THROTTLING DDoS ATTACKS

Saraiah Gujjunoori, Taqi Ali Syed, Madhu Babu J, Avinash D
Radhesh Mohandas and Alwyn R. Pais
Information Security Lab, Department of Computer Engineering, NITK, Surathkal, Karnataka, India

Keywords: Source throttling, Distributed denial of service, Integer factorization, CPU stamps, Request stamping.

Abstract: Distributed Denial of Service poses a significant threat to the Internet today. In these attacks, an attacker runs a malicious process in compromised systems under his control and generates enormous number of requests, which in turn can easily exhaust the computing resources of a victim web server within a short period of time. Many mechanisms have been proposed till date to combat this attack. In this paper we propose a new solution to reduce the impact of a distributed denial of service attack on a web server by throttling the client's CPU. The concept of source throttling is used to make the client pay a resource stamp fee, which is negligible when the client is making a limited number of requests but becomes a limiting restriction when he is making a large number of requests. The proposed solution makes use of the integer factorization problem to generate the CPU stamps. We have packaged our solution as an API so that existing web applications can easily deploy our solution in a layer that is transparent to the underlying application.

1 INTRODUCTION

In modern web applications, the web client makes a request which takes very little effort to compose, but causes the server to process lots of data and compose the response. This disparity in the computation efforts of the server and the client is usually of an order of magnitude and works very well in the favour of an attacker when he modifies a web client to launch an application level attack against a server. Hence a bunch of compromised machines in the hands of an attacker can easily launch a denial of service attack against even the biggest server farms and succeed. The fundamental flaw in such a web transaction is that the entire cost of processing the request falls more or less on the server end and very little is shared by the client. Hence learning from the postal system, we can impose a cost to a web client to make each such request. This cost can be collected in different ways and in our work we propose to collect it in terms of CPU cycles and call them CPU stamps which have to accompany each request. Collection of these CPU stamps has a source throttling effect on the web application, thereby reducing the impact of the attack by an order of magnitude.

For such a scheme to work, we have to exploit an algorithm that takes lot of CPU cycles to calculate a stamp (which the client has to pay) but very few cycles to verify the stamp (which the server computes). In our solution we propose to use the popular integer factorization problem as the mathematical complexity to impose and tune the computation difficulty on the client. Any system that uses a server and a client that can be subjected to a denial of service attack can use our solution, but for our study we take a web application server which has some unique needs making it difficult to use existing solutions.

A web server is designed to accept requests from multiple clients across the internet which makes it difficult to filter requests based on IP address which can be easily accomplished in hardware. In most cases the clients are unauthenticated and even if we impose certain kind of authentication, most applications allow creation of users on the internet which involves little or no manual verification of end-user information provided. Even if we manually verify all the end-users of the application, a single user who bears malice in his heart can inflict all the necessary damage to bring the web application to a grinding halt by sending valid requests through his authenticated account by tweaking the web client a

Gujjunoori S., Ali Syed T., Babu J. M., D. A., Mohandas R. and R. Pais A. (2009).
THROTTLING DDoS ATTACKS.
In *Proceedings of the International Conference on Security and Cryptography*, pages 121-126
DOI: 10.5220/0002229001210126

bit. Our solution aims to make the web server responsive to legitimate users with reasonable overhead and tolerable failure rates when the server is actively being subject to a denial of service attack.

In this paper we propose a technique to generate CPU stamps, force an attacker to recompute the stamp when the source address or the request changes. We provide a heuristic to guard replay attack by discarding requests at the beginning of processing.

The rest of the paper is organised as follows. Section 2 will describe about the problem. Related work is presented in section 3. Section 4 shows the working of the proposed solution. Section 5 describes about the application of the proposed solution. Implementation results will be presented in section 6 and section 7 will conclude this paper.

