Chapter 14. Denial of Service

Perhaps one of the most popular types of attacks, and the most widely publicized, is the distributed denial of service (DDoS) attack. This attack is a form of denial of service (DoS), in which a large network of devices flood a server with requests, slowing down the server or rendering it unusable for legitimate users.

DoS attacks come in many forms, from the well-known distributed version that involves thousands or more coordinated devices, to code-level DoS that affects a single user as a result of a faulty regex implementation, resulting in long times to validate a string of text. DoS attacks also range in seriousness from reducing an active server, to a functionless electric bill, to causing a user's web page to load slightly slower than usual or pausing their video mid-buffer.

Because of this, it is very difficult to test for DoS attacks (in particular, the less severe ones). Most bug bounty programs outright ban DoS submissions to prevent bounty hunters from interfering with regular application usage.

By the end of this chapter, you will understand a wide variety of common DoS attacks, several advanced forms of DoS attack—and the concepts shared behind all DoS attacks. With this knowledge, you will be able to attack web applications by developing your own DoS attacks.

WARNING

Because DoS vulnerabilities interfere with the usage of normal users via the application, it is most effective to test for DoS vulnerabilities in a local development environment where real users will not experience service interruption.

With a few exceptions, DoS attacks usually do not cause permanent damage to an application, but do interfere with the usability of an application for legitimate users. Depending on the specific DoS attack, sometimes it can be very difficult to find the DoS sink that is degrading the experience of your users.

Regex DoS

Regex DoS–based vulnerabilities are some of the most common forms of DoS in web applications today. Generally speaking, these vulnerabilities range in risk from very minor to medium, often depending on the location of the regex parser.

Taking a step back, regular expressions are often used in web applications to validate form fields and make sure the user is inputting text that the server expects. Often this means only allowing users to input characters into a password field that the application has opted to accept, or only put a maximum number of characters into a comment so the full comment will display nicely when presented in the UI.

Regular expressions were originally designed by mathematicians studying formal language theory to define sets and subsets of strings in a very compact manner. Almost every programming language on the web today includes its own regex parser, with JavaScript in the browser being no exception.

In JavaScript, regex are usually defined one of two ways:

```
const myregex = /username/; // literal definition
```

```
const myregex = new regexp('username'); // constructor
```

A complete lesson on regular expressions is beyond the scope of this book, but it is important to note that regular expressions are generally fast and very powerful ways of searching or matching through text. At least the basics of regular expressions are definitely worth learning.

For this chapter, we should just know that anything between two forward slashes in JavaScript is a regex literal: `/test/`.

Regex can also be used to match ranges:

```
const lowercase = /[a-z]/;
const uppercase = /[A-Z]/;
const numbers = /[0-9]/;
```

We can combine these with logical operators, like OR:

```
const youori = /you|i/;
```

And so on. You can test if a string matches a regular expression easily in JavaScript:

```
const dog = /dog/;
dog.test('cat'); // false
dog.test('dog'); // true
```

As mentioned, regular expressions are generally parsed really fast. It's rare that a regular expression functions slowly enough to slow down a web application. That being said, regular expressions can be specifically crafted to run slowly. These are called *malicious regexes* (or sometimes *evil regexes*), and they are a big risk when allowing users to provide their own regular expressions for use in other web forms or on a server. Malicious regexes can also be introduced to an application accidentally, although it is probably a rare case when a developer is not familiar enough with regex to avoid a few common mistakes.

Generally speaking, most malicious regex are formed using the plus "+" operator, which changes the regex into a "greedy" operation. Greedy regex test for one or more matches rather than stopping at the first match found.

A malicious regex will result in backtracking whenever it finds a failure case. Consider the regex: `/^((ab)*)+$/`. This regex does the following:

1. At the start of the line defines capture group `((ab)*)+`.
2. `(ab)*` suggests matching between 0 and infinite combinations of `ab`.
3. `+` suggests finding every possible match for #2.
4. `$` suggests matching until the end of the string.

