Content Delivery Networks: Protection or Threat?

Sipat Triukose, Zakaria Al-Qudah, and Michael Rabinovich
{*sipat.triukose*}{*zakaria.al-qudah*}{*michael.rabinovich*}*@case.edu*

EECS Department
Case Western Reserve University

Abstract. Content Delivery Networks (CDNs) are commonly believed to offer their customers protection against application-level denial of service (DoS) attacks. Indeed, a typical CDN with its vast resources can absorb these attacks without noticeable effect. This paper uncovers a vulnerability which not only allows an attacker to penetrate CDN's protection, but to actually use a content delivery network to amplify the attack against a customer Web site. We show that leading commercial CDNs – Akamai and Limelight – and an influential research CDN – Coral – can be recruited for this attack. By mounting an attack against our own Web site, we demonstrate an order of magnitude attack amplification though leveraging the Coral CDN. We present measures that both content providers and CDNs can take to defend against our attack. We believe it is important that CDN operators and their customers be aware of this attack so that they could protect themselves accordingly.

1 Introduction

Content Delivery Networks (CDNs) play a crucial role in content distribution over the Internet. After a period of consolidation in the aftermath of the .com bust, CDN industry is experiencing renaissance: there are again dozens of content delivery networks, and new CDNs are sprouting up quickly.

CDNs typically deploy a large number of servers across the Internet. By doing this, CDNs offer their customers (i.e., content providers) large capacity on demand and better end-user experience. CDNs are also believed to offer their customers the protection against application-level denial of service (DoS) attacks. In an application-level attack, the attacker sends regular requests to the server with the purpose of consuming resources that would otherwise be used to satisfy legitimate end-users' requests. These attacks are particularly dangerous because they are often hard to distinguish from legitimate requests. Since CDNs have much larger aggregate pool of resources than typical attackers, CDNs are supposed to be able to absorb DoS attacks without affecting the availability of their subscribers' Web sites.

However, in this paper, we describe mechanisms that attackers can utilize to not only defeat the protection against application-level attacks provided by CDNs but to leverage their vast resources to amplify the attack. The key mechanisms that are needed to realize this attack are as follows.

– Scanning the CDN platform to harvest edge server IP addresses. There are known techniques for discovering CDN edge servers, based on resolving host names of CDN-delivered URLs from a number of network locations [16].
– Obtaining HTTP service from an arbitrary edge server. While a CDN performs edge server selection and directs HTTP requests from a given user to a particular server, we show an easy way to override this selection. Thus, the attacker can send HTTP requests to a large number of edge servers from a single machine.
– Penetrating through edge server cache. We describe a technique with which the attacker can command an edge server to obtain a fresh copy of a file from the origin even if the edge server has a valid cached copy. This can be achieved by appending a random query string to the requested URL ("<URL>?<random_string>"). Thus, the attacker can *ensure* that its requests reach the origin site.
– Reducing the attacker's bandwidth expenditure. We demonstrate that at least the CDNs we considered transfer files from the origin to the edge server and from the edge server to the user over decoupled TCP connections. Thus, by throttling or dropping its own connection to the edge server, the attacker can conserve its own bandwidth without affecting the bandwidth consumption at the origin site.

Combining these mechanisms together, the attacker can use a CDN to amplify its attack. To this end, the attacker only needs to know the URL of one sizable object that the victim content provider delivers through a CDN. Then, the attacking host sends a large number of requests for this object, each with a different random query string appended to the URL, to different edge servers from this CDN. (Different query strings for each request prevent the possibility of edge servers fetching the content from each other [9] and thus reducing the strength of the attack.) After establishing each TCP connection and sending the HTTP request, the attacker drops its connection to conserve its bandwidth.

Every edge server will forward every request to the origin server and obtain the object at full speed. With enough edge servers, the attacker can easily saturate the origin site while expending only a small amount of bandwidth of its own. Furthermore, because the attacker spreads its requests among the edge servers, it can exert damage with only a low request rate to any given edge server. From the origin's perspective, all its requests would come from the edge servers, known to be trusted hosts. Thus, without special measures, the attacker will be hidden from the origin behind the edge servers and will not raise suspicion at any individual edge server due to low request rate. The aggregation of per-customer request rates across all the edge servers could in principle detect the attacker, but doing this in a timely manner would be challenging in a large globally distributed CDN. Hence, it could help in a post-mortem analysis but not to prevent an attack. Even then, the attacker can use a botnet to evade traceability.