2 PROBLEM DESCRIPTION

To clarify our idea, we will use the following hypothetical profile and work with some hypothetical numbers. Suppose http://www.mysearch.com is designed to handle a maximum of 4000 searches per second (sps). A search requires the application server to talk to a database server. Serving a search request is more expensive than serving a static page. During peak day times the traffic reaches around 3000 requests per second (rps) and drops to 500 in the early hours of morning. There are roughly 2000 searches and 1000 main page accesses every second. A valid search request for a nonexistent keyword in the database is probably the most expensive as it misses all caches and in the worst attack, the attacker creates the keyword dynamically. Now a distributed attack is launched against the website and it starts receiving 40,000 sps. The website will be able to respond to only 1 out of 11 requests and the number of valid users who get a response will be lesser than 10%. Now instead of wasting valuable resources to respond to the fake requests, we propose a solution to filter them out so that after a period of time, the site will be able to service at least 2500 genuine rps.

3 RELATED WORK

Adam Back (Adam Back, 2002) proposed a Hashcash based solution for Denial of Service. He computes a token which can be used as a proof-of-work.

XiaoFeng Wang (XiaoFeng Wang and Michael K. Reiter, 2003) proposed a puzzle mechanism called *puzzle auction*. In this, the auction lets each client determine the difficulty of the puzzle it solves and allocates server resources first to the client that solved the difficult puzzle when the server is busy.

T.Aura (Tuomas Aura, Pekka Nikander and Jussipekka Leiwo, 2000) showed how the robustness of authentication protocols against DoS attacks can be improved by asking the client to commit its computational resources to the protocol run before the server allocates its resources.

M.Abadi (Martin Abadi, Mike Burrows, Mark Manasse, and Ted Wobber, 2003) proposed memory bound functions for use in cryptographic puzzles. Drew Dean (Drew Dean and Adam Stubblefield, 2001) implemented puzzles for protection of SSL against DoS attacks.

Ari Juels (Ari Juels and John Brainard, 1999) had proposed a cryptographically based countermeasure against connection depletion attacks like TCP SYN flooding.

4 PROPOSED SOLUTION

Prime Factoring is the act of splitting an integer into the unique set of primes (factors) which, when multiplied together, form the original integer. No good algorithms exist to solve this problem in polynomial time and the best algorithm which solves this problem in less complexity is general number field sieve in $O(\exp((64/9b)^{1/3}.(\log b)^{2/3}))$ for a b-bit integer.

Threshold Value is the number of requests that a server can handle without straining its resources. It is defined as a predetermined percentage of the maximum number of requests that a server can handle.

4.1 Proposed Solution

Notations:

- *N*: An integer and a product of two primes
- *p, q*: prime factors of *N*.
- *Ndigits, pdigits, qdigits*: Number of digits in *N,p,q*.

The sequence of operation is as follows.

- A client sends a request to the web server for a webpage.
- The Server starts a session and sends '*N*' along with the JavaScript to factorize it.
- The Client computes *p* and *q* values and sends '*N*, *p*, *q*' values to the server.

- The server verifies whether the product of the factors sent by the client is equal to the *'N'* value sent by the server (*N=p*q)*. If this condition is not satisfied or the values not sent by the client, the server will drop the request.

4.1.1 Description of our Solution

When the server is facing normal flow of traffic, we do not interfere with the web application. When the number of requests arriving at the server crosses the threshold value, our solution is invoked and the server starts sending '*N*' to all the clients.

A user using a web browser will experience a momentary delay when the JavaScript calculates the values '*p'* and '*q'* on his client machine but then his request gets through when presented to the server.

An attacker who is using a malicious client will not send these '*p'* and '*q'* values and his requests get dropped. If he now modifies his client to read the JavaScript and compute '*p'* and '*q'*, the number of requests that he can send will drop down drastically. If the distributed attack sustains or deepens, we can increase the number of digits in '*N'* and this will throttle the malicious clients further without increasing any load on the server.

