Algorithmic Complexity Attacks and Low-Bandwidth DoS

Tom Magerlein

December 13, 2017

Abstract

In an era where large botnets provide a means to knock services offline by simply overwhelming them with traffic in distributed denial of service (DDoS) attacks, a sometimes-overlooked alternative threat is the algorithmic complexity attack. Such attacks make use of carefully-constructed requests that cause high-cost operations to be executed on the target, slowing it down so as to be unusable. Because these attacks may be carried out with many fewer attacking machines and requests, they are a variety of low-bandwidth denial of service attack. Without the flood of packets that comes with a DDoS attack, algorithmic complexity attacks can also be more difficult to detect as they are happening, and to defend against. In this paper, the general principles of such attacks are examined in the context of several examples, and an overview of potential mitigation techniques is presented.

Introduction One of the more troublesome security issues that has grown with the expansion of the internet is that of denial-of-service (DoS) attacks, where adversaries will attempt to prevent legitimate users from being able to access a target website or service. There is an almost infinite variety of such attacks, but the one which has gotten a lot of press in recent years is the distributed DoS (DDoS) attack, where a large number of attacking computers, typically a botnet, will flood a target with traffic in an attempt to prevent users requests from being served. In a large DDoS attack, targets may receive traffic at hundreds of gigabits per second.

There exists another class of DoS attacks, so-called low-bandwidth DoS (LBDoS) attacks, which take an alternative approach. Rather than tying up all of a service's bandwidth with incoming connections, such attacks attempt to use small payloads to disrupt targets by exhausting any of various resources on the computers processing them. Even within these attacks there is a great deal of variety, including such things as the infamous "ping of death" and slow HTTP attacks. In the case of the former, mishandling of a malformed packet causes the receiver to crash; in the latter, HTTP connections are kept alive for much longer than intended by sending incomplete headers, tying up threads on the server while they wait for the headers to be completed. In both cases, relatively small payloads can easily make servers unavailable to legitimate users.

One particular variety of LBDoS attack, the algorithmic DoS, or simply algorithmic complexity attack, takes aim at the design of the code running on target machines. These

attacks, which were first formally described in [1], attempt to force algorithms running on the target to exhibit their worst-case time complexity by giving carefully crafted inputs. This type of attack will attempt to exhaust all available CPU time or I/O bandwidth, preventing other requests from being served. It is these attacks which will be considered in this paper.

To the Community My first exposure to the idea that this type of attack existed was in a passing reference during an algorithms lecture in fall 2016. I didn't think much of it at the time, but I have since become more interested in security, and particularly in the security implications of these types of design decisions. As for the importance of their study, low-bandwidth DoS attacks are known to have been used by attackers in the wild, e.g. in some of the attacks described in [2], and the information necessary to carry out simple ones against poorly-defended targets is available online.

Common Attack Surfaces While in principle any data structure or algorithm which has a bad worst-case running time that runs on user input could be the target of an algorithmic complexity attack, there are a few which are particularly common. Hash tables have been the subject of extensive study with regard to such attacks, as they are extremely common in practice, especially in the context of network-facing software. Under normal circumstances, a hash table will provide average-case $O(1)$ insertion and lookup, provided it is not overloaded. However, if an attacker can cause hash collisions to occur with high frequency, they can force many operations on the table to take $O(n)$ time. One example of such an attack is proposed in [3], where collisions are engineered in a hash table within Linux' Netfilter, within a module called conntrack which manages open connections. If the attacker knows the hash function being used in advance, generation of inputs which will cause collisions is not terribly difficult.

Another common target for algorithmic complexity attacks is regular expression evaluators, in regular expression denial-of-service (ReDoS) attacks. The features necessary to evaluate advanced regular expressions (in future referred to as regexps, to avoid confusion with regular expressions without such features), i.e. those with various features like backreferences, result in evaluation algorithms which require exponential time for some regexps and inputs [4]. Backreferences can essentially only be resolved using backtracking methods, and regexps which include a repeated group, where that group can break up the input in multiple ways. Thus, on a long input matching that group (but not matching the whole regexp, so that it has to be matched many times), it is possible for there to be exponentially many divisions of the input which must be explored. Vulnerabilities to this kind of attack appear or have appeared in many applications; CVE-2017-15010 and CVE-2016-4055 are examples.

One particularly significant possible target of algorithmic complexity attacks is network

intrusion detection systems (NIDS). These systems are used to help protect systems and networks against many varieties of attack, so disabling such a system would be a good first step towards breaking into the network it protects. In [5], the authors demonstrate an attack on the Snort NIDS which is a variety of the above-mentioned regular-expression DoS attacks. Most NIDS in some way make use of rule- or signature-matching in identifying potentially malicious packets. These rules often take the form of regexps, and so can be susceptible to backtracking attacks in the same way. Smith et al. claim that they were able to disable Snort entirely using a single packet every couple of seconds using this method.