While our attack primarily targets the origin server and not the CDN itself (modulo the cache pollution threat to the CDN discussed in Section 5), it is likely

to disrupt the users' access to the Web site. Indeed, a Web page commonly consists of a dynamic container HTML object and embedded static content - images, multimedia, style sheets, scripts, etc. A typical CDN delivers just the embedded content, whereas the origin server provides the dynamic container objects. Thus, by disrupting access to the container object, our attack will disable the entire page.

This paper makes the following main contributions:

- We present a DoS attack against CDN customers that penetrates CDN caches and exploits them for attack amplification. We show that customers of three popular content delivery networks (two leading commercial CDNs – Akamai and Limelight – and an influential research CDN – Coral) can be vulnerable to the described attack.
- We demonstrate the danger of this vulnerability by mounting an end-to-end attack against our own Web site that we deployed specially for this purpose. By attacking our site through the Coral CDN, we achieve an order of magnitude attack amplification as measured by the bandwidth consumption at the attacking host and the victim.
- We present a design principle for content providers' sites that offers a definitive protection against our attack. With this principle, which we refer to as "no strings attached", a site can definitively protect itself against our attack at the expense of a restrictive CDN setup. In fact, Akamai provides an API that can facilitate the implementation of this principle by a subscriber [12].
- For the cases where these restrictions prevent a Web site from following the "no strings attached" principle, we discuss steps that could be used by the CDN to mitigate our attack.

With a growing number of young CDN firms on the market and the crucial role of CDNs in the modern Web infrastructure (indeed, Akamai alone claims to be delivering 20% of the entire Web traffic [2]), we believe it is important that CDNs and their subscribers be aware of this threat so that they can protect themselves accordingly.

2 Background

In this section we outline the general mechanisms behind content delivery networks and present some background information on the CDNs used in our study.

2.1 Content Delivery Networks

A content delivery network (CDN) is a shared infrastructure deployed across the Internet for efficient delivery of third-party Web content to Internet users. By sharing its vast resources among a large number of diverse customer Web sites, a CDN derives the economy of scale: because different sites experience demand peaks ("flash crowds") at different times, and so the same slack capacity can be used to absorb unexpected demand for multiple sites.

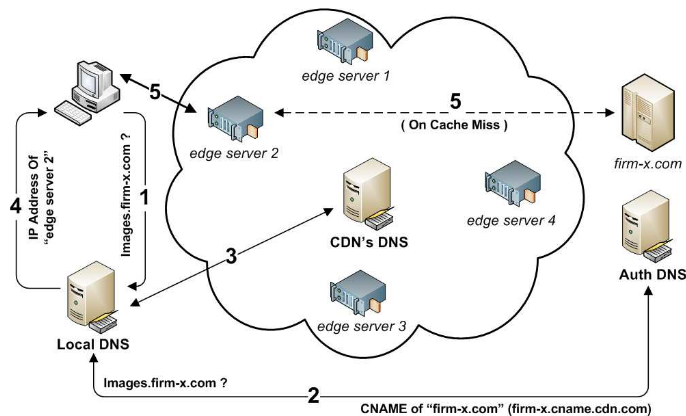


Fig. 1. Content Delivery Network

Most CDNs utilize domain name system (DNS) to redirect user requests from the origin Web sites hosting the content to the so-called *edge servers* operated by the CDN. The basic mechanism is illustrated in Figure 1. If content provider firm-x.com wants to deliver HTTP requests for images.firm-x.com, the provider configures its DNS server to respond to queries for images.firm-x.com not with the IP address of the server but with a so-called canonical name, e.g., "images.firm-x.com.CDN-name.net". The user would now have to resolve the canonical name, with a query that will arrive at the DNS responsible for the CDN-name.net domain. This DNS server is operated by the CDN; it can therefore select an appropriate edge server for this client and respond to the query with the selected server IP address. Note that the content provider can selectively outsource some content delivery to a CDN while retaining the responsibility for the remaining content. For example, the content provider can outsource all URLs with hostname "images.firm-x.com" as described above while delivering content with URL hostnames "www.firm-x.com" from its own origin site directly.