4.1.2 Countermeasures Against the Throttling

The strength of our solution lies in the mathematical complexity of the integer factorization problem. Since no algorithms exist to solve this problem in polynomial time the attacker will not try to optimize this computation but will try to get around the computation by finding a hole in the protocol. In this section we discuss various scenarios where the attacker actively modifies his malicious client and tries to tweak the hosts launching the distributed attack and the countermeasures that we need to have in place to defend against such modifications.

Case 1: At this case the attacker observes that the server is sending the same '*N*' for all requests. He computes the prime factors once and appends these factors to every request. This is a form of replay attack. To counter this we will dynamically generate '*p'*, the first factor of the prime from a variable that changes with time.

Case 2: Now if the attacker has full control over the zombies which he is using to launch the DDoS attack, he can compute the value of '*p'* on one system and propagate it quickly to all the remaining systems and launch a replay attack in the time slot. To guard against this our solution generates *'q'* dynamically as a function of client's IP address. Fixed cost functions are used to generate these values dynamically so that there is no over head on the server. Such attacks are extremely unlikely as the communication delay to propagate the computation to all the systems will be comparable to the cost of computing it at the individual node itself.

Case 3: He may try to pre compute the primes in the entire prime space. As per Table 1, the number of primes (*NP*) increases with the number of digits and becomes too huge and storage becomes a limiting factor and such attacks are difficult with zombie machines which have limited amount of resources. The communication overhead of drawing it from a central database will make such attacks infeasible.

Case 4: The attacker might try to guess the value of '*N*' from its previous values, the IP address, the server time and other variables that he can find out. He may even get access to the exact code or algorithm that we use to generate '*p'* and '*q'*. So we select a random combination of primes from the set of primes and design the mapping functions such that the selected primes are uniformly chosen across this combination. We further change this combination periodically to prevent the attacker from tabulating the combination restriction the usefulness of such tabulation further.

Case 5: In this extreme case when the attacker has access to fast interconnects and resources if he successfully launches the attack in *case 2* and also has access to all the mapping functions in *case 4*, he may find out that the value of '*q'* is reused on individual nodes. To ward off this attack we can compute '*q'* from a different source with sufficient entropy or flush the combination at a much faster rate. As a result of this flushing all existing connections will need to be reset and hence we would not suggest this to be applied unless needed.

Case 6: In this case the attacker satisfies the condition *N=p*q*, but the factors sent by the attacker are bluffed. To counter this we generate '*q'* from the IP address using a hash function that is changed periodically. The server then verifies that the '*q'* value sent in the request is not bluffed by recalculating it from the source IP after verifying '*N'*. The drawback of this algorithm is that once the value of '*q'* is computed by the client, he can reuse the same in further requests thereby necessitating a periodic change of algorithm to compute '*q'*. In another countermeasure to this type of attack the server maintains a table in which the '*N*' values sent to every client for every request has to be stored. But this will be a memory storage load on the server and can be a problem at the server if the attacker is

sending large number of rps. We have implemented this step for the sake of completeness but this part of the algorithm is not activated unless this particular attack profile is matched.

4.1.3 Algorithms

The two algorithms that can be used to generate the '*p'* and '*q'* dynamically are presented in this section. The first algorithm is for selecting the '*q'* which is based on the client's IP address (*cip*) and the second algorithm is for selecting the '*p'* which is based on the time in milliseconds from when the server is booted to the current time (*st*).

The server should select the '*Ndigits*' based on the number of requests coming to the server and this should be varying between 8 to16 digits. Based on the selection of '*Ndigits*', *'pdigits'* and *'qdigits'* should be selected on the following criteria.

$$pdigits = Ndigits/2,\ qdigits = (Ndigits+1)/2 \qquad (1)$$

The above criterion (1) is to ensure that there are no easy factors served out. To implement the dynamically generation of '*p'* and '*q'* values we first stored pre-computed primes between 4 and 8 digits in a two dimensional array called *primes.* The number of primes (*NP*) in each digit (i.e., 4, 5, 6, 7, and 8) is tabulated below in Table 1.