Testing this regex with the input `abab` will actually run pretty quickly and not cause much in the way of issues. Expanding the pattern outwards, `abababababab` will also run quite fast. If we modify this pattern to `abababababababa` with an extra "a", suddenly the regex will evaluate slowly, potentially taking a few milliseconds to complete. This occurs because the regex is valid until the end, in which case the engine will backtrack and try to find combination matches:

- `(abababababababa)` is not valid.
- `(abababababababa)(ba)` is not valid.
- `(abababababa)(baba)` is not valid.
- Many iterations later: `(ab)(ab)(ab)(ab)(ab)(ab)(ab)(a)` is not valid.

Essentially, because the regex engine is attempting to exhaustively try all possible valid combinations of `(ab)`, it will have to complete a number of combinations equal to the length of the string before determining the string itself is not valid (after checking all possible combinations).

A quick attempt of this technique using a regex engine is shown in Table 14-1.

Table 14-1. Regex (malicious input) time to match (`/^((ab)*)+$/`)

Input	Execution time
abababababababababababa (23 chars)	8 ms
ababababababababababababa (25 chars)	15 ms
abababababababababababababa (27 chars)	31 ms
ababababababababababababababa (29 chars)	61 ms

As you can see, the input constructed for breaking the regex parser using this evil or malicious regex results in doubling the time for the parser to finish matching with every two characters added. This continues onward and eventually will easily cause significant performance reduction on a web server (if computed server side) or totally crash a web browser (if computed client side).

Interestingly enough, this malicious regex is not vulnerable to all inputs, as Table 14-2 shows.

Table 14-2. Regex (safe input) time to match (`/^((ab)*)+$/`)

Input	Execution time
abababababababababab (22 chars)	<1 ms
abababababababababababab (24 chars)	<1 ms
abababababababababababababab (26 chars)	<1 ms
abababababababababababababababab (28 chars)	>1 ms

This means that a malicious regular expression used in a web application could lie dormant for years until a hacker found an input that caused the regex parser to perform significant backtracking. Thus, it would seem to appear out of nowhere.

Regex DoS attacks are more common than you might think and can easily take down a server or render client machines useless if the proper payload can be found. Note that OSS is often more vulnerable to malicious regex, as few developers are capable of detecting malicious regex.

Logical DoS Vulnerabilities

With logical DoS vulnerabilities, server resources are drained by an illegitimate user. As a result, legitimate users experience performance degradation or loss of service (as shown in Figure 14-1).

Regex is an easy introduction to DoS vulnerabilities and exploiting DoS because it provides a centralized starting place for researching and attempting attacks (anywhere a regex parser is present). It is important to note, however, that due to the expansive nature of DoS, DoS vulnerabilities can be found in almost any type of software!

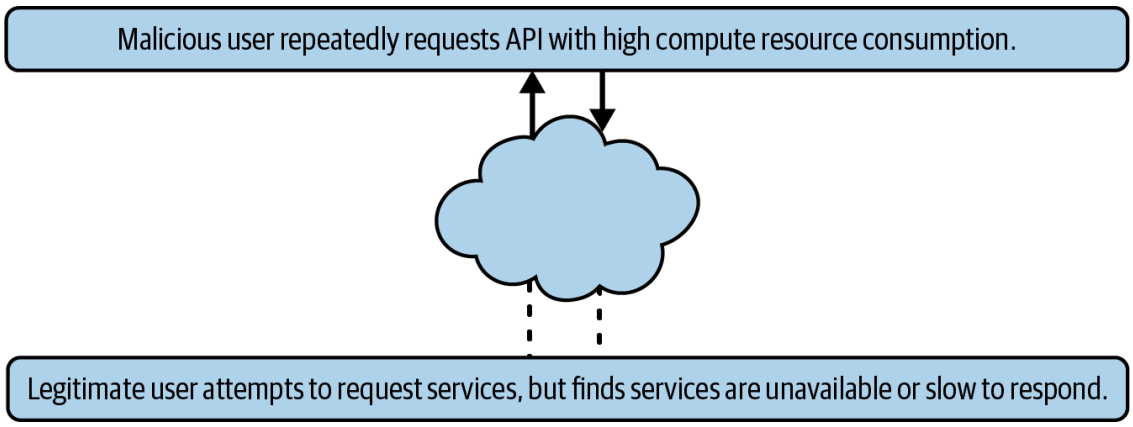

Figure 14-1. Server resources are drained by an illegitimate user, creating performance degradation or loss of service for legitimate users

Logical DoS vulnerabilities are some of the hardest to find and exploit, but they appear more frequently in the wild than expected. These require a bit of expertise to pin down and take advantage of, but after mastering techniques for finding a few, you will probably be able to find many.