When an edge server receives an HTTP request, it fetches the indicated object from the origin site and forwards it to the client. The edge server also caches the object and satisfies subsequent requests for this objects locally, without contacting the origin site. It is through caching that a CDN protects the origin Web site from excessive load, and in particular from application-level DoS attacks.

2.2 Akamai and Limelight

Akamai [1] and Limelight [11] are two leading CDN providers representing two basic approaches to content delivery. Akamai attempts to increase the likelihood of finding a nearby edge server for most clients and thus deploys its servers in a large number of network locations. Its platform comprises 40,000 servers in 950 networks in 70 countries. Limelight concentrates its resources in fewer "massively provisioned" data centers (around 18 according to their map) and connects each data center to a large number of access networks. This way, it also claims direct connectivity to nearly 900 networks. The two companies also differ in their approach to DNS scalability, with Akamai utilizing a multi-level distributed DNS system and Limelight employing a flat collection of DNS servers and IP anycast [13] to distribute load among them.

Most importantly, either company employs vast numbers of edge servers, which as we will see can be recruited to amplify a denial of server attack on behalf of a malicious host.

2.3 Coral

Coral CDN [8, 4] is a free content distribution network deployed largely on the PlanetLab nodes. It allows any Web site to utilize its services by simply appending a string `".nyud.net"` to the hostname of objects' URLs. Coral servers use peer-to-peer approach to share their cached objects with each other. Thus, Coral will process a request without contacting the origin site if a cached copy of the requested object exists anywhere within its platform. Coral currently has around 260 servers world-wide.

3 The Attack Components

This section describes the key mechanisms comprising our attack and our methodology to verify that CDNs under study support these mechanisms.

3.1 Harvesting Edge Servers

CDN edge server discovery is based on resolving hostnames of CDN-delivered URLs from a number of network locations. Researchers have used public platforms such as PlanetLab to assemble large numbers of edge servers for CDN performance studies [16]. An attacker can employ a botnet for this purpose.

We previously utilized the DipZoom measurement platform [5] to harvest around 11,000 Akamai edge servers for a separate study [18]. For the present study, we used the same technique to discover Coral edge servers. We first compile a list of URLs cached by Coral CDN. We then randomly select one URL and resolve its hostname into an IP address from every DipZoom measurement point around the world. We repeat this process over several hours and discover 263 unique IPs of Coral cache servers. Since according to Coral website, there are around 260 servers, we believe we essentially discovered the entire set.

3.2 Overriding CDN's Edge Server Selection

To recruit a large number of edge servers for the attack, the attacker needs to submit HTTP requests to these servers from the same attacking host, overriding CDN's server selection for this host. In other words, the attacker needs to bypass DNS lookup, i.e., to connect to the desired edge server directly using its raw IP address rather than the DNS hostname from the URL. We found that to trick this edge server into processing the request, it is sufficient to simply include the HTTP host header that would have been submitted with a request using the proper DNS hostname.

One can verify this technique by using *curl* - a command-line tool for HTTP downloads. For example, the following invocation will successfully download the

object from a given Akamai edge server (206.132.122.75) by supplying the expected host header through the "-H" command argument:

```
curl -H Host:ak.buy.com http://206.132.122.75/.../207502093.jpg
```

We verified that this technique for bypassing CDN's server selection is effective for all three CDNs we consider.

3.3 Penetrating CDN Caching

The key component of our attack is to force the attacker's HTTP requests to be fulfilled from the origin server instead of the edge server cache. Normally, requesting a cache server to obtain an object from its origin could be done by using HTTP Cache-Control header. However, we were unable to force Akamai to fetch a cached object from the origin this way: adding the Cache-control did not noticeably affect the download performance of a cached object.

As an alternative, we exploit the following observation. On one hand, modern caches use the entire URL strings, including the search string (the optional portion of a URL after "?") as the cache key. For example, a request for foo.jpg?randomstring will be forwarded to the origin server because the cache is unlikely to have a previously stored object with this URL. On the other hand, origin servers ignore unexpected search strings in otherwise valid URLs. Thus, the above request will return the valid foo.jpg image from the origin server.

Verification To verify this technique, we first check that we can download a valid object through the CDN even if we append a random search string to its URL, e.g., "ak.buy.com/db_assets/ large_images/093/207502093.jpg?random". We observed this to be the case with all three CDNs.

