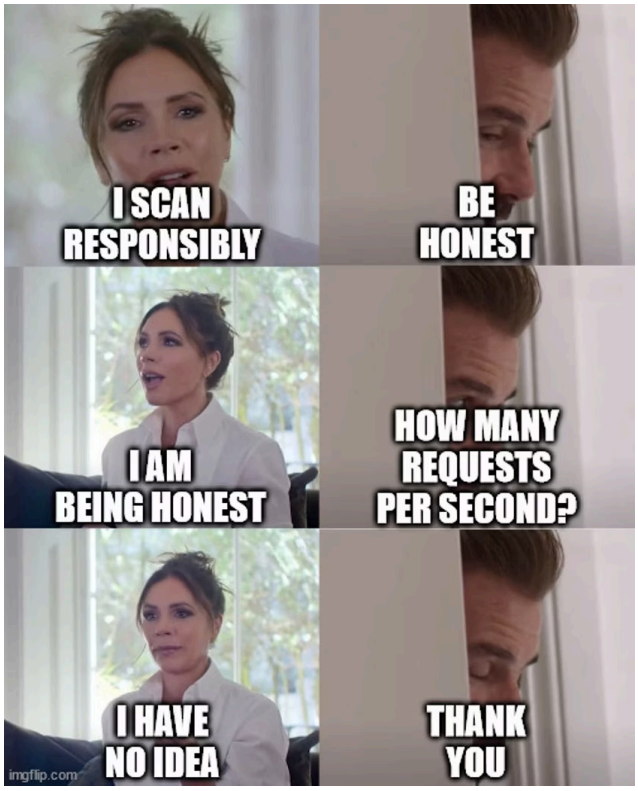
Aggressive scanning in bug bounty (and how to avoid it)

BY CRYPTOCAT · MARCH 18, 2024 · LAST UPDATED ON MARCH 6, 2025

Presented by CryptoCat

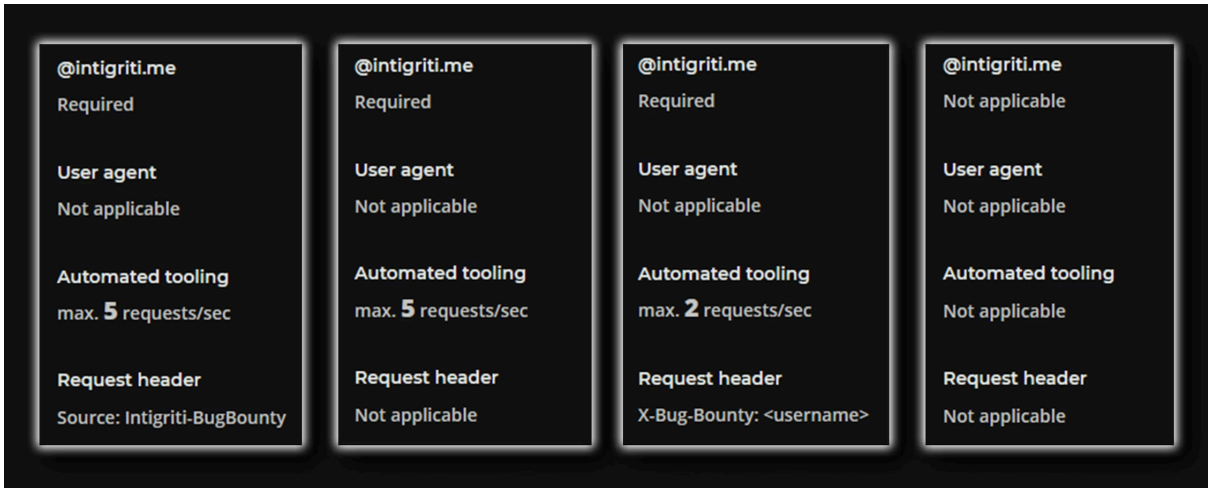
What is aggressive scanning?



In bug bounty, researchers are expected to configure automated tools and scanners to remain within the defined limits of the program's requirements. Any activity outside these limits can be defined as "aggressive" or "intrusive".

Hunters with experience in penetration testing will be familiar with this concept. Clients and testers agree on the rules of engagement and define whether intrusive testing is permitted and to what extent. Intrusive tests could produce better results in the same timeframe but can have negative consequences that we'll outline soon.

Here are a few examples of the requirements taken from random public programs currently listed on the Intigriti platform.