Defense and Risk Mitigation In designing a system to be resilient against algorithmic complexity attacks, there exist a couple of possible strategies. The first and most reliable is to avoid data structures and algorithms which have bad worst-case behavior altogether. Clearly, this prevents the exploitation of worst-case behavior for denial of service. Unfortunately, doing this may have other drawbacks, like making the handling of non-malicious requests slower. If using algorithms with good worst-case complexity is not practical, building systems in such a way that it is difficult for attackers to engineer LBDoS payloads may be the next best approach. While this does not necessarily guarantee safety, it is a step in the right direction.

A classic example of optimizing for worst-case complexity comes from sorting. Quicksort has worst-case complexity $O(n^2)$, and since naïve implementations choose pivots extremely predictably — often the first or middle element, or the median of the first, middle, and last elements — it is not difficult to generate inputs which will trigger this worst-case behavior. In place of quicksort, there exist numerous other average-case $O(n \log n)$ sorting algorithms whose worst-case complexity is also $O(n \log n)$, for instance merge sort [6, pp. 35-37]. By choosing such an algorithm instead of quicksort, there is no input which will cause $O(n^2)$ behavior, thus preventing the attack. In some cases, this will also resolve ReDoS attacks. Many, but not all, regexps which are susceptible to ReDoS can be rewritten into a form where they are not. Alternatively, the authors of [5] describe an algorithm which uses memoization to avoid reevaluating pieces of regexps repeatedly, which they demonstrated was able to completely prevent their original attack against Snort.

If choosing a algorithm which has a good worst-case complexity is not practical, an alternative is to make it difficult or impossible to predict what inputs will cause the algorithm to exhibit its worst-case behavior. Randomness is a critical tool for doing this. Going back to the quicksort example, another way of defending against algorithmic complexity attacks is to choose the pivots randomly from the elements of the list. This does not in principle prevent the $O(n^2)$ behavior of quicksort (though it does make it enormously unlikely to occur) [6, pp. 179-181], but it does make it so that attackers cannot engineer lists which are hard to sort: such lists require that the pivot be the largest or smallest element at every

step, and there is no way to force this to be the case if the pivot is chosen randomly.

A similar, if more complex, approach is used in practice to defend hash tables against engineered collisions. Each hash table has a value associated with it, which is chosen randomly and used when computing the hashes of elements much in the same way a password salt would be [3]. Without knowing this value, an attacker would in theory be unable to determine the hashes of inputs to the table, so it would be impossible to intentionally choose inputs which cause collisions.

In practice, this type of randomization may be less effective, particularly in the case of randomized hash tables. While in the case of quicksort a new random pivot is chosen at every step, the random value for a hash table must remain the same as long as the table exists — if it changes, inputs will no longer hash into the same location in the table, rendering the table useless. This provides a weakness which can be exploited. In [7] and [3], the authors describe techniques for guessing the random value of a hash table, and even the type of hash function being used if that is unknown, by observing collisions that occur during normal operation of the hash table. Once the random value of a hash table is determined, it is just as susceptible to engineered hash collisions as a non-randomized hash table. Thus, while randomization makes attacks against hash tables more difficult, it is not a perfect solution.

Conclusion All of that said, how serious a threat are algorithmic complexity attacks? Clearly, such vulnerabilities exist and can be exploited, so the risk associated with them is nonzero. That said, these attacks require considerably more technical knowledge to discover and exploit, so more traditional DDoS attacks, when they are effective, are likely to be the most common threat. But that is not to say that they should not be considered as a possibility when designing web-facing systems, as it only takes one vulnerability to bring such a system down.

References

[1] S. A. Crosby and D. S. Wallach. "Denial of Service via Algorithmic Complexity Attacks." In: *USENIX Security Symposium*. 2003, pp. 29–44.

[2] T. J. O'Connor. *The Jester Dynamic: A Lesson in Asymmetric Unmanaged Cyber Warfare*. Tech. rep. SANS Institute, Dec. 2011.

[3] N. Bar-Yosef and A. Wool. "Remote algorithmic complexity attacks against randomized hash tables". In: *E-business and Telecommunications* 23 (2007). Ed. by J. Filipe and M. S. Obaidat, pp. 162–174.

[4] J. Kirrage, A. Rathnayake, and H. Thielecke. "Static Analysis for Regular Expression Denial-of-Service Attacks". In: *Network and System Security: 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings*. Ed. by J. Lopez, X. Huang, and R. Sandhu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 135–148. ISBN: 978-3-642-38631-2. DOI: 10.1007/978-3-642-38631-2_11. URL: https://doi.org/10.1007/978-3-642-38631-2_11.

[5] R. Smith, C. Estan, and S. Jha. "Backtracking Algorithmic Complexity Attacks against a NIDS". In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. Dec. 2006, pp. 89–98. DOI: 10.1109/ACSAC.2006.17.

[6] T. H. Cormen et al. *Introduction to Algorithms*. 3rd ed. The MIT Press, 2009. ISBN: 0262033844, 9780262033848.

[7] R. J. Tobin and D. Malone. "Hash pile ups: Using collisions to identify unknown hash functions". In: *2012 7th International Conference on Risks and Security of Internet and Systems (CRiSIS)*. Oct. 2012, pp. 1–6. DOI: 10.1109/CRISIS.2012.6378946.