending request 8 :
 [('0', 0), TimeoutError()]
INFO: The batch is sent successfully.
Results:
Batch results:
 Send time: 2019-04-04 20:48:23:
 End time:  2019-04-04 20:48:43:

 Request id '0':

    Group 0 - 6 item(s):
       {'send_time_min': '2019-04-04 20:48:23.000082'}
       {'send_time_max': '2019-04-04 20:48:23.007559'}
       {'response_time_min': '2019-04-04 20:48:26.279034'}
       {'response_time_max': '2019-04-04 20:48:26.320966'}
       {'status_code': 200}
       {'headers_length': '224 bytes'}
       {'headers': {'Cache-Control': 'no-store, no-cache',
                    'Connection': 'close',
                    'Content-Length': '48',
                    'Content-Type': 'application/json',
                    'Date': 'Thu, 04 Apr 2019 18:48:26 GMT',
                    'Server': 'nginx/1.14.0 (Ubuntu)'}}
       {'body_length': '48 bytes'}
       {'body': {'count': 1, 'time': '2019-04-04 18:48:23.252420'}}
    Group 1 - 1 item(s):
       {'send_time': '2019-04-04 20:48:23.006701'}
       {'response_time': '2019-04-04 20:48:26.261353'}
       {'status_code': 404}
       {'headers_length': '181 bytes'}
       {'headers': {'Connection': 'close',
                    'Content-Length': '48',
                    'Content-Type': 'application/json',
                    'Date': 'Thu, 04 Apr 2019 18:48:26 GMT',
                    'Server': 'nginx/1.14.0 (Ubuntu)'}}
       {'body_length': '48 bytes'}
       {'body': {'count': 0, 'time': '2019-04-04 18:48:23.267498'}}
   Group 2 - 1 item(s):
      {'send_time': '2019-04-04 20:48:23.001110'}
      {'response_time': '2019-04-04 20:48:23.349994'}
      {'status_code': 500}
      {'headers_length': '181 bytes'}
      {'headers': {'Connection': 'close',
                   'Content-Length': '38',
                   'Content-Type': 'application/json',
                   'Date': 'Thu, 04 Apr 2019 18:48:23 GMT',
                   'Server': 'nginx/1.14.0 (Ubuntu)'}}
      {'body_length': '38 bytes'}
      {'body': {'time': '2019-04-04 18:48:23.264756'}}

Request id (continued) '0':

   Status code      Amount
   -------------  --------
   200                   6
   404                   1
   500                   1
   Total                 8

   Body length (bytes)      Amount
   ---------------------  --------
   38                            1
   48                            7
   Total                         8

   Headers      Amount
   ---------  --------
   5                 2
   6                 6
   Total             8

   Headers bytes      Amount
   ---------------  --------
   181                     2
   224                     6
   Total                   8

   Number of groups: 3
   Ignored:      ['body', 'Date']
   Always match: ['Connection', 'Content-Type', 'Server']
   Never match:  ['Cache-Control', 'status_code']
```

Figure C.8: *The figure shows sending of the 'voucher_redeem_single_very_insecure' batch, and getting the results.*

If there were many equivalent result-groups, it would have been advantageous to automatically compare the contents just like we compared two requests figure C.4. Comparing two result groups can be done using the following command:

```
racer> comp [result id 1] [result id 2]
```

Figure C.9 shows the results of comparing the first group '0' of success responses with the second group '1' of a not-found response. It highlights the response code difference, the difference in body content, send and receive times (omitted) and finally indicates that that the `Cache-Control` header is missing from the second result group.



```
racer> comp 0 1
INFO: Comparison of result groups 0 and 1 in request '0'
      of batch 'voucher_redeem_single_very_insecure':

 fail: {
    normal: {
        status_code: {
    - 200
    + 404
        }
        body: {
      {
    -       "count": 1,
    ?                ^
    +       "count": 0,
    ?                ^
    -       "time": "2019-04-04 18:48:23.252420"
    ?                                     ^^ ^^
    +       "time": "2019-04-04 18:48:23.267498"
    ?                                     ^^ ^^
      }
        }
        headers_length: {
    - 224 bytes
    + 181 bytes
        }
    ** timing differences omitted **
    }
    custom: {
    }
    missing: {
        Cache-Control: {
            ['no-store, no-cache', None]
        }
    }
 }
```

Figure C.9: *The figure shows the comparison of the first two result groups after sending the 'voucher_redeem_single_very_insecure' batch.*