Turbo Intruder: Embracing the billion-request attack

Turbo Intruder: Embracing the billion-request attack - James Kettle, PortSwigger Research.

- Published: 2019-01-25
- Original: <https://portswigger.net/research/turbo-intruder-embracing-the-billion-request-attack>
- Preserved from: https://portswigger.net/research/turbo-intruder-embracing-the-billion-request-attack (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Turbo Intruder: Embracing the billion-request attack | PortSwigger Research

Turbo Intruder: Embracing the billion-request attack

[image: James Kettle]

James Kettle

Director of Research

@albinowax

-

**Published: **Friday, 25 January 2019 at 11:17 UTC

-

**Updated: **Tuesday, 16 August 2022 at 09:14 UTC

-

[image: Turbo Intruder]

Automated web application attacks are terminally limited by the number of HTTP requests they can send. It's impossible to know how many hacks have gone off the rails because you didn't quite manage to bruteforce a password, missed a race condition, or failed to find a crucial folder.

In this presentation I introduce, demo and distribute Turbo Intruder - a research grade open source Burp Suite extension built from scratch with speed in mind. I also discuss the underlying HTTP abuse that enables it to go so fast, so you can attain similar speeds in any tools you happen to write.

This is the livestream recording from Bugcrowd's LevelUp #03 online conference:

Video tags are not supported by your browser.

If videos aren't your thing, here's a brief overview of when and how to use it.

Turbo Intruder is a Burp Suite extension for sending large numbers of HTTP requests and analyzing the results. It's intended to complement Burp Intruder by handling attacks that require exceptional speed, duration, or complexity. The following features set it apart:

-

Fast - Turbo Intruder uses a HTTP stack hand-coded from scratch with speed in mind. As a result, on many targets it can seriously outpace even fashionable asynchronous Go scripts.

-

Scalable - Turbo Intruder can achieve flat memory usage, enabling reliable multi-day attacks. It can also be run in headless environments via the command line.

-

Flexible - Attacks are configured using Python. This enables handling of complex requirements such as signed requests and multi-step attack sequences. Also, the custom HTTP stack means it can handle malformed requests that break other libraries.

-

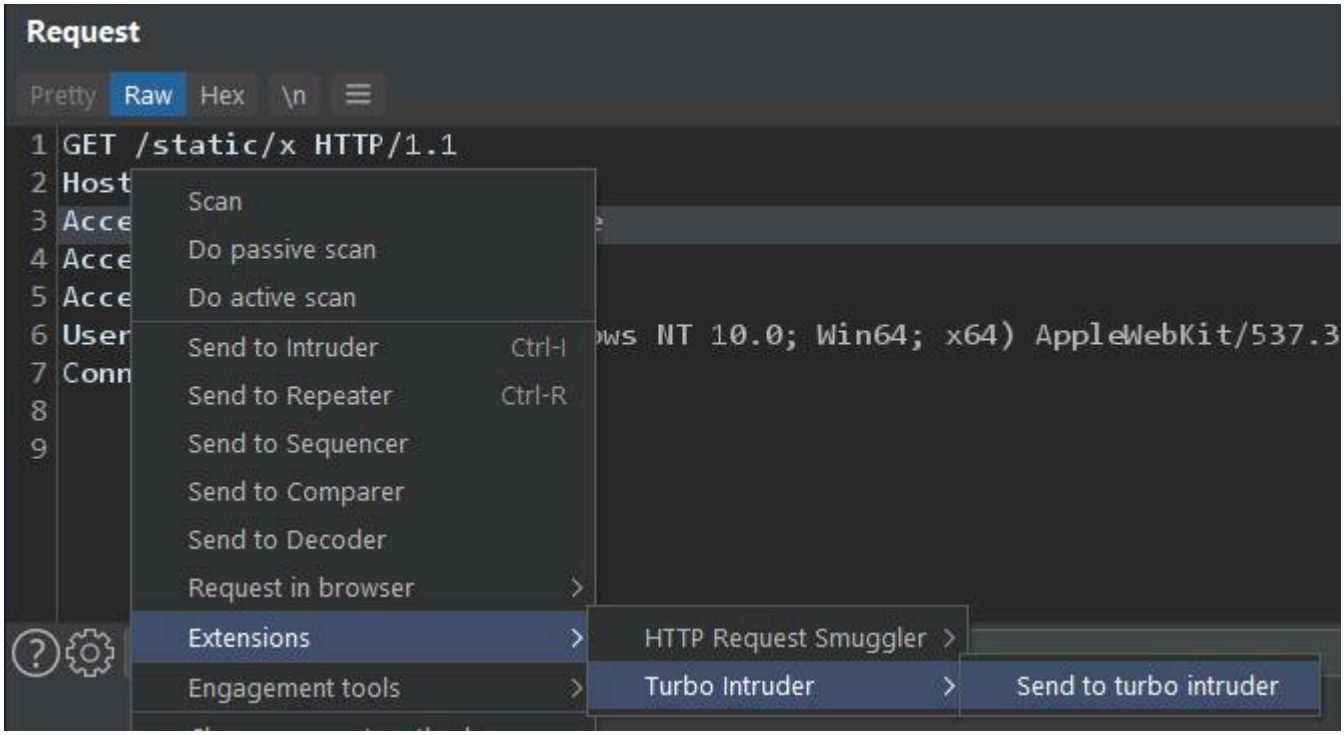
Convenient - Boring results can be automatically filtered out by an advanced diffing algorithm adapted from Backslash Powered Scanner. This means you can launch an attack and obtain useful results in two clicks.