Next, we measure the throughput of downloading a cached object from a given edge server. To this end, we first issue a request to an edge server for a regular URL (without random strings) and then measure the download throughput of repeated requests to the same edge server for the same URL. Since the first request would place the object into the edge server's cache, the performance of subsequent downloads indicates the performance of cached delivery.

Finally, to verify that requests with random strings are indeed forwarded to the origin site, we compare the performance of the first download of a URL with a given random string (referred to as "initial download" below) with repeated downloads from the same edge server using the same random string (referred to as "repeat download") and with the cached download of the same object. The repeat download would presumably be satisfied from the edge server cache. Therefore, if changing the random string leads to distinctly worse download performance, while repeat downloads show similar throughout to the cached download, it would indicate that the initial requests with random strings are processed by the origin server.

Trial Number	1	2	3	4	5	6	7	8	9	10	Average
Limelight	775	1028	1063	1009	958	1025	941	1029	1019	337	918
Akamai	1295	1600	1579	1506	1584	1546	1558	1570	1539	1557	1533

Table 1. The throughput of a cached object download (KB/s). Object requests have no appended random string.

String Number	1	2	3	4	5	6	7	8	9	10	Average
Initial Download	130	156	155	155	156	155	156	147	151	156	152
Repeat Download	1540	1541	1565	1563	1582	1530	1522	1536	1574	1595	1555

Table 2. Initial vs. repeat download throughput for Akamai (KB/s). Requests include appended random strings.

We select one object cached by each CDN: a 47K image from Akamai[1] and a 57K image from Limelight[2]. (The open nature of Coral allows direct verification, which we describe later.) Using a client machine in our lab (129.22.150.231), we resolve the hostname from each URLs to obtain the IP address of the edge server selected by each CDN for our client. These edge servers, 192.5.110.40 for Akamai and 208.111.168.6 for Limelight, were used for all the downloads in this experiment.

Table 1 shows the throughput of ten repeated downloads of the selected object from each CDN, using its regular URL. These results indicate the cached download performance. Tables 2, and 3 present the throughput of initial and repeat downloads of the same objects with ten different random strings.

The results show a clear difference in download performance between initial and repeat downloads. The repeat download is over 10 times faster for the Akamai case and almost 7 times faster for Limelight. Furthermore, no download with a fresh random string, in any of the tests, approaches the performance of any repeat downloads. At the same time, the performance of the repeat download with random strings is very similar to the cached download. This confirms that a repeat download with a random string is served from the cache while appending a new random string defeats edge server caching in both Akamai and Limelight.

In the case of Coral CDN, we verify its handling of random search strings directly as follows. We setup our private Web server on host saiyud.case.edu (129.22.150.231) whose only content is an object http://saiyud.case.edu/pic01.jpg. Given the open nature of Coral CDN, a client can now download this object through Coral by accessing URL "http://saiyud .case.edu.nyud.net/pic01.jpg". Next, we obtain the edge server selected by Coral for our client by resolving the hostname saiyud.case.edu.nyud.net. Then, we use this server (155.246.12.164) explicitly for this experiment with the technique from Section 3.2.

[1] "ak.buy.com/db_assets/large_images/093/207502093.jpg"

[2] "modelmayhm-8.vo.llnwd.net/d1/photos/081120/17/4925ea2539593.jpg"

String Number	1	2	3	4	5	6	7	8	9	10	Average
Initial Download	141	111	20	192	196	125	166	128	18	140	124
Repeat Download	611	876	749	829	736	933	765	1063	847	817	828

Table 3. Initial vs. repeat download throughput for Limelight (KB/s). Requests include appended random strings.

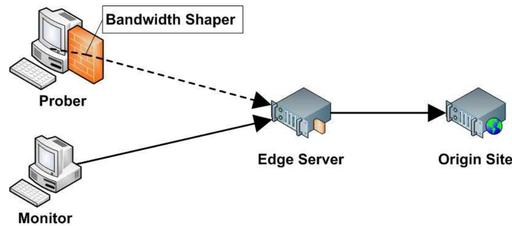


Fig. 2. Decoupled File Transfers Experiment

To check that Coral caches our object, we requested pic01.jpg from the above edge server three times without a random search string and verified that the log on our web server recorded only one access of pic01.jpg. This means the other downloads were served from the edge server cache. Then, we again issued three requests of pic01.jpg to this edge server, but now with a different random search string in each request. This time, our Web server log recorded three accesses of pic01.jpg from the edge server. We conclude that appending a random string causes Coral edge server to fetch the file from the origin regardless of the state of its cache, as was the case with Akamai and Limelight.

