Web fuzzing for hackers

BY AYOUB AND ORWA ATYAT · AUGUST 20, 2026

Want to read the latest version? Visit the link below:
https://www.intigriti.com/researchers/blog/hacking-tools/web-fuzzing-for-hackers

Fuzzing has been around for as long as web applications have. In fact, the term itself was coined back in 1988, when Barton Miller, a professor at the University of Wisconsin, was working over a dial-up connection during a thunderstorm and noticed that the resulting line noise was consistently crashing the UNIX utilities he was running. Web fuzzing is no different. Despite the rise of automated scanners and other security tooling, it remains one of the most productive techniques in application security, including bug bounty.

In this article, we'll explore web fuzzing from the ground up: what it actually is, the tooling and wordlists that make it work, and how to fuzz to discover more content and find more security vulnerabilities.

Let's dive in!

Special thanks to GodfatherOrwa!
This article was co-authored by Orwa Atyat (@GodfatherOrwa), a security researcher and bug bounty hunter well-known for uncovering critical bugs through creative fuzzing techniques. If you wish to dive deeper into his work, be sure to check out his personal security blog!

https://orwaatyat.medium.com/

What is web fuzzing

Web fuzzing is the practice of sending large volumes of crafted input at a web target, such as URLs, query parameters, request headers, body parameters, and cookies, to discover new behavior about that target that isn't documented, isn't intended to be exposed, or isn't correctly handled. The idea is simple: if we don't know what's there, we ask, over and over, in as many ways as we can think of, and let the responses tell us.

At its core, fuzzing is a technique for discovery. For instance, we can uncover:

- Unreferenced API endpoints, application routes and files
- Unreferenced query or body parameters
- Unexpected behavior such as app crashes (app-level DoS)
- And any other input handling that misbehaves under conditions the developers didn't anticipate.

Some of those discoveries are directly exploitable, while others result in key pieces of information that lead us to exploitable behavior later on.

Importance of web fuzzing

As listed above, fuzzing can help us discover directly exploitable vulnerabilities and other types of quirky behavior that can lead to bugs. Therefore, understanding how to fuzz and when to is key. Failing to understand renders this technique ineffective and might actually produce more noise rather than any meaningful results.

Let's list a few common practical mistakes to understand the concept further.

1. Fuzzing without a clear objective

Before launching any fuzzing tool, we need to get down to the basics: what are we actually looking for? Fuzzing for files and endpoints is fundamentally different from fuzzing for query parameters, which, in turn, is different from fuzzing request headers or cookies. Each task requires a different wordlist, and possibly a different tool or configuration.

On top of that, certain request elements, such as the request method, our IP address, User-Agent, or other request headers like `Host`, `X-Requested-With`, and `Content-Type`, can significantly influence the HTTP response we receive. Failing to account for these behaviors can again lead to more noise rather than the results we are looking for.

2. Using the wrong wordlist

Generic wordlists with commonly occurring keywords are often used for fuzzing, however, it is important to remember that such wordlists are limited and may not generate meaningful results. For instance, some technologies use a specific naming convention that may not be part of a generic wordlist. We can use this to our advantage to craft more relevant wordlists that can help us produce more accurate results.

We'll cover wordlists in depth in the next section.

3. Failing to adhere to rate limits

Web fuzzing generates a high volume of requests, and most production targets enforce rate limiting in some form. Sending more requests than the target can handle or than its WAF can tolerate will get us blocked and will render all our results inaccurate. Therefore, throttling our requests isn't just practical for complying with program rules, it’s also necessary to gather any meaningful results.

Wordlists

One of the most overlooked factors in web fuzzing is the wordlist itself. In simple terms, the wordlist defines the scope of what we're looking for, every request our fuzzer sends is derived from it. If a path, filename, or keyword isn't in the wordlist, it won't be found. This makes wordlist selection one of the most impactful decisions in the entire process.

Generic wordlists, such as SecLists' common.txt, are a reasonable starting point. However, as previously implied, they won't account for your target's adopted naming conventions, technology stack, or company-specific terminology. A custom wordlist tailored to the target will almost always outperform a generic one, simply because it tests for things that are actually likely to exist on that specific application.

