Separating the wheat from the chaff

Winnowing

Attackers impose a high price on your limited resources. We look at some ways to separate the good from the bad.

By Kurt Seifried

As you may have noticed, spam, web-based attacks, and denial-of-service (DoS) attacks have gotten rather bad over the past few years. Currently, spam makes up about 90% of email messages; in other words, for every legitimate email you get, your server has to handle around nine spam messages. For online retailers, the stakes can be much higher. DoS attacks at the height of the shopping season can have a hugely negative impact. So, what can you do to deal with these problems?

Reward, Ignore, or Punish?

When it comes to dealing with behavior and changing it, you have some options: You can reward good behavior, punish bad behavior, ignore the bad behavior (if you're lucky), or respond with some combination thereof. If things get really ugly (e.g., the server is starting to tip over), you can begin taking desperate measures, which are probably better than having the server die (well, usually).

Dropping Connections

One way to deal with overload is simply to start dropping connections early; the sooner you can stop dealing with an attacker, the sooner you can deal with real clients. Unfortunately, for some services like SSH, if you're using password-based authentication, you need to give people some time to type in the password. An alternative is to drop connections randomly, the idea being that by dropping lots of connections, clients will get another chance to connect. Unfortunately, if it's a large DoS, the attacker's connections will outnumber legitimate ones and, you'll be back where you started. So, just dropping connections probably won't work too well.

Progressive Timeouts

A common and effective way to prevent brute-force attacks is to implement a progressive timeout – if you get the password wrong, you have to wait a second; if you get it wrong again, you have to wait two seconds, then four seconds, and so on. This reduces the number of attempts an attacker can make. To make this really effective, you can apply the timeout not just to the current session but to all con-

nections from that IP address or network. The downside of this method is that people stuck behind a NAT box (like everyone on public WiFi networks or many corporate environments) will be affected by other users who fail to log in (intentionally or otherwise).

Reputation-Based Systems

This strategy lends itself more to email-based systems because of the (usually) centralized nature of email servers (legitimate ones). Reputation services like SORBS and Spamhaus [1] simply allow your mail servers to ignore badly behaved clients (known spammers, residential ISPs, etc.). The upside is that these services are cheap and easy to use, both computationally and operationally. The bad news is that these approaches tend to be both overreaching (innocent systems get included) and underreaching (spammers create new systems quickly, and it takes time for them to get on the lists). Also, you need server software that supports reputation-based lookups, which is mostly supported in email servers and not much else.

Greylisting

Greylisting [2] is simple and effective for protocols that support error conditions that basically say, "come back later," like email. This won't work as well for interactive protocols like SSH and the various WWW protocols – who wants to wait more then a second for a web page to load? The reason greylisting works is that it makes clients behave properly, and it increases the amount of resources required to connect and send an email. So, apart from email, this solution is largely out.

Hashcash

What if you could really make attackers pay, computationally or otherwise, with either minimal or no effect on legitimate clients? Hashcash [3] is one such attempt. Many computational problems have solutions that are easy to verify but hard to create. A simplistic example would be to generate two prime numbers, multiple them, and send the result to the remote end, which then has to factor the number and send the result back. Another class of solutions is to pick a secret value, hash it, and send the hash to the client (the added benefit is that the work needed on the server is very minimal).

Hashcash uses a pretty clever and simple technique to create problems that are easy for the server to create and verify but computationally difficult for the client to solve. Instead of requiring a complete match of the hash, Hashcash generates a SHA-1 hash that it sends to the client. The client then has to find a partial collision. For example, the server hashes the word "secret" and sends "fc-683cd9ed1990ca2ea10b84e5e6f-ba048c24929" to the client. The client then has to find something that partially matches and sends the result back (e.g., "00000000000000000000000000000000000 0024929"). The advantage here is that you can easily dial up the work load needed to solve the problem by requiring a more complete match (each bit required would be in theory twice as much work). Even better, a variety of languages (C, Java, Python, C#, .NET, JavaScript, even a shell script) have working implementations of Hashcash.

If Hashcash is so great, why does no one use it? For one thing, you would need to retrofit a ton of software to support it, and that never happens in a hurry. Hashcash also has significant downsides. It uses SHA-1, which, as you learned in my column on password storage [4], actually makes it easier for attackers to use brute force because SHA-1 is so optimized for speed.

A better choice would be something like bcrypt, which includes a work function to increase the time needed. Unfortunately, these approaches have the effect of punishing legitimate clients heavily; for example, if Hashcash were to become widely used for email, anyone running an email list server would have to buy a ton of server hardware just to calculate Hashcash values so they could send email. However, if Hashcash were combined with reputation-based systems and whitelisting of known good clients (e.g., the aforementioned mailing list server), it would solve a lot of problems but would be somewhat complex to implement (okay, that's a bit of an understatement).

A Bouncer at the Door

If you can't make people pay to get in, maybe you can put a bouncer at the door and only let the nice people in. In web terms, you've probably run into CAPTCHAs [5] – those mangled words that you have to type in – but CAPTCHAs have several problems. The first is that many have been broken by automated systems, and the second is that, by the time someone gets to the CAPTCHA, they have already established a connection. Not to mention the annoyance for users: I'm averaging two to three attempts now to get them right.

An alternative method is to require clients to behave like clients and not like automated bots. In web terms, one trick to accomplish this is to place JavaScript or Flash content in a web page that then makes a request to the server. If the client requests a web page and then fails to make the request before asking for more pages, you have either a bot or someone who doesn't have JavaScript or Flash installed (and you can ask them nicely to enable it for the site). Although this is not an ideal solution, because you will be locking out some- security-conscious users, it can go a long way to preventing bots from hammering your site.

Conclusion

There are no easy ways to prevent DoS attacks, especially if the bad guys have access to 10,000-node botnets. However, over time, the number of infected hosts will only get worse (e.g., all those Windows XP machines that never get updates, embedded systems, etc.). One interesting potential side effect of things like Hashcash is that by requiring infected clients to do a lot of computation to send spam or launch attacks, people might actually notice their machine is slow enough to warrant fixing it. You can only hope. ■■■

INFO

[1] DNS blacklists: *http://en.wikipedia.org/wiki/Comparison_of_DNS_blacklists*

[2] Greylisting: *http://greylisting.org/*

[3] Hashcash: *http://hashcash.org/*

[4] "Security Blanket" by Kurt Seifried, *Linux Magazine*, November 2011, pg. 46, *http://www.linuxpromagazine.com/Issues/2011/132/Security-Lessons-Password-Storage*

[5] reCAPTCHA: *http://www.google.com/recaptcha*