Table 1: Number of primes in each digit.

Ndigits	4	5	6	7	8
NP	1061	8363	68906	586081	5096871

Algorithm 1: Generate q

```
GenerateQ(qdigits,NP,primes,cip)
{
 cip="A.B.C.D"
   ipMapValue=2^24*A+2^16*B+2^8*C+D
 qMapValue=(ipMapValue) mod NP
 return primes[qdigits][qMapValue]
}
```

In the above algorithm the *cip* represents the clients IP address and it is in the form of *A.B.C.D*. *ipMapValue* is the value that is generated from the client IP address and this value is unique for each client. From the total set of *primes*, we choose a random combination and call it '*selectedPrimes'* array. So the '*q'* value generated for each client will be unique. The '*NP*' in the above algorithm represents the number of primes in '*selectedPrimes*' array.

Algorithm 2: Generate p

```
GenerateP(pdigits,NP,primes,st)
{
   pMap=(st) mod NP
 return primes[pdigits][pMap]
}
```

In the above algorithm the *st* represents the number of milliseconds since the server boot. As *st* differs for every millisecond the '*p'* value generated will be unique for each client.

5 APPLICATION

Now continuing the http://www.mysearch.com example that we used earlier in section 2, we are serving 2000 dynamic plus 1000 static rps during the normal traffic profile. In the worst case, the attacker is sending an additional 40,000 void searches. So we are receiving 42,000 search requests and 1000 static requests. We now respond to the attack by prepending a JavaScript that does the stamping computation and sets a valid stamp in the HTTP header to every request received. So out of the 42,000 search requests, we should be able to respond with a static redirect page with the JavaScript for at least 39,000 of them. A genuine user should be able to get this new page by repeating his requests to the main page. Now the attacker usually would have stored the old request and will not be able to modify his request to include the stamp and all his requests will be redirected only to a static page. The genuine users will be using popular browsers and will be able to get a new page with the JavaScript within a few refreshes. Now the JavaScript does a second long computation for every request that the browser sends and the user will be able to continue working with a tolerable latency. When this new search request comes in with the stamp, we treat it with higher priority and open up the server resources to it. Eventually we will converge to a point around which we will be able to serve all the stamped search requests while using the remaining resources to serve the requests without a stamp with a new page. If we are serving 2000 searches, we can still serve 20,000 static pages. Over a period of time the genuine users cross the filter after a few retries. If the attacker is able to reconfigure his resources to calculate the stamp, then he will be able to send only a fraction of his original requests. He will be able to send in something like 400 search requests instead of 40,000 and the damage will be contained if not eliminated. Now, if the attacker attaches an invalid stamp, he may pass through the initial filter, but we can still verify the stamp with a fraction of the cost of serving the request and drop it pretty early in the pipeline. Once the attack stops, we can remove the JavaScript attachment and restore normalcy. We also have the flexibility of changing the JavaScript to invalidate any pre-computation efforts by the

attacker. Further a simple inexpensive hardware can be installed to send this static redirect, reducing the load on the server almost completely.

6 IMPLEMENTATION RESULTS

In this section we present the results obtained by implementing the proposed solution.

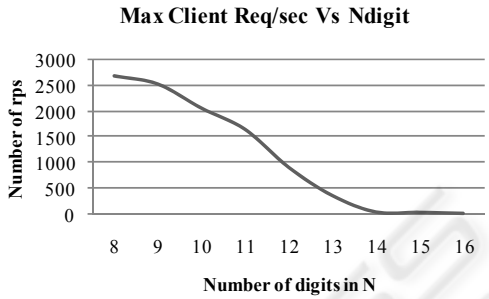
- *Clients:* Intel core2 Duo CPU with processor speed 3.00 GHz and 2.99 GHz, 2 GB RAM, Windows XP professional operating system.
- *Server:* Intel Xeon Quad CPU, processor speed 3.60 GHz each, 4 GB RAM, Win 2003 server.