We've covered the process of building custom wordlists in depth in a previous article, including extracting keywords from in-page content, JavaScript files, and URL paths, fingerprinting the technology stack, and combining everything into a single final wordlist file. If you haven't read it yet, we'd highly recommend doing so before continuing, the techniques covered there directly apply to everything we'll discuss from here on.

Web fuzzing tools

Now that we understand the importance of a good wordlist, let's take a look at the tools we can use to put them to work.

There are plenty of (open-source) fuzzing tools available, and each comes with its own strengths. Rather than listing every option out there, we'll focus on a select few that are widely used in the bug bounty community and explain what each one excels at.

> **Tool configuration**
>
> While some of the tools listed below can indeed be deployed for multiple tasks, the way you configure the tool remains crucial. The configuration for a content discovery scan may vastly differ from that of a fuzzing task, whose main objective is to find new subdomains or a WAF bypass for a SQL injection.

Content discovery

Ffuf (Fuzz Faster U Fool)

Ffuf (Fuzz Faster U Fool) is arguably the most popular web fuzzing tool among bug bounty hunters today. It's fast, flexible, and supports fuzzing virtually any part of an HTTP request, such as the URL, request headers, request body, cookies, and so on.

It's also capable of handling multiple wordlists simultaneously and offers powerful filtering options to filter through noise based on common response elements such as status code, response size, word count, and line count. Beyond content discovery, ffuf is also commonly deployed for virtual host (VHost) enumeration by fuzzing the `Host` header, a technique that can reveal internal applications and staging environments hosted on the same server.

Burp Suite Intruder

Burp Suite's Intruder is worth mentioning for cases where we want to fuzz without switching from our proxy interceptor. While it's less performant than the command-line alternatives, its strength lies in its ability to fuzz directly from intercepted requests, we can highlight any part of a request parameter, header, or body value and fuzz it in place. This makes Intruder the easiest tool for targeted, context-specific fuzzing where we've already identified an interesting request.

Subdomain bruteforcing

OWASP Amass is better known as a passive reconnaissance tool, but its `enum` subcommand also supports active subdomain bruteforcing. By providing it with a wordlist, Amass will systematically resolve subdomains against the target, combining bruteforcing with passive data sources to build a comprehensive map of the target's subdomains.

Parameter discovery

Arjun and Burp Suite's Param Miner are purpose-built for parameter discovery. Arjun works by sending crafted requests with parameter names and observing how the application responds, detecting reflected values, changes in response size, or any other suspicious behavior.

Param Miner, on the other hand, takes a similar approach but runs as a Burp extension, integrating directly into our proxy interceptor. Discovering hidden parameters is often what turns an otherwise uninteresting endpoint into something worth testing further.

It is crucial to note that the tool itself is only half the equation. The correct configuration, i.e., the way you send requests and analyze responses, is what separates a meaningful fuzzing task from thousands of meaningless results. Taking the time to understand our target's behavior and configuring the necessary filters accordingly will save us hours of manual review and ensure that the results are actually worth investigating.

Fuzzing for content discovery

Now that we've covered the tooling and wordlists, let's take a look at how we can put them to use. Content discovery is one of the most common applications of web fuzzing, the goal is to uncover files, directories, API endpoints, and other resources on a target that aren't directly referenced or publicly documented.

However, effective content discovery isn't just about pointing a tool at your target. Understanding what we're looking for and how our target is structured before we begin is what makes all the difference.

Fuzzing for subdomains

Subdomain enumeration is often the first step in expanding our attack surface. While passive sources like certificate transparency logs and DNS datasets provide us a great starting point, they won't reveal subdomains that have never been publicly indexed. Bruteforcing fills that gap by resolving possible hostnames from a wordlist against the target's DNS.