So, these programs permit automated tools *but* only if they send 2-10 requests per second (RPS), depending on the program. Maybe we'll decide to launch a **gobuster** scan to fuzz the web directories of one of the targets. If we did, we'd immediately see the scan is considerably faster than 5 RPS, but there's no built-in way to measure it. We know we've just broken the terms of the program (Ooops), but we don't know by how much.

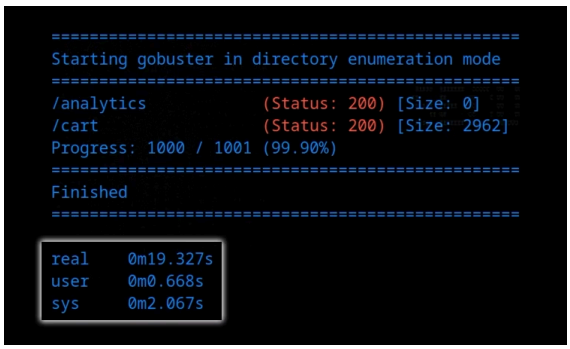
Some tools will display these statistics by default. Others reveal them when you add a verbose or debug flag. If not, there are a few techniques you could use that vary in accuracy due to environmental factors:

- Test the tool against your own web server and see how many RPS you receive.
- Proxy the tool (e.g., through **burp**) and see how many RPS are sent.
- Check the network traffic (e.g., **wireshark**).
- Use a tool like **time** and then calculate the RPS manually.

I went with the fourth option, **time**. It's not the most accurate since it measures the overall program execution time, but it's good enough in this case. I was also careful to test against a Portswigger lab rather than a real target and used a small wordlist (1000).

```
head -n 1000 /usr/share/wordlists/dirb/common.txt > words.txt

time gobuster dir -u https://LAB-ID.web-security-academy.net/ -w words.txt
```



The final execution time was 20 seconds, indicating **gobuster** sent ~50 RPS, significantly more than any of our chosen programs permit.

Let's repeat our default config experiment with another common tool, **ffuf**. To switch things up, we'll brute-force login credentials this time, but the principle is the same—it's an automated tool, and we need to deploy it responsibly.

```
ffuf -c -ic -request new.req -request-proto https -w /usr/share/seclists/Passwords/probable-v2-top1575.txt -fr "Invalid"
```

```
:: Progress: [1575/1575] :: Job [1/1] :: 60 req/sec :: Duration: [0:00:23] :: Errors: 0 ::
```

Thankfully, **ffuf** does the calculations for us and displays the RPS during execution. The credential attack finishes in 23 seconds with an average of 60 RPS – an even more severe breach of the rules of engagement. Later, we'll see how to configure these tools correctly.

Why should you avoid it?

There are valid reasons why companies might want to control the levels of traffic received by researchers. Firstly, *performance*. It should go without saying that hammering targets with HTTP requests can lead to service degradation. In the worst-case scenario, the impact can be so severe that it leads to downtime (DoS).

Secondly, *noise*. Most organizations have teams dedicated to threat detection and incident response. Their job is to monitor for alerts of potential attacks and respond accordingly. When they see a flood of traffic from an automated tool, e.g., **sqlmap**, they need to investigate and determine whether it's a real attack from a malicious actor, *or* a researcher who's failed to comply with the rules of engagement. Not only does this waste valuable resources, but it can also lead to actual attacks slipping through the net.

Remember: behind every bug bounty program there's a program manager who needs to justify the benefits of bug bounty to the organization. While they might personally appreciate how effective it is for increasing the overall security posture, other teams often only see the negative impact and disruption to the business is the most critical (and visible) drawback.

We've looked at why companies *want* you to avoid aggressive scanning, but why *should* you? Let's return to the pentesting example for a moment. When a client contracts a penetration test, all parties involved

agree on the rules of engagement. They might limit which assets can be tested as some are too important to risk any downtime. They could also set limits on the times on which certain assets can be tested, e.g., outside of business hours.

Additionally, they could limit the tools that can be deployed due to performance issues or the risk posed to systems. If automated tools are permitted, it might be necessary to throttle them to stay within an agreed RPS, either during business hours or permanently. These are a small subset of the restrictions that can be incorporated into a pentest contract but are especially relevant to our aggressive scanning example.