To study the effectiveness of the proposed solution, we developed a website that represents a typical portal. We developed a version incorporating the solution (*WSolution*) and other without it (*WoSolution*). The *WoSolution* website consists of 27 pages each having multiple database connections in it. The *WSolution* website consists of an extra HTML page with of a JavaScript which makes the client browser to factorize '*N*'. When a request comes to the website without the proper cookies, this static page is served and the client is then redirected to the proper web page. The server retrieves the number of rps from the Windows performance counters, and when it exceeds a threshold value the server invokes the proposed solution and starts sending out a *'N*' value using cookies in each response. The client responds with the factors and the server will verify it. If the proposed condition (*N=p*q)* is satisfied by the client, the server will then respond with the actual page and the '*N'* value in the session variable will be flushed out.

Table 2: Latency in milliseconds of browsers to calculate factors.

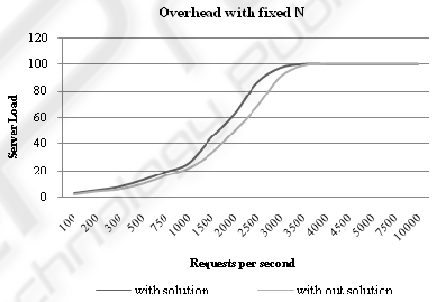
Ndigits	IE	Mozilla	Opera	Chrome	CCB
5	0	0.2	0	0.2	0
6	0	0.2	0	0.2	0
7	0	0.2	0	0.4	0
8	0	1.8	0	2.2	0.01
9	6	1.8	3	2.2	0.05
10	34	16	22	18	0.05
11	44	23	28	25	0.07
12	265	134	147	147	0.43
13	318	163	169	173	5.60
14	2512	1269	1347	1398	6.62
15	4975	2475	2659	2866	44.6
16	49820	25069	19859	28173	67.5

We measured the latencies of different web browsers to factor the primes and tabulated it in Table 2. CCB in the above table represents a custom command line browser written in C#.Net. By measuring the latencies of the JavaScript computation on most popular browsers, we observe that a 14 digit '*N'* values give about 2 seconds latencies on the browsers, which should be tolerable to an end-user.



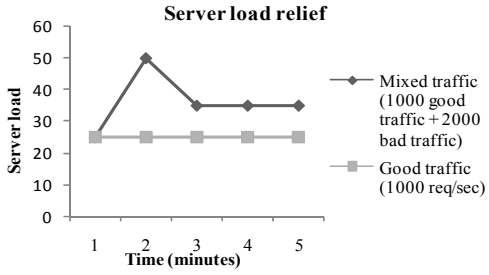
Graph 1: Overhead on requests/sec with fixed N.

From Graph 1 we can see that the overhead of our solution is not significant when '*N*' is fixed.



Graph 2: Server load relief on server.

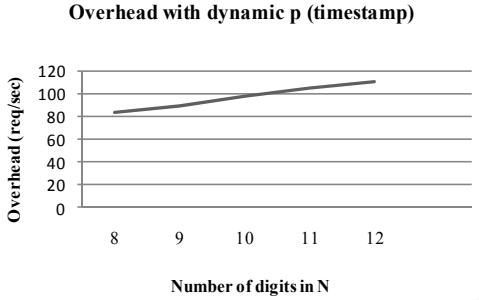
In the Graph 2 we sent a steady 1000 rps through client 1 which can compute '*p'* and '*q*'. This causes a steady load of about 25% on the server. After 1 minute we sent an attack traffic of about 2000 rps from client 2 which does not compute '*p'* and '*q*'. The server load increases till our threshold limit is hit. Then our solution is invoked and we start serving '*N'*. The server drops the attack traffic and treats them as static pages. We can see that the server load falls down once our solution kicks in.