3.4 Amplifying the Attack: Decoupled File Transfers

We showed in Section 3.3 that one can manipulate a CDN edge server to download the file from the origin server regardless of the content of its cache and therefore penetrate CDN's protection of a Web site against a DoS attack. We now show that the attacker can actually recruit an edge server to consume bandwidth resources from the origin site without expending much of the attacker's own bandwidth.

In particular, we will show that edge servers download files from the origin and upload them to the client over decoupled TCP connections, so that the file transfer speeds over both connections are largely independent[3]. In fact, this is a natural implementation of an edge server, which could also be rationalized by the desire to have the file available in the cache for future requests as soon as possible. Unfortunately, as we will see, it also has serious security implications.

Verification To demonstrate the independence of the two file transfers, we setup two client computers, a prober and a monitor as shown in figure 2. The prober has the ability to shape its bandwidth or cut its network connection right after sending the HTTP request. The monitor runs the regular Linux network stack.

The prober requests a CDN-accelerated object from an edge server E with an appended random string to ensure that E obtain a fresh copy from the origin server. The prober shapes its bandwidth to be very low, or cuts the connection

[3] We do not claim these are completely independent: there could be some interplay at the edge server between the TCP receive buffer on the origin-facing connection and the TCP send buffer on the client-facing side. These details are immaterial to the current paper because they do not prevent the attack amplification we are describing.

String Number	1	2	3	4	5	6	7	8	9	10	Average
Limelight	1058	1027	721	797	950	759	943	949	935	928	907
Akamai	1564	1543	1560	1531	1562	1589	1591	1600	1583	1544	1567

Table 4. The download throughput (KB/s) of the monitor client. The monitor request is sent 0.5s after the probing request.

altogether after sending the HTTP request. While the prober is making a slow (if any) progress in downloading the file, the monitor sends a request for the same URL with the same random string to E and measures its download throughput. If the throughput is comparable to the repeat download throughput from Section 3.3, it means the edge server processed the monitor's request from its cache. Thus, the edge server must have completed the file transfer from the origin as the result of the prober's access even though the prober has hardly downloaded any content yet. On the other hand, if the throughput is comparable to that of the initial download from Section 3.3, then the edge server has not acquired the file and is serving it from the origin. This would indicate that the edge server matches in some way the speed of its file download from the origin to the speed of its file upload to the requester.

Because edge servers may react differently to different behavior of the clients, we have experimented with the prober (a) throttling its connection, (b) going silent (not sending any acknowledgements) after sending the HTTP request, and (c) cutting the connection altogether, with sending the reset TCP segment to the edge server in response to its first data segment. We found that none of three CDNs modify their file download behavior in response to any of the above measures. Thus, we present the results for the most aggressive bandwidth savings technique by the requester, which includes setting the input TCP buffer to only 256 bytes – so that the edge server will only send a small initial amount of data (this cuts the payload in the first data segment from 1460 bytes to at most 256 bytes), and cutting the TCP connection with a reset after transmitting the HTTP request (so that the edge server will not attempt to retransmit the first data segment after timeouts).

The experiments from the previous subsection showed that both Akamai and Limelight transferred their respective object from origin with the throughput of between 100 and 200KB/s (an occasional outlier in the case of Limelight notwithstanding). Given that either object is roughly 50K in size, we issue the monitoring request 0.5s after the probing request, so that if our hypothesis of the full-throttle download is correct, each edge server will have transferred the entire object into the edge server cache by the time of the monitoring request arrival.

The results are shown in Table 4. It shows that the download throughputs measured by the monitor matches closely those for repeat downloads from Section 3.3. Thus, the monitor obtained its object from the edge server cache. Because the edge server could process this request from its cache only due to the download caused by the request from the prober, and the latter downloaded only a negligible amount of content, we have shown that, with the help of the edge server, the prober can consume (object-size)/0.5s, or roughly 100KB/s, of the origin's bandwidth while expending very little bandwidth of its own.