So, what happens if a pentester subsequently breaches these rules of engagement? Of course, mistakes happen; the client may provide a verbal or written warning in the first instance, providing no serious harm was caused. Let's say the pentester proceeds to perform intrusive tests, leading to service degradation or downtime. Not only is this unprofessional behavior unlikely to result in future work or recommendations, but it could also lead to more severe consequences, e.g., contract cancellation or financial penalties.

Bug bounty is no different! When you start hacking on a program, you implicitly agree to the terms. Suppose you break the rules of engagement by exceeding the defined max RPS for automated tools. In that case, you will receive a warning from our community support team, politely requesting that you double-check your configuration. Repeated violations will be followed by further sanctions, potentially leading to invalidated reports, removal from the program, or even suspension from the platform.

How can you avoid it?

Understand the requirements. Before you start hacking on a new program, ensure you thoroughly read the scope and rules of engagement. We also recommend familiarising yourself with the community code of conduct and researcher terms and conditions.

Configure your tools accordingly. Most tools maximize performance by sending as many RPS as possible. You are responsible for reading the documentation, configuring, and testing the tool before applying it against the bug bounty target.

Configuring Common Tools

The list of automated tools used by ethical hackers is endless, and each one will have a different syntax for limiting the RPS. We recommend reviewing the documentation for options referencing keywords like "rate limit" and "throttle". A fast, generic way to do this is using **grep**, e.g.,

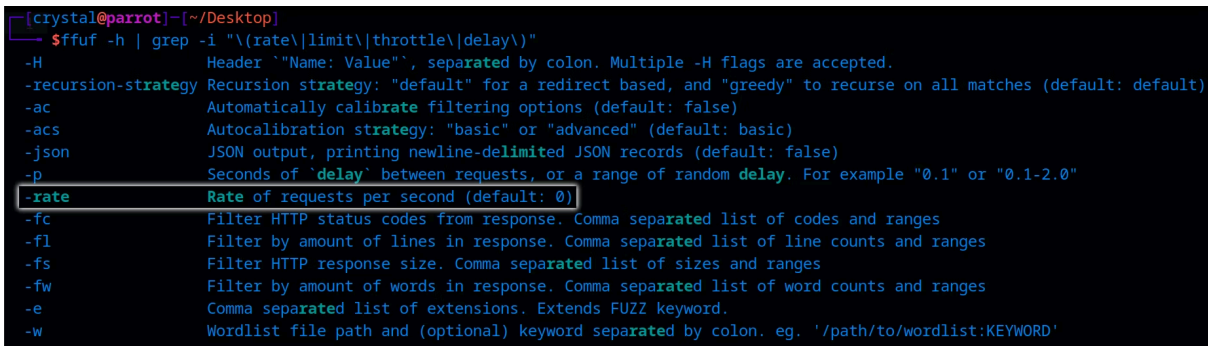
```
tool_name -h | grep -i "\(rate\|limit\|throttle\|delay\)"
```

If the command doesn't yield any valid results, add some additional keywords, e.g. "time", "sec" or "request". Alternatively, you might need to review the full help menu output, as many tools offer an "advanced" help menu for finetuning the configuration.

Let's practice with some common automated tools!

ffuf

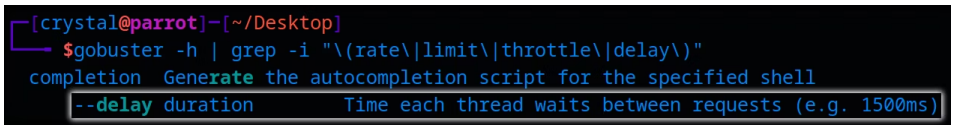
```
ffuf -h | grep -i "\(rate\|limit\|throttle\|delay\)"
```



What can we take away from the output? The default rate is `0` (unlimited), and we can adjust the "rate of requests per second" with the `-rate` option. Therefore, a value of `5` would limit the tool to 5 RPS.

gobuster

```
gobuster -h | grep -i "\(rate\|limit\|throttle\|delay\)"
```



Notice the difference in both syntax and behavior compared to `ffuf`. Specifically, the flag is `--delay` rather than `-rate`, and instead of providing a value representing the number of RPS, users must indicate the time to wait between each request. Therefore, users could input a value of `200ms` or `0.2s` to achieve a limit of 5 RPS... Or could they?

