Chapter 32. Defending Against DoS

DoS attacks usually involve the use of system resources, which can make detecting them a bit difficult without robust server logging. It can be difficult to detect a DoS attack that occurred in the past if it came through legitimate channels (such as an API endpoint).

As such, a first measure against DoS-style attacks should be building up a comprehensive enough logging system in your server that all requests are logged alongside their time to respond. You should also manually log the performance of any type of async “job"-style functions, such as a backup that is called through your API but runs in the background and does not generate a response once it completes. Doing this will allow you to find any attempts (accidental or malicious) at exploiting a DoS vulnerability (server side) that would have otherwise been difficult and time-consuming.

As discussed in Chapter 14, DoS attacks are structured with one or more of the following results in mind:

- Exhaust server resources
- Exhaust client resources
- Request unavailable resources
- Deny access to resources

The first two are easier to exploit without direct knowledge of the server or client ecosystem. We need to consider all four of these potential threats when building a plan for mitigating DoS threats.

Protecting Against Regex DoS

Regex DoS attacks might be the easiest form of DoS to defend against, but they require prior knowledge of how the attacks are structured (as shown in "Regex DoS"). With a proper code review process, you can prevent regex DoS sinks (*evil* or *malicious* regex) from ever entering your codebase.

You need to look for regex that perform significant backtracing against a repeated group. These regex usually follow a form similar to `(a[ab]*)+`, where the `+` suggests to perform a greedy match (find all potential matches before returning), and the `*` suggests to match the subexpression as many times as possible.

Because regular expressions can be built on this technology, but without DoS risk, it can be time-consuming and difficult to find all instances of evil regex without false positives. This is one case where using an OSS tool to either scan your regular expressions for malicious segments or using a regex performance tester to manually check inputs could be greatly useful. If you can catch and prevent these regex from entering your codebase, you have completed the first step toward ensuring your application is safe from regex DoS.

The second step is to make sure there are no places in your application where a user-supplied regex is utilized. Allowing user-uploaded regular expressions is like walking through a minefield and hoping you memorized the safe-route map correctly. It will take a huge coordinated effort to maintain such a system, and it is generally an all-around bad idea from a security perspective. You also want to make sure that no applications you integrate with utilize user-supplied regex or make use of poorly written regular expressions.

Protecting Against Logical DoS

Logical DoS is much more difficult to detect and prevent than regex DoS. Much like regex DoS, logical DoS is not exploitable under most circum-

stances unless your developers accidentally introduce a segment of logic that can be abused to eat up system resources.

That said, systems without exploitable logic do not typically fall prey to logical DoS. However, it is possible because DoS is measured on a scale instead of binary evaluation, and a well-written app could still be hit by a logical DoS (assuming the attacker has a huge amount of resources in order to overwhelm the typically performant code).

As a result, we should think of exposed functionality in terms of DoS risk—perhaps high/medium/low. This makes more sense than vulnerable/secure, as DoS relies on consumption of resources that is difficult to categorize compared to other attacks like XSS, which is completely binary. Either you have an XSS exploit or you do not. Period.

With DoS, you may find it extremely difficult to exploit code, easy to exploit code, and some in between. A user on a powerful desktop might not notice an exploitable client-side function, but perhaps a user on an older mobile device would. Generally speaking, we call the extremely difficult to exploit code "safe" and the other two categories "vulnerable." It is safer for us to err on the side of caution while evaluating the security of an application.

To protect against logical DoS, we need to identify the areas of our codebase in which critical system resources are used.

Protecting Against DDoS

Distributed denial of service attacks (DDoS) are much more difficult to defend against than DoS attacks that originate from a single attacker. While single-target DoS attacks often target a bug in application code (like an improperly written regex or a resource-hogging API call), DDoS attacks are usually much simpler by nature.

Most DDoS attacks on the web originate from multiple sources but are controlled by a centralized source. This is orchestrated via a single attacker or group of attackers who distribute malware by some channel.

This malware runs in the background of legitimate PCs and may come packaged with a legitimate program. The legitimate PCs can be controlled remotely due to a backdoor the malware provides, enabling them to be used en masse to do the hacker's bidding.