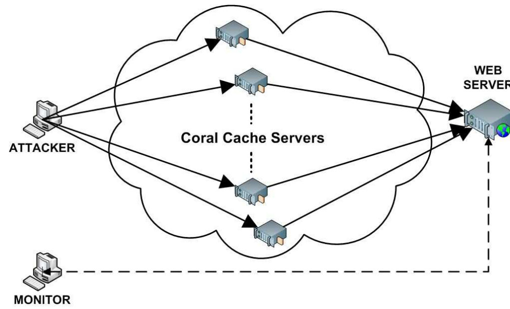


Fig. 3. DoS Attack With Coral CDN

4 End-to-End Attack

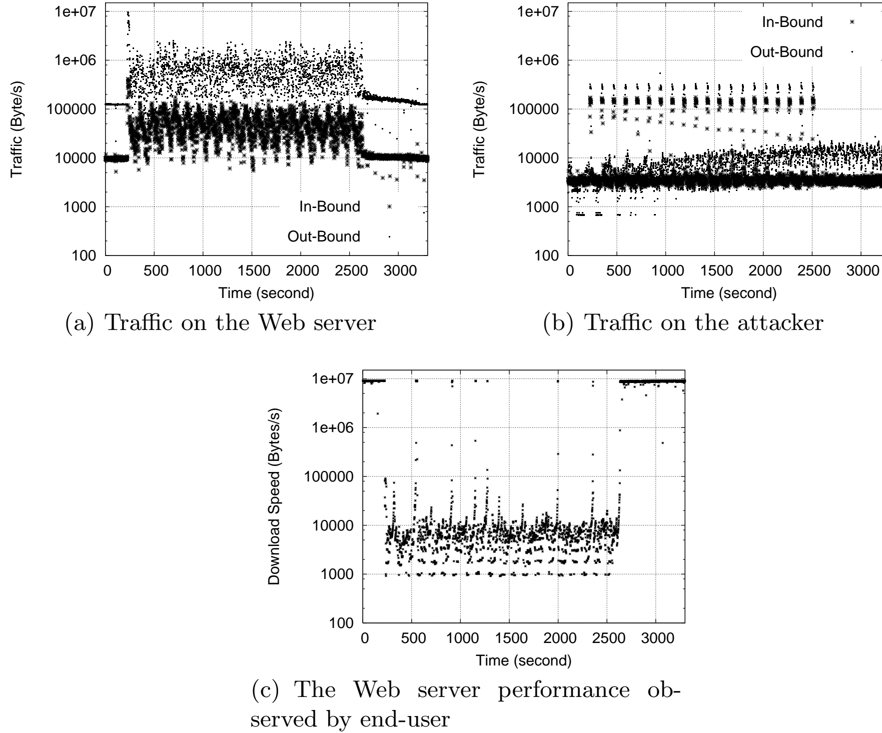
This section demonstrates the end-to-end attack that brings together the vulnerabilities described in the previous section. To do so, we setup our own web server as a victim and use the Coral CDN to launch the amplified DoS attack against this server. This way, we can show the effusiveness of our attack without affecting any existing Web site; further, due to elaborate per-node and per-site rate controls imposed by Coral [4] we do not affect the Coral platform either. In fact, our experiments generate only roughly 18Kbps of traffic on each Coral server during the sustained attack and under 1Mbps during the burst attack - hardly a strain for a well-provisioned node. Our results show that even our modest attempt resulted in over an order of magnitude attack amplification and two-three orders of magnitude service degradation of the web site.

We should note that after a number of attacks, Coral apparently was able to correlate our request pattern across their nodes and block our subnet from further attacks. This, however, happened only *after* a number of successful attacks. The mitigation methods we describe in Section 6 would allow one to prevent these attacks *before* they occur. Furthermore, a real attacker could use a botnet to change the attacking host at will and negate the benefit of even post-mortem detection. We discuss data-mining-based protection in more detail in Section 4.4.

4.1 The Setup

Figure 3 shows our experimental setup. The victim web server hosts a single 100K target object. The attacker host issues a number of requests for this object with different random strings to each of the Coral cache servers. To reduce its traffic load, the attacker sets an artificially small input TCP buffers of 256 bytes for its HTTP downloads and terminates its connections upon the arrival of the first data packet. The monitor acts as a regular client. It downloads the object directly from the victim web site once a second to measure the performance perceived by an end-user.