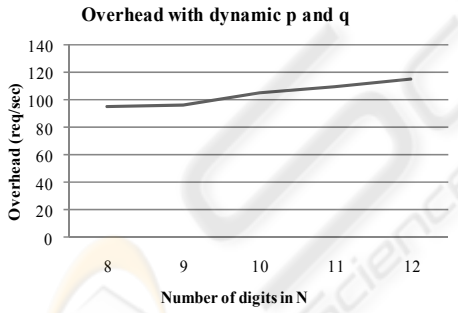
Graph 3: Max client requests/sec Vs Number of digits in N.

From Graph 3 we can clearly the throttling effect on the malicious clients where in the total number of requests that they can send can be made to fall down by a factor of 100 by increasing the number of digits to 14. This means that an attacker who has compromised 100 zombies will be able to inflict only the damage possible by one such machine thereby loosing the effectiveness of the attack.

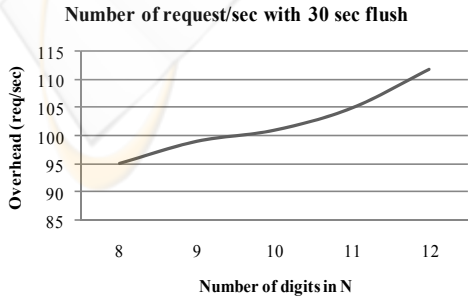
Now in graphs numbered 4, 5, and 6 we list the overhead on the server in terms of reduced number of dynamic requests that are served at 100% CPU load. As expected this loss does not increase significantly even when we are generating the values '*p'* and '*q'* dynamically for every request. The overhead is bounded by 120 rps in all cases which is less than 4% for our application.



Graph 4: Overhead at server when p is generated dynamically.



Graph 5: Overhead at server when p and q are dynamic.



Graph 6: Overhead on number of rps with p and q generated dynamically from selected primes.

7 CONCLUSIONS

In this paper we proposed an approach to contain a DDoS attack at the application level. We came up with a solution to generate stamps on the web browsers that are easily verifiable at the server. Our algorithm is further tuneable to throttle the client CPU when the attack deepens. We come up with a strategy to distinguish between genuine requests and malicious requests and drop the later much earlier in the transaction during a DDoS attacks. We proposed two different algorithms for dynamic generation of primes. There is no considerable overhead on the web server because of deploying the proposed solution. As a whole we saw less than 5% overhead on the server to verify the timestamp and serve the additional JavaScript.

REFERENCES

L. Stein, 2002. The World Wide Web security faq, http://www.w3.org/Security/Faq/

Rob Malda, 1997. Slash dot faq. http://slashdot.org/faq/slashmeta.shtml#sm600

B. Clifford Neuman and Stuart G. Stubblebine, 1993. *A Note on the Use of Timestamps as Nonce.*

Roger M. Needham and Michael D.Schroeder, 1978. *Using encryption for authentication in large networks of computers.*

Adam Back, 2002. *Hashcash - A Denial of Service Counter-Measure.*

XiaoFeng Wang and Michael K. Reiter, 2003. *Defending against denial-of-service attacks with puzzle auctions.*

Tuomas Aura, P Nikander and Jussipekka Leiwo, 2000. *DOS-Resistant Authentication with Client Puzzles.*

Martin Abadi, M Burrows, Mark Manasse, and T Wobber, 2003. *Moderately hard, memory-bound functions.*

Drew Dean and Adam Stubblefield, 2001. *Using client puzzles to protect TLS.*

Cynthia Dwork and Moni Naor, 1992. *Pricing via processing or combatting junk mail.*

Ari Juels and John Brainard, 1999. *Client puzzles: A cryptographic countermeasure against connection depletion attacks.*

A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, 1996. *Handbook of Applied Cryptography.*

D. J. Bernstein, 2006. *Integer factorization.*