PCs are not the only devices vulnerable to this type of attack. Both mobile devices and IoT devices (routers, hotspots, smart toasters, etc.) can be targeted, often more easily than desktop computers.

Regardless of the devices compromised and used in the DDoS attack, the devices en masse are referred to as a *botnet*. The word *botnet*, as you may notice, comprises the words *robot* and *network*, suggesting a network of robots used to do someone's bidding (generally for evil).

DDoS attacks usually do not target logic bugs, but instead attempt to overwhelm the target by sheer volume of legitimate-looking traffic. By doing this, actual users are kept out, or the application experience for legitimate users is slowed dramatically.

DDoS attacks cannot be prevented. However, they can be mitigated in a number of ways. The easiest way is to invest in a *bandwidth management* service. These services are developed by many vendors on the market but ultimately perform analysis on each packet as it passes through their servers. The services run well-established scans on the packet to determine if it appears to be coming in a malicious pattern or not. If a packet is determined to be malicious, it will not be forwarded to your web server. These bandwidth management services are effective because they are capable of intercepting large quantities of network requests, while your application's infrastructure (especially in hobby and small business applications) is likely not.

Additional measures can be implemented in your web application architecture to mitigate DDoS risk. One common technique is known as *blackholing*, whereby you set up a number of servers in addition to a main application server (see Figure 32-1).

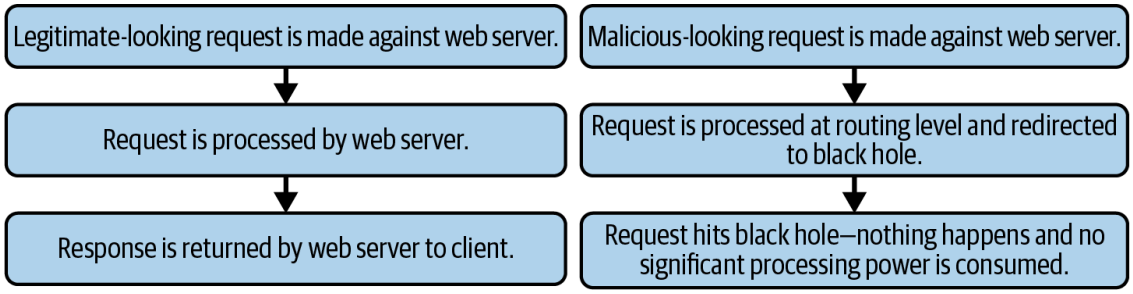

Figure 32-1. Blackholing is a strategy for mitigating DDoS attacks against your web application

Suspicious-looking (or repeated) traffic is sent to a *blackhole* server, which appears to function like your application server, but performs no operations. Legitimate traffic is routed to your legitimate web application server as usual. Unfortunately, while black holes are effective at rerouting malicious traffic, they may also reroute legitimate traffic if not targeted with sufficient accuracy. Blackholes do a good job against small DDoS attacks but do not perform well against large-scale DDoS attacks.

With any of these techniques, keep in mind that oversensitive filters will likely block legitimate traffic as well. Because of this, it is ideal to have deep metrics on the usage patterns of your legitimate users prior to implementing any aggressive DDoS mitigation measures.

Summary

DoS attacks come via two major archetypes: single attacker (DoS) and multiple attackers (DDoS). Most, but not all, DDoS attacks are performed by overwhelming server resources rather than via bug exploitation. Because of this, countermeasures for DDoS may also cause difficulty for legitimate users. Single-attacker DoS attacks, on the other hand, can be mitigated by smart application architecture that prevents users from being able to take over application resources for a long period of time. Regular-expression-based DoS attacks can be mitigated by implementing a static analysis tool (like a linter) to scan regular expressions in your codebase and warn if any appear to be “evil” syntactically.

Because of their general ease of exploitation, DoS-style attacks are rampant throughout the web. Even if you don't expect your application to be a target of DoS attacks, implement anti-DoS mitigations once you can afford it just in case you become a target in the future.