First off, we need to think about what makes a DoS attack work. DoS attacks are usually based around consuming server or client hardware resources, leaving them unavailable for legitimate purposes. This means that we want to first look for occurrences in a web application that are resource intensive. A nonextensive list could be:

- Any operation you can confirm operates synchronously
- Database writes
- Drive writes
- SQL joins
- File backups
- Looping logical operations

Often, complex API calls in a web application will contain not only one but multiple of these operations. For example, a photo-sharing applica-

tion could expose an API route that permits a user to upload a photo. During upload, this application could perform:

- Database writes (store metadata regarding the photo)
- Drive writes (log that the photo was uploaded successfully)
- SQL join (to accumulate enough data on the user and albums to populate the metadata write)
- File backup (in case of catastrophic server failure)

We cannot easily time the duration of these operations on a server that we do not have access to, but we can use a combination of timing and estimation to determine which operations are longer than others. For example, we could start by timing the request from start to finish. This could be done using the browser developer tools.

We can also test if an operation occurs synchronously on the server by making the same request multiple times at once and seeing if the responses are staggered. Each time we do this, we should script it and average out perhaps one hundred API calls so our metrics are not set off by random differences.

Perhaps the server gets hit by a traffic spike when we are testing, or begins a resource-intensive cron job. Averaging out request times will give us a more accurate measure of what API calls take significant time.

We can also approximate the structure of backend code by closely analyzing network payloads and the UI. If we know the application supports these types of objects:

- User object
- Album object (user HAS album)
- Photo object (album HAS photos)
- Metadata object (photos HAVE metadata)

we can then see that each child object is referenced by an ID:

```
// photo #1234
{
```

```
    image: data,
    metadata: 123abc
  }
```

We might assume that `users`, `albums`, `photos`, and `metadata` are stored in different tables or documents depending on if the database used is SQL or NoSQL. If, in our UI, we issue a request to find all metadata associated with a user, then we know a complex join operation or iterative query must be running on the backend. Let's assume this operation is found at the endpoint `GET /metadata/:userid`.

The scale of this operation varies significantly depending on the way a user utilizes the application. A power user might require significant hardware resources to perform this operation, whereas a new user would not.

We can test this operation and see how it scales, as shown in Table 14-3.

Table 14-3. `GET /metadata/:userid` by account archetype

Account type	Response time
New account (1 album, 1 photo)	120 ms
Average account (6 albums, 60 photos)	470 ms
Power user (28 albums, 490 photos)	1,870 ms

Given the way the operation scales based on user account archetype, we can deduce that we could create a profile to eat up server resource time via `GET /metadata/:userid`. If we write a client-side script to reupload the same or similar images into a wide net of albums, we could have an account with 600 albums and 3,500 photos.

Afterward, simply performing repeated requests against the endpoint `GET /metadata/:userid` would result in significant reduction in server performance for other users unless the server-side code is extremely robust and limits resources on a per-request basis. It's possible these requests would just timeout, but the database would likely still be coordinating resources even if the server software times out and doesn't send the result back to the client performing the request.

That's just an example of how logical DoS attacks are found and exploited. Of course, these attacks differ by case—hence the "logical DoS" as defined by the particular application logic in the application you are exploiting.

Distributed DoS

With DDoS, server resources are drained by a large number of illegitimate users. Because they are requesting en masse, they may even be able to perform standard requests. At scale this will drown out server resources for legitimate users (see Figure 14-2).