Tools like Amass support this through their `enum` subcommand, combining passive results with active bruteforcing in a single scan. For a more lightweight approach, we can use a tool like dnsx to bruteforce subdomains directly by providing it with a list of hostnames:

```
$ dnsx -d example.com -w /path/to/wordlist.txt -silent
app.example.com
api.example.com
support.example.com
blog.example.com
cloud.example.com
admin.example.com
mail.example.com
```

DNS bruteforcing with DNSX

This will try to resolve each subdomain and return any that respond with a valid DNS response. However, it is important to mention that this method is not always feasible, for instance, when our target resolves every subdomain under a root domain. In practice, this means every entry in our wordlist would return a valid response, regardless of whether the subdomain actually exists or hosts anything significant. When this happens, our results become useless unless we further adapt our fuzzing method.

For instance, some tools like dnsx can help and handle this for us. Using its `-wd` flag enables wildcard filtering, automatically detecting and excluding wildcard responses from the output. For tools that don't offer built-in wildcard handling, we'd need to determine what the HTTP response of a non-existing subdomain looks like and filter it out manually based on response size, word count, or status code.

Permutations

Orwa Atyat (GodfatherOrwa)

When fuzzing for subdomains, don't limit yourself to a single pattern. If you're targeting a specific subdomain like `admin`, run all possible variations:

- `admin-FUZZ.target.com` `admin-stg.target.com`, `admin-dev.target.com`
- `FUZZ-admin.target.com` `vpn-admin.target.com`, `api-admin.target.com`
- `adminFUZZ.target.com` `admintest.target.com`, `adminnew.target.com`
- `FUZZadmin.target.com` `devadmin.target.com`, `oldadmin.target.com`
- `admin.FUZZ.target.com` `admin.dev.target.com`, `admin.staging.target.com`

Each pattern reflects a different naming convention, and you'd be surprised how often one of them turns up a result the others miss.

Also, if a bug was patched on the main subdomain, always fuzz for other instances. Staging and development environments are often left unpatched for weeks after a fix goes to production, making them an easy win.

Virtual host enumeration

Aside from subdomains, some developers use a single host to serve multiple applications at once. In practice, this works by parsing the Host header and routing to the correct web root. As you may know, this could be of interest to us as enumerating virtual hosts can help us further expand our attack surface.

To effectively enumerate virtual hosts, we'll have to fuzz the `Host` header rather than resolving DNS names:

```
$ ffuf -u https://app.example.com -w /path/to/wordlist.txt -H "Host: FUZZ.example.com"

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v2.1.0-dev
________________________________________________

 :: Method           : GET
 :: URL              : https://app.example.com
 :: Wordlist         : FUZZ: /path/to/wordlist.txt
 :: Header           : Host: FUZZ.example.com
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200-299,301,302,307,401,403,405,500
________________________________________________

staging                 [Status: 200, Size: 4821, Words: 312, Lines: 87, Duration: 142ms]
dev                     [Status: 403, Size: 1655, Words: 122, Lines: 11, Duration: 96ms]
internal                [Status: 302, Size: 0, Words: 1, Lines: 1, Duration: 118ms]
:: Progress: [16/16] :: Job [1/1] :: 12 req/sec :: Duration: [0:00:01] :: Errors: 0 ::
```

Virtual host fuzzing with Ffuf

Fuzzing for virtual hosts

Orwa Atyat (GodfatherOrwa)

When fuzzing for virtual hosts using Ffuf, I'd recommend matching all response codes with `-mc all` and filtering out empty responses with `-fs 0`. This ensures we don't miss anything, a `302` redirect, a `403 Forbidden`, or even a `500 Internal Server Error` can all indicate a live virtual host worth investigating. A `403` in particular is not a dead end, protected applications often have weaker security than their public-facing variants.

Threads

One thing to watch out for is the thread count. Setting it too high can cause packet loss on your end, which leads to missed results. Depending on what the bug bounty program rules allow, on a VPS, I often see 350 threads as a reasonable ceiling. On a home connection, start around 50–100 and tune from there. It is important to remember that more threads doesn't mean better results.

Fuzzing for application routes, files & API endpoints

Fuzzing to find unreferenced files, API endpoints & app routes is possibly the most common use case for web fuzzing. A simple content discovery scan can quickly reveal forgotten admin panels, exposed configuration files, debug endpoints, or entire API versions that were never meant to be publicly accessible.