We use the identical machines for both the victim web server and the attacker: a dual core AMD Opteron 175 CPU with 2 GB memory and a gigabit link. The Web server is Apache 2.2.10 with the number of concurrent clients set to 1000 to increase parallelism. The monitor is a desktop with Intel P4 3.2GHz CPU, 1GB memory and a gigabit link. We use a set of 263 Coral cache servers to amplify the attack in our experiment.



(a) Traffic on the Web server

(b) Traffic on the attacker

(c) The Web server performance observed by end-user

Fig. 4. The effects of a sustained DoS attack.

	In-Bound (B/s)	Out-Bound (B/s)	Total (B/s)
Server	40,528	515,200	555,728
Attacker	13,907	31,759	45,666

Table 5. Average traffic increase during the attack period.

4.2 A Sustained Attack

To show the feasibility of sustaining an attack over a long period of time, we let the attacker send 25 requests to each of the 263 Coral cache servers every two minutes, repeating this cycle 20 times. Thus, this is an attempt to create a 40-minute long attack. The effects of this attack are shown in Figure 4.

Figures 4(a) and 4(b) depicts the in-bound and out-bound per-second traffic on the web server and the attacker before, during, and after the attack. Table 5 shows the average *increase* of traffic during the attack on the server and the attacker. As seen from this table, the attack increases overall traffic at the origin site by $555,728$ Byte/s (4.45 MBps), or almost almost half of the 10Base Ethernet link bandwidth. Moreover, this load is imposed at the cost of only $45,666$ Byte/s traffic increment to the attacker, or a quarter of a T1 link bandwidth. Thus, the attacker was able to use a CDN to amplify its attack by an order of magnitude over the entire duration of the attack.

Figure 4(c) shows the dynamics of the download performance (measured as throughput) as seen by the monitor, representing a regular user to our web site.

The figure indicates a dramatic degradation of user-perceived performance during the attack period. The download throughput of the monitor host dropped by 71.67 times on average over the entire 40-minute attack period, from 8824.2KB/s to 123.13KB/s.[4]

In summary, our attack utilized a CDN to fill half of the 10Base Ethernet link of its customer Web site at the cost of a quarter of T1 link bandwidth for 40 minutes. A more aggressive attack (using more edge servers and a larger target file) would result in an even larger amplification.

4.3 A Burst Attack

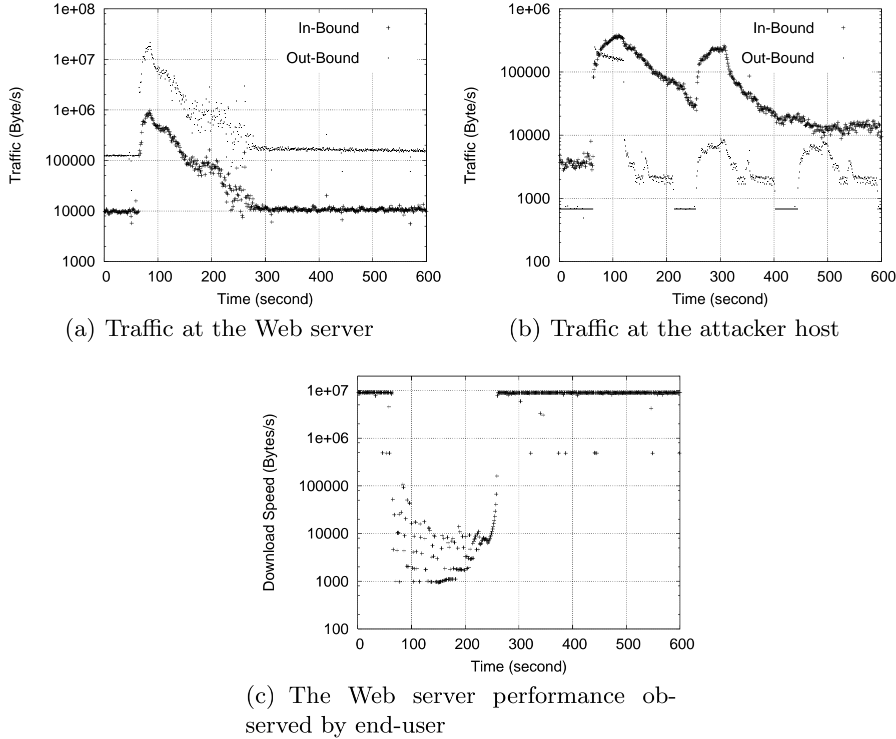
A CDN may attempt to employ data mining over the arriving requests to detect and block our attack. While we discuss in Section 4.4 why this would be challenging to do in a timely manner, we also wanted to see what damage the attacker could inflict with a single burst of requests to minimize a chance of detection. Consequently, in this experiment, the attacker sends a one-time burst of 100 requests to each of the 263 Coral servers. This apparently tripped Coral's rate limiting, and only around a third of the total requests made their way to the victim Web server. However, as we will see below, these requests were more than enough to cause damage.