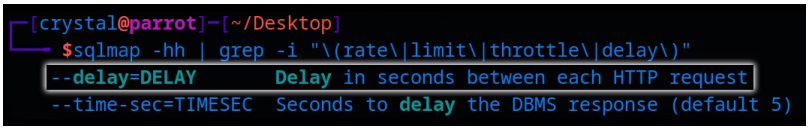
Well, actually, no. If we used one of the earlier techniques to measure the RPS, we'd discover it still vastly exceeds our desired limit. Hopefully, this emphasizes the importance of *verifying* that the tool is performing as you expect before setting it loose on a real target.

If we explore the help menu again, we'll find a `threads` flag with a default value of `10`. In other words, ten concurrent threads perform tasks each time we run `gobuster`. We configured a delay of `0.2s` between each request, *but* that's on a per-thread basis, not overall. To correct the configuration, we can either increase the delay, decrease the threads, or a combination of the two. However, reducing the threads is recommended to ensure consistent accuracy.

sqlmap

```
sqlmap -h | grep -i "\(rate\|limit\|throttle\|delay\)"
```

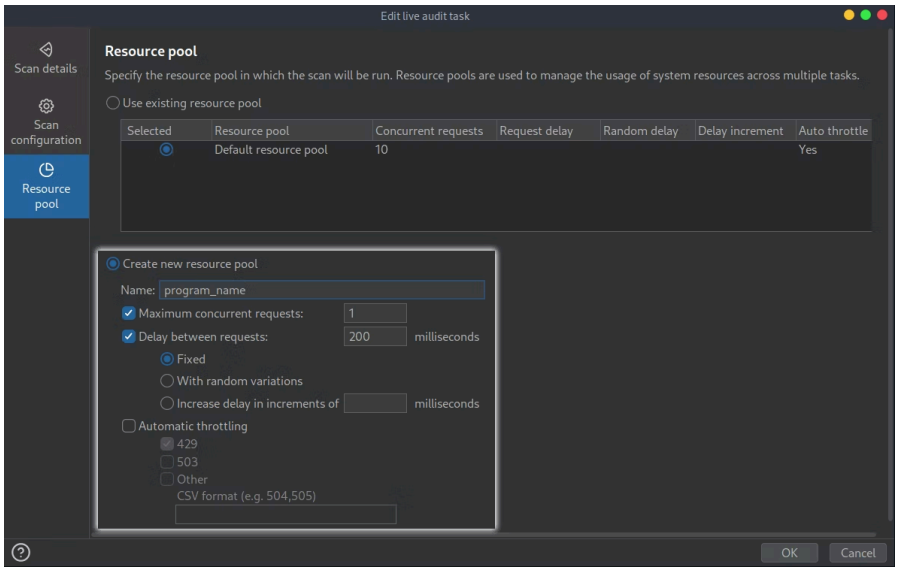
This time, our `grep` command yields no applicable results. Reviewing the help menu will confirm that `sqlmap` offers an advanced help menu that is accessible with the `-hh` flag.



We can specify a delay in seconds with the `--delay` flag (similar to `gobuster`). To limit the tool to 5 RPS, we could provide a value of `0.2`, *providing* we don't have the same issue regarding threads. Thankfully, the help menu confirms the default threads is already at `1`.

burp

How could we forget Burp Suite? The notoriously popular web proxy tool allows users to [configure resource pools per task](). One benefit to this approach is the ability to throttle different tasks according to the program requirements or bug hunter's priorities. We recommend including the program name in the resource pool title to easily keep track.



As with other tools we've reviewed, `burp` can configure the delay between requests *and* the number of concurrent requests (threads). It's important to calculate these values precisely and verify the result.

Conclusion

Hopefully, this article has conveyed the importance of responsible testing in bug bounty, and you can appreciate the negative implications aggressive scanning has on organizations, as well as the consequences for researchers breaching the rules of engagement.

Remember that scanning restrictions are *per program*, not per tool. Five tools running 5 RPS is 25 RPS. That said, the limit is *per program*, so nothing is stopping you from running automated tools on targets from multiple programs at once, so long as you configure each tool correctly and don't forget about threads.

Finally, we've referenced the community code of conduct several times, so let's end with a fitting quote.

> **"Be gentle when conducting automated tests or scanners. Some programs may disallow automated testing of any kind or impose rate limits. It is of utmost importance to follow these rules, as a violation may cause service degradation. Do not conduct any disproportionate testing that may affect service performance. In the event that you notice slower response times or unexpected behaviour that may be related to your testing, make sure to tell us about it."**