Before we start fuzzing, it's worth taking a moment to understand how the target behaves to your incoming requests. As you may have noticed, despite RFCs, the internet is not so defined. Not every application responds to requests the same way, and failing to account for this can lead to misleading results or no results at all.

For instance, single-page applications (SPAs) are a common example. Many modern frameworks like React, Angular, and Vue are configured to return a `200 OK` response for every route, regardless of whether the application route actually exists. Moreover, since the routing is handled client-side rather than on the server, the content length of each HTTP response may even be the same. Under the default configuration, this could confuse our fuzzing tool, making every entry in our wordlist appear to be a valid file or endpoint. To counter this behavior, we'll have to set strict filters on response size, word count, and other response elements, or otherwise, our output becomes meaningless. It is also crucial to note that a `200 OK` doesn't always mean we've found something, we need to verify that the response content actually differs from the application's default response.

Rate limiting and WAF rules are another factor we need to account for. Content discovery generates a high volume of requests, and most production environments enforce some form of rate limiting or WAF protection. Sending requests too aggressively not only risks getting our IP blocked but also negatively affects our results, as rate-limited responses may look different from normal responses. Throttling our requests with appropriate delay settings and monitoring for sudden shifts in response patterns will keep our results accurate and our access intact.

Additionally, some application routes and API endpoints will only respond when specific conditions are met. An API endpoint might require a specific `Content-Type` request header (such as `Content-Type: application/json`) to respond with anything useful. Other API endpoints may only respond to `POST` or `PUT` requests. At the same time, other applications may have middleware that checks whether an anti-CSRF request header, such as the `X-Requested-With` header, was included in the request before serving a response. If we're only fuzzing with default `GET` requests and no custom headers, we're potentially missing an entire layer of the application.

Let's now get to the practical part of fuzzing.

Extension fuzzing

Beyond fuzzing for directory and file names, appending file extensions to our wordlist entries can uncover backup files, configuration files, and other sensitive data that would otherwise go unnoticed. This is particularly effective because many of these files are not referenced anywhere in the application, they simply exist on the server from a previous deployment, a forgotten backup, or a development artifact.

Fuzzing for file extensions

Orwa Atyat (GodfatherOrwa)

When fuzzing for extensions, I always make sure to include a few high-value ones that many hunters overlook. `.md` files are at the top of my list. With the rise of AI-generated applications, developers and AI tools frequently produce Markdown documentation that ends up deployed to production. Beyond that, `.db` (SQLite databases), `.sql` (database dumps), `.env` (local environment variables), `.xml` (configuration files), and archive formats like `.7z`, `.zip`, and `.tar.gz` are all worth including.

Hidden directory and file patterns

Orwa Atyat (GodfatherOrwa)

Standard content discovery will enumerate common directory and file names, but some paths are only discoverable by prefixing entries with specific characters. Dotfiles and dot-directories like `.git`, `.env`, and `.svn` are a well-known example, but they're not the only pattern worth testing.

There are four prefix patterns I always test on every target, as they consistently catch what normal directory bruteforcing misses:

- `/.FUZZ`: hidden dot-directories and dotfiles (`.git/`, `.env`, `.DS_Store`)
- `/-FUZZ`: flag-style hidden directories (`-backup`, `-old`, `-tmp`)
- `/~FUZZ`: user home and tilde backup directories (`~admin/`, `~root/`)
- `/../FUZZ`: path traversal into parent directories

Each pattern targets a different naming convention, and running all four ensures we're not leaving anything on the table.

Technology-specific fuzzing

As we discussed in the wordlists section, tailoring our approach to the target's technology stack will help us find significantly more content. This applies not only to the wordlist itself but also to the extensions and naming patterns we test for.

IIS fuzzing

Orwa Atyat (GodfatherOrwa)

IIS and ASP.NET targets are a good example of why technology-specific fuzzing matters. These servers expose file types that Apache or Nginx would never serve, `.asmx` (legacy SOAP web services), `.ashx` (generic HTTP handlers, often used for debug functionality), `.aspx` (and don't forget `.aspx.bak`), `.asp` (classic ASP, ancient, but still in production), `.dll` (compiled binaries that can leak symbols), and in rare cases even `.exe` (self-hosted services).