The dynamics of this attack are shown in Figure 5. We should mention that this experiment was performed with the attacker host going completely silent instead of resetting the connection right after receiving the first data packet. With this setup, the Coral servers performed multiple retransmission attempts for the unacknowledged first data packet of the response. This lead to a slight increase of the attacker bandwidth consumption. However, even with this increase, the attacker achieves an effective attack amplification, by more than the factor of 50 at its peak.

As one can see from Figure 5, a single burst attack can have a long-lasting effect on the web site. Its bandwidth consumption increased by an order of magnitude or more for 85 seconds. The attack amplification of at least an order of magnitude lasted for almost two minutes (114 seconds). The average download performance seen by the monitor dropped three orders of magnitude, from the average of 8.6 MB/s during the normal period to 8.4 KB/s for over three minutes. These long-lasting effects are caused by the pending requests accumulated at the server, which take a long time to resolve and prolong the the attack.

We conclude that a burst attack can cause a significant temporary disruption of a Web site. By repeating burst attacks from random botnet nodes at random times, the attacker can lead to intermittent availability and erratic performance of its victim site.

[4] We should note that the absolute performance numbers regarding the web server performance should be taken with a grain of salt because they depend on server tuning. Tuning a web server, however, is not a focus of this paper, and our measurements reflect a typical configuration.



(a) Traffic at the Web server

(b) Traffic at the attacker host

(c) The Web server performance observed by end-user

Fig. 5. The effects of a burst DoS attack.

4.4 Discussion: Extrapolation to Commercial CDNs

We have shown above the end-to-end effect of our attack with Coral CDN. Since we can only assess the effect by observing a degraded performance, we could not perform a similar demonstration with commercial CDNs without launching a DoS attack against the affected content provider. We considered to try to degrade the performance of the content provider "just a bit", but realized that either this degradation would be in the noise, in which case our demonstration would be inconclusive, or the degradation would be noticeable, in which case it is a DoS attack unless the content provider consented to our experiment.

While we could not safely replicate our Coral attack with commercial CDNs, we conclusively showed that an attacker could make the origin site consume almost 1Mpbs of its bandwidth (i.e., transmit a file of roughly 50K in at most 0.5s – see Section 3.4), at the expense of negligible bandwidth of its own. Simply replicating this action, using different random strings and different edge servers, would allow the attacker to saturate the content provider bandwidth or other resources. In theory, one could imagine a CDN to use some clever data mining to detect and block the attacker that replicates these actions. However, such data mining would be challenging and at best only provide partial protection. Indeed:

– It cannot protect against a burst attack. Because the attack consumes very little resources on the attacking host, the attacker can send a large number

of requests to a large number of edge servers almost instantaneously. As we saw in Section 4.3, because of queuing of pending requests, a single burst can affect the content provider for a long time.
- A CDN cannot perform this data mining at individual edge servers or even data centers because each server will only see a very low request rate from the attacker. For example, to saturate a T3 line, the attacker must send only 45 requests per second (less if a larger than 50K object were used in the attack). Assuming a CDN with 500 locations, this translates into less than one request per ten second to each data center. Thus, the data mining by a CDN has to be centralized.
- Performing centralized data mining over global request rates requires transferring large amounts of data, in real time, to the central location. Although CDNs do provide global usage reports to their customers, detecting our attack requires data at the fine granularity of individual clients' requests to individual URLs. As an example, Akamai's EdgeSuite service provides usage reports only at 1-minute granularity and with aggregated information such as numbers of clients accessing various Akamai locations and their overall request rates to the subscriber's content. The timeliness with which they can "drill down" to individual clients and URLs is unclear.
- Even if real-time centralized data mining were possible, the attacker can further complicate the detection by using a botnet and/or employing multiple objects in