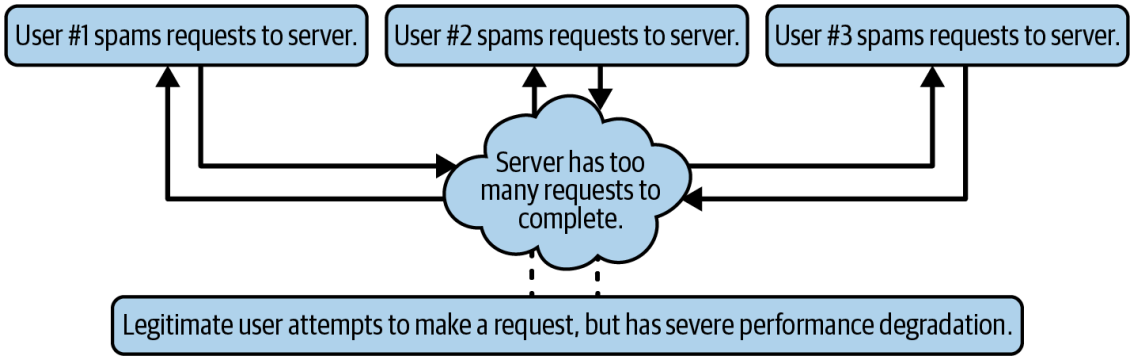

Figure 14-2. DDoS server resources are being drained by a large number of illegitimate users en masse

DDoS attacks are a bit outside of the scope of this book to cover comprehensively, but you should be familiar at least conceptually with how they work. Unlike DoS attacks where a single hacker is targeting either another client or a server to slow them down, distributed attacks involve multiple attackers. The attackers can be other hackers or networked bots (*botnets*).

Theoretically, these bots could exploit any type of DoS attack, but on a wider scale. For example, if a server utilizes regex in one of its API endpoints, a botnet could have multiple clients sending malicious payloads to the same API endpoint simultaneously. In practice, however, most DDoS attacks do not perform logical or regex-based DoS and instead attack at a lower level (usually at the network level instead of at the application level). Most botnet-based DDoS attacks will make requests directly against a server's IP address, not against any specific API endpoint. These re-

quests usually are User Datagram Protocol traffic in an attempt to drown out the server's available bandwidth for legitimate requests.

As you would imagine, these botnets are not usually devices all owned by a single hacker, but instead are devices that a hacker or group of hackers has taken over via malware distributed on the internet. They are real computers owned by real people but with software installed that allows them to be controlled remotely. This is a big issue because it makes detecting the illegitimate clients much harder. (Are they real users?)

If you gain access to a botnet, or can simulate a botnet for security testing purposes, it would be wise to try a combination of both network- and application-level attacks.

Any of the aforementioned DoS attacks that run against a server are vulnerable to DDoS. Generally speaking, DDoS attacks are not effective against a single client, although perhaps seeding massive amounts of regex-vulnerable payloads that would later be delivered to a client device and executed could be in scope for DDoS.

Advanced DoS

While the most common types of DoS attack have already been covered within this chapter, due to the nature of DoS, many obscure and less common forms exist. Any automated effort that degrades or impedes the useability of a web application can be considered denial of service, regardless of its origin.

YoYo Attacks

A more modern method of attacking a web application via DoS, YoYo attacks make use of autoscale features baked into large cloud providers like Amazon Web Services (AWS) or Google Cloud Platform (GCP). The theory behind an autoscaler is that a web application can make use of limited compute resources (e.g., low CPU, low RAM) in times where there is little traffic. This saves on cost. However, when loads increase, the cloud

provider will automatically increase the amount of compute resources allocated to a given web application—alongside the resulting hardware bill.

YoYo attacks rely on this autoscale functionality, whereby an attacker floods a web application with a huge number of requests in a short period of time—invoking the autoscaler to increase hardware resources. The cloud provider scales up compute resources for the target web application, only to have the traffic suddenly stop. At this point, the cloud provider will soon have to begin decreasing compute resources.

Once the compute resources are scaled back to baseline, the attacker begins another onslaught of requests to invoke an upscale. This process is repeated over and over again.