On the other hand it's undeniably harder to use, and the network stack isn't as reliable and battle-tested as core Burp's. Finally, I should mention that it's designed for sending lots of requests to a single host. If you want to send a single request to a lot of hosts, I recommend ZGrab.

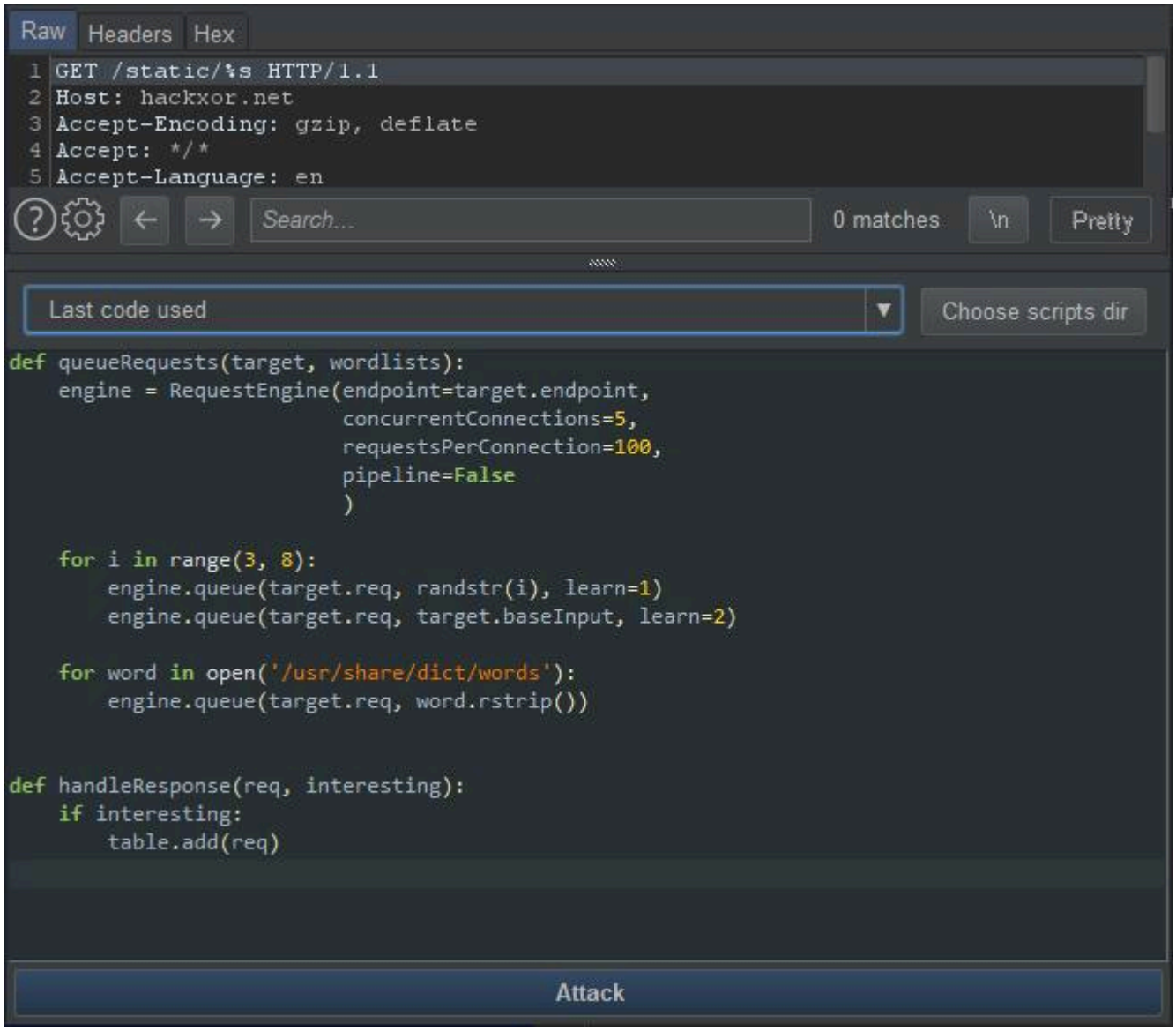
Getting started

Install Turbo Intruder into Burp Suite using the BApp Store under the Extender tab. If you prefer to build it from source, use gradle build fatJar then load src/build/libs/turbo-intruder-all.jar via Extender->Extensions->Add.

To use it, simply highlight the area you want to inject over, then right click and 'Send to Turbo Intruder':



This will open a window containing your request and a Python snippet looking something like this:



You can customise this code depending on what you want to achieve with your attack.

Basic use

You'll notice that the area of the request that you highlighted has been replaced with '%s' - this is where your payloads will be placed. To begin with you'll probably just want to change '/usr/share/dict/words' to a more suitable wordlist. Other than that the default script should work out of the box for simple use cases. You can start/stop an attack using ctrl+enter.

A word of caution

Turbo Intruder achieves speed through network-level efficiency so it should remain relatively performant even on poor network connections; my first live demo was done in a pub basement using wifi borrowed from the coffee shop next door, and still achieved a few hundred requests per second.

This means that the target website will probably be the limiting factor for how fast it can run. As it uses low numbers of concurrent connections it should be unlikely to cause the classic DoS situation where a server's connection pool is consumed and nobody else can connect. However

targeting a resource intensive page may make the server struggle and slow down for everyone, so monitoring application performance during attacks is advised.

Filtering boring results

In Turbo Intruder, responses don't get placed in the results table automatically. Instead, the 'handleResponse' function is called, in which you can decide whether to add the response to the table. Here's a simple example:

```
def handleResponse(req, interesting): if '200 OK' in req.response: table.add(req)
```

Coding a custom response check here isn't always necessary - if you queue a few requests with the 'learn' parameter then Turbo Intruder will learn those responses as boring, and then set the 'interesting' argument based on whether each new response looks like a boring one. This strategy is used by the default script. For further information on this filtering strategy, check out my presentation on Backslash Powered Scanner.

If you'd like to define your own filters based on response attributes, you can achieve this easily using decorators:

```
@MatchStatus(200,204) @MatchSizeRange(100,1000) def handleResponse(req, interesting): table.add(req)
```

For further information on this approach, check out the full decorator documentation.

Speed tuning

If speed is important for your attack, there's quite a few steps you can take to make it go faster.

First, reduce the outbound traffic requirement per request by deleting unnecessary headers and cookies from your request to make it as small as possible.

Then, explore whether you can trim the response size by using the HEAD method, the Range header (as spotted by Agarri), and the Accept-Encoding header.

Next, pick the fastest request engine. Typically Engine.HTTP2 is the fastest if it works, followed by a well-tuned Engine.THREADED, followed by Engine.BURP2 then Engine.BURP.

Next, tune the engine settings. If you ended up with either of the Burp engines, this just means tuning 'concurrentConnections' which is just the number of threads Turbo Intruder uses to send requests. If you're using the first two engines, you'll want to tune the pipeline, requestsPerConnection, and concurrentConnections arguments, probably in that order. Your goal should be to find values that maximise the RPS (Requests Per Second) value, while keep the Retries counter close to 0. The optimum values here are highly server dependent, but the max speed I've achieved to a remote server so far is 30,000 RPS.