On IIS targets, I also fuzz for domain-suffix variations by appending or prepending the target's domain name to wordlist entries, as IIS applications frequently use these naming patterns for internal routes and virtual directories.

Fuzzing for parameters

Developers often process query and body parameters to make the application interactive. Making parameter discovery our next logical step after enumerating all application routes and API endpoints. Have a look at the following practical example.

search.php

```php
<?php

$search_query = isset($_GET['search_query']) ? htmlspecialchars($_GET['search_query']) : '';
$limit = isset($_GET['limit']) ? (int)$_GET['limit'] : 20;
$sort = isset($_GET['sort']) ? htmlspecialchars($_GET['sort']) : 'relevance';

$affiliate_id = isset($_GET['affiliate_id']) ? $_GET['affiliate_id'] : null;

$discount = 0;

if ($affiliate_id) {
    $check = mysqli_query($conn, "SELECT id FROM affiliates WHERE code = '$affiliate_id'");
    if (mysqli_num_rows($check) > 0) {
        $discount = 20;
    }
}

$query = "SELECT * FROM products WHERE name LIKE '%$search_query%' ORDER BY $sort LIMIT $limit";
$results = mysqli_query($conn, $query);
?>

...
```

Hidden parameter

As you can observe in the code snippet above, the `search_query`, `limit`, and `sort` query parameters are all referenced throughout the page. However, upon a closer look, the underlying PHP code also processes a fourth parameter, `affiliate_id`, which can lead to a common sever-side injection attack, SQLi.

Without access to the source code, we could never have identified this parameter. This is what makes parameter fuzzing important. The discovery of parameters is also quite similar to how we've previously learned to discover unreferenced application routes and API endpoints. Tools such as Arjun and Param Miner are purpose-built for parameter discovery and come equipped with a default wordlist. They work by sending requests with possible parameter names and analyzing the application's response for changes

in HTTP response size, status code, reflections in the response body, or observable delays in response time.

If we were to fuzz the endpoint from our practical example, we'd have been able to spot the `affiliate_id` as it changes the response size when supplied as a query parameter.

```
$ arjin -u https://example.com/search.php?search_query=test --stable

    _
   /_| _ '
  (  |/ /(//) v2.2.7
      _/

[*] Probing the target for stability
[*] Analysing HTTP response for anomalies
[*] Analysing HTTP response for potential parameter names
[*] Logicforcing the URL endpoint
[✓] parameter detected: limit, based on response similarity
[✓] parameter detected: sort, based on response similarity
[✓] parameter detected: affiliate_id, based on response similarity
[+] Parameters found: limit, sort, affiliate_id
```

Hidden parameter bruteforcing with Arjun

While most parameters are never meant to be hidden, you will encounter cases where certain query or body parameters, often part of a legacy codebase, are neither referenced nor present in any client-side resource. At the same time, they may still be processed in various ways within the application's frontend or backend. This often, as in our case, leads to injection attacks such as XSS and SQLi or other vulnerability types.

Fuzzing for parameters

Orwa Atyat (GodfatherOrwa)

The endpoints I always prioritize for parameter fuzzing are search pages, API endpoints, product pages, and profile or settings pages. Basically, anywhere the application is already handling query or body parameters. If an endpoint already accepts one parameter, there's a possibility it accepts others that aren't exposed in the UI.

For wordlists, I'd recommend SecLists Discovery/Web-Content parameter lists or Assetnote's parameter wordlists. And when filtering results, again, `-fs 0` is your best friend, it strips out empty responses so you're only looking at parameters the application actually reacted to.

Fuzz with different content types!

It's also worth noting that API endpoints may process request parameters differently, depending on the Content-Type request header. Sometimes, a parameter may return no visible changes when supplied as a query parameter but may be processed when sent as a JSON blob in the request body. Testing all possibilities remains essential.