YoYo attacks not only cause degradation of user experience for the end user during the periods at which the hardware reallocation is occurring, but they also cause an influx in cloud hosting costs. These costs can be significant if the attack is run over any substantial period of time.

Compression Attacks

Applications that allow file uploads are often vulnerable to application-layer DoS attacks involving malformed files. Consider a video-hosting website akin to YouTube: such a website accepts uploads in the form of video files but then runs the video file through a compressor and optimizer (e.g., FFMPEG) prior to serving it to end users. This process reduces bandwidth strain and ensures that all videos are in browser-compatible file formats.

During the compression and optimization steps, however, the raw video file provided by the end user must be parsed. In this case, a carefully crafted video file could draw significantly on server-side compute resources, which would degrade the user experience for all others making use of the service.

These forms of DoS vulnerability are becoming more common as an increasing number of websites rely on user-submitted data. For example,

CVE-2021-38094 describes such an attack against the open source FFMPEG (version 4.1) video compression and conversion library.

In the CVE-2021-38094 scenario, a video file run through FFMPEG that contains data outside of the expected bounds for a valid input file will trigger an integer overflow bug within the function `filter_sobel()` contained in the file *libavfilter/vf_convolution.c*. Once this overflow occurs, a series of problematic function calls follow, leading to very high CPU and memory consumption and possible crash of the FFMPEG process. This, of course, degrades or breaks the service relying on FFMPEG for other legitimate users.

Compression-based DoS attacks are often hard to find because many web applications rely on third-party software for dealing with file uploads. Tools like FFMPEG and ImageMagick simplify the upload and handling of complex files for developers—but also expose significant surface area for attackers to craft and utilize malformed upload payloads.

Proxy-Based DoS

DoS attacks often require significant compute resources on behalf of the attacker to be performed. When attacking a web application with significant compute resources, it can often come at a cost in order to hit the web application with enough requests to degrade user performance.

In these cases, some attackers look for mechanisms to steal compute resources from legitimate web applications and retarget the web application's network requests at a target website. This reduces the cost of a DoS attack and makes it more difficult for the victim's web application to determine who is causing the attack.

Consider a search engine like Google, Bing, or Yahoo! These search engines have huge swaths of computing resources dedicated to crawling and caching data from new websites.

These crawlers scan newly found websites, grade them, and store bits of data in their own databases. As the crawlers operate, they require a small

amount of compute resources for each page requested. Interestingly, most web application developers purposefully allowlist crawlers in their *robots.txt* file in order to get their application indexed so that more legitimate users can find them.

Because these search engine crawlers have significant compute resources out of the gate, it is possible at times to "trick" the crawler into pushing too many crawling requests toward a target website for DoS purposes. By utilizing new domains and subdomains, which proxy requests to a target web application, a hacker can trick crawlers into indexing the same content over and over again. Scaled up to thousands or millions of subdomains, it's possible to significantly degrade a web application's performance via this methodology without having to pay for all of the attack's compute resources yourself.

I have dubbed these forms of DoS attack *proxy-based* DoS because the hacker uses an intermediary service such as a search engine to send traffic to the target web application rather than sending the traffic to the target themselves. As web applications—especially web applications used for testing and crawling other web applications—evolve, this form of attack will become more and more common among hackers.

Summary

Classic DDoS attacks are by far the most common form of DoS, but they are just one of many attacks that seek to consume server resources so that legitimate users cannot. DoS attacks can happen at many layers in the application stack—from the client, to the server, and in some cases even at the network layer. These attacks can affect one user at a time or a multitude of users, and the damage can range from reduced application performance to complete application lockout.

When looking for DoS attacks, it's best to investigate which server resources are the most valuable, then start trying to find APIs that use those resources. The value of server resources can differ from application to application, but could be something standard, like RAM/CPU usage, or

more complicated, like functionality performed in a queue (user a → user b → user c, etc.).

While typically only causing annoyance or interruption, some DoS attacks can leak data as well. Be on the lookout for logs and errors that appear as the result of any DoS attempts.