Beyond SynFloods: Guarding Web Server Resources from DDoS Attacks

Srikanth Kandula kandula@mit.edu
Shantanu Sinha ssinha@mit.edu
Dina Katabi dk@mit.edu
Matthias Jacob mjacob@cs.princeton.edu

Problem. Denial-of-Service attacks on web servers take many forms. In this paper, we look at a new breed of application-level attacks. An attacker compromises a large number of dummy clients (by means of a worm, virus or Trojan horse) and causes the clients to flood the web server with well-formed HTTP requests that download large files or generate complex database queries. Such requests cause the web server to expend costly server resources like sockets, disk bandwidth, database sub-system bandwidth and worker processes on these dummy users. As a result, performance seen by legitimate users will degrade, eventually leading to denial of service. These attacks are hard to counter as the *malicious requests are indistinguishable from legitimate requests* at the server. Further, the dummy requests arrive from a large number of geographically distributed machines; thus, they cannot be filtered on source IP addresses or arrival patterns.

Prior work has looked at network/transport level DDoS attacks such as SYN flood and bandwidth attacks [1] and proposed a few solutions [2], [3]. We assume that a subset of these solutions protect a web server from both SYN flood and bandwidth attacks and focus on application-level attacks.

Approach. Despite the distributed nature of clients participating in a DDoS attack, typically a small group of human operators initiates and manages the attack. By requiring the clients to interact with their human operator before they access server resources, we limit the speed of the DDoS attack and make the human attacker a shared bottleneck.

In our system, a web-server can be in either of two modes, NORMAL and UNDER_ATTACK. The server behavior is unchanged in NORMAL mode. When the web server perceives resource depletion beyond an acceptable limit it shifts to the UNDER_ATTACK mode. In this mode, the server continues to serve connections that were established during the NORMAL mode. The server asks new clients to solve a puzzle that is easy to solve by a human but difficult to compute by a machine, before providing access to the system. Depending upon the desired level of protection, the puzzle could be a variation on the following text: "We are suspecting a DDoS attack on Foo. To access Foo, type in the text box KEY_i after replacing the number 6 by 2" or a URL embedded in an image - a CAPTCHA [4]. Keys are large strings chosen randomly from a large and sparsely populated space, such that, it is unlikely that a client will be able to guess a valid key.

One concern is that the user might not be willing to solve the puzzle. In this case, our system behaves like current systems which handle these attacks by asking the user to "come back later". The user can still choose to ignore the puzzle and "come back later"; solving the puzzle enables the user immediate access to the server.

Challenges. Incorporating a human in the loop has been used to counter automated user account creation and e-mail spam. However, using this approach to prevent a DDoS attack on web server resources is different due to the following challenges.

1. The puzzle should be sent and validated without allocating any TCB's or sockets at the server, while ensuring correct TCP congestion control semantics.
2. The client's TCP stack and the browser should not be modified.
3. Mechanism should be transparent to web caches.
4. A normal user would have to manually enter the key just once per browsing session, potentially consisting of multiple TCP connections.
5. If the system is experiencing a flash crowd rather than a DDoS attack, the mechanism should be benign.
6. Validation should be independent of the source IP address as malicious users could share IP with ordinary users due to NAT or spoofing.
7. One puzzle allows access to only one client, so it is useless for an attacker to solve a puzzle and distribute it to a large number of worms.
8. Switching from NORMAL to UNDER_ATTACK mode (and vice versa) should be inexpensive and transparent to ongoing sessions.
9. Mechanism should work when requests are handled by a server farm.

Ongoing Implementation. We are working on an implementation running Apache on Linux to address the above challenges - some aspects of which are discussed below. Currently, we use CAPTCHA-like images(1-2 pkts) as puzzles but are experimenting with natural language puzzles which are smaller in size. The puzzle is returned in an HTML form. To solve the puzzle, a human user types the answer(key) and submits the form creating an HTTP request containing `GET /validate?answer=KEY`$_i$. On a new connection request (i.e. SYNs to the web server), we want to send a puzzle and validate the key without allocating any TCB's or sockets at the server. The server responds to SYN packets with a SYN Cookie. The client receives the SYN cookie, increases its congestion window to two packets, transmits a SYNACKACK and the first data packet that usually contains the HTTP request. The kernel at the server end does not create a new socket upon completion of the TCP handshake. Instead the SYNACKACK packet is discarded. When the server receives the client's data packet, if the header of the HTTP request is not of the form `GET /validate?answer=KEY`$_i$, then this packet begins an HTTP session and is not an attempt at validating a key. The server replies with a new puzzle (1-2 pkts) as the HTTP response and immediately resets the connection (using the TCP RST flag). Otherwise, the kernel checks the cryptographic validity of the key. If the check succeeds, a socket is established and the request is delivered to the application. Note that this scheme preserves TCP congestion control semantics and prevents attacks that hog TCB's and sockets by establishing connections that exchange no data.

The above scheme creates the following per-session overhead when the server is in UNDER_ATTACK mode; two hashes to validate the answer, a few memory accesses to look at HTTP headers, fetching a puzzle and sending it to the client. To ensure that a user needs to solve a puzzle once even if the session contains multiple HTTP 1.0 connnections, the server uses a cookie at the client. Again note that the attacker cannot mount an attack by replicating a cookie because each cookie is mapped to a single key and the server constrains the number of connections using the same key to be small (e.g. four).

Assuming that worms are not equipped with OCR software or natural language parsers, the rate at which malicious clients gain access is equal to the rate at which the human operators solve puzzles. To prevent attackers from distributing one puzzle's answer to a herd of clients, the server constrains the number of active TCP connections per key.

REFERENCES

[1] CERT, "Cert advisory ca-2001-19 code red worm exploiting buffer overflow in iis indexing service dll," 2002.
[2] Livio Ricciulli, Patrick Lincoln, and Pankaj Kakkar, "Tcp syn flooding defense," .
[3] Thomer M. Gil and Massimiliano Poletto, "MULTOPS: A Data-Structure for bandwidth attack detection," pp. 23–38.
[4] L. von Ahn, M. Blum, N. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," .