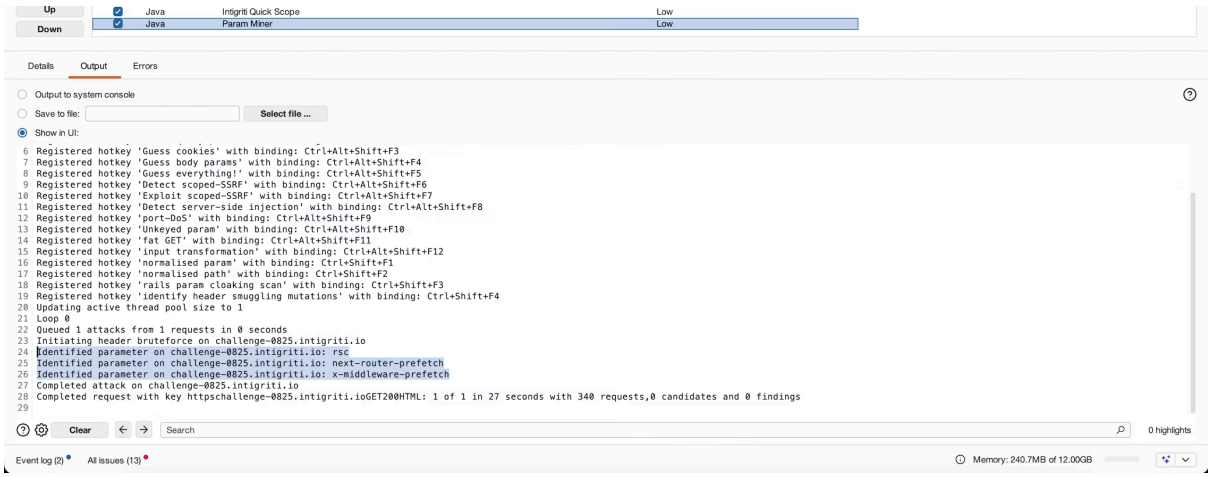
Fuzzing for request headers

Security vulnerabilities or misconfigurations originating from request headers are often overlooked. Developers introduce custom headers to implement access controls or other features. For us, it's the

ideal place to fuzz, as we can identify request headers that aren't supposed to be accessible to external clients and help us find new vulnerabilities, such as Cache Poisoning, or other quirky behavior that could lead to one.

Similar to fuzzing parameters, fuzzing request headers goes paired with analyzing changes in response elements such as response length, content type, and response time. However, it is important to note that some applications won't show any difference in response unless the header is also supplied with a corresponding value.

One of the best ways to enumerate request headers is with Param Miner, a Burp Suite plugin that rapidly fuzzes a list of headers and analyzes responses.



Bruteforcing request headers with Param Miner

Identifying interesting headers

Orwa Atyat (GodfatherOrwa)

When fuzzing for headers, these are the ones I always include in my list, as they frequently reveal hidden behavior:

- `X-Forwarded-For` and `X-Real-IP` can be used for IP spoofing or bypassing IP-based access controls
- `X-Debug` and `X-Dev` sometimes activate debug mode, exposing stack traces or verbose error output with sensitive data (such as environment variables)
- `X-Internal` can toggle internal-only feature flags
- `X-Admin-Token` in rare cases is used for admin impersonation
- `X-HTTP-Method-Override` allows switching the request method server-side, which can bypass method-based restrictions
- `X-Original-URL` can override the request path entirely, useful for bypassing path-based access controls

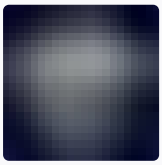
The important thing to watch for isn't just status code changes, pay close attention to response size and timing. A small difference in either can indicate that the server is processing the header differently, even if the visible response looks the same.

Conclusion

Web fuzzing remains one of the most effective techniques for uncovering hidden attack surfaces on modern web applications. In this article, we've covered what web fuzzing is, the common mistakes that render it ineffective, how to select the right wordlists and tooling, and how to fuzz web apps to return meaningful results.

So, you've just learned something new about web fuzzing... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, possibly earn a bounty on your next submission!

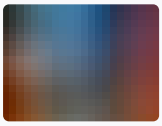
START HACKING ON INTIGRITI TODAY



AUTHOR

Ayoub

Senior security content developer



CO-AUTHOR

Orwa Atyat

External security researcher Orwa Atyat (GodfatherOrwa)