If you've achieved a good speed by this point, it's time to optimise your callback code. In particular, table.add() places a heavy load on the Swing thread so you can reduce the CPU burden on your system by analysing each response to decide whether to place it in the results table. Also, avoid regex and looped-string concatenation.

Finally, if you need yet more speed, take advantage of Turbo Intruder's command-line operation by renting a server as close to the target as possible. You know you want to.

Please note that depending on what endpoint you target, the application request processing may be the bottleneck. Your attack is never going to go fast if a weedy server has to execute 15 SQL queries to process each request. On the bright side, it'll probably be easy to trigger race conditions...

Long-running attacks

Turbo Intruder can comfortably perform attacks requiring millions of requests, provided you follow two key principles.

First, don't save responses into the results table unless necessary. Every response you place in the table nibbles a bit of your RAM, so it's best to use the decorator system to filter out the junk.

The second point is a general programming principle: conserve memory by streaming data instead of buffering it. For example, don't pre-load wordlists into memory like this:

```
words = open('/usr/share/dict/words').readlines() for word in words: engine.queue(target.req, word.rstrip())
```

instead, process them line by line:

```
for word in open('/usr/share/dict/words'): engine.queue(target.req, word.rstrip())
```

Finding race conditions

The default script uses a streaming attack style, which is great for minimising memory usage but not ideal for finding race conditions. To find a race condition you'll want to ensure all your requests hit the target in as small a window as possible, which can be achieved using the purpose-built 'gate' system demonstrated in race.py.

For a real life example of using Turbo Intruder to find a race condition, check out Password reset code brute-force vulnerability in AWS Cognito.

Built-in wordlists

Turbo Intruder has two built-in wordlists - one for launching long-running bruteforce attacks, and one containing all words observed in in-scope proxy traffic. The latter wordlist can lead to some quite interesting findings that would normally require manual testing to identify. You can see how to use these in specialWordlists.py.

Command line usage

From time to time, you might find you want to run Turbo Intruder from a server. To support headless use it can be launched directly from the jar, without Burp.

You'll probably find it easiest to develop your script inside Burp as usual, then save and launch it on the server:

```
java -jar turbo.jar <scriptFile> <baseRequestFile> <endpoint> <baseInput>
```

Example: `java -jar turbo.jar resources/examples/basic.py resources/examples/request.txt https://example.net:443 foo`

The command line support is pretty basic - if you try to use this exclusively you'll probably have a bad time. Also, it doesn't support automatic interesting response detection as this relies on various Burp methods.

Swapping request engines

Turbo Intruder has been architected to let you easily swap between different network stacks, using the optional 'engine' argument to RequestEngine. By default it'll use the speedy Engine.THREADED options, but in some situations you may prefer to take advantage of Burp's stability or upstream proxy handling by using Engine.BURP instead. **Update 2021/08/24:** I've added a custom HTTP/2 stack available as Engine.HTTP2, and you can also use Burp's HTTP/2 stack via Engine.BURP2.

Debugging problems

If the 'failed' counter in the stats panel starts rapidly increasing, that's a good sign that something is amiss. It could be a problem with your script, the target website, or Turbo Intruder's network stack. If you're interested in helping make the tool better, I've made a debug script to help you identify and potentially report the source of the problem.

Multiple parameters

You can easily use multiple parameters simply by passing them in as a list.

Multi-host attacks

Each request engine can only speak to a single TCP endpoint, but if you want to send requests to multiple websites you can achieve this by creating multiple request engines. Turbo Intruder was designed with single-host attacks in mind, and holding large numbers of TCP connections open at once doesn't scale very well. Also, the stats panel only reflects one request engine. I might address this at some point in the future but for now, if you want to send a small number of requests to a large number of hosts, I'd recommend using ZGrab.

Attack state

As you learn to write more advanced attacks, you might want to track state between requests. The request engine has a dict called userState which you can use for this. I've written an example script that demonstrates using this feature to enumerate usernames via a timing attack.

More examples

To learn about other advanced features available in Turbo Intruder, check out the full list of examp le scripts.

Editor settings

You can customise some editor settings like line-numbers and font size via the settings menu:

[image: image]

Final note

As ever, I've got many plans for future improvements to Turbo Intruder. We also look forward to eventually introducing some features from Turbo Intruder's network stack into core Burp Suite tools.

Good luck, have fun, and try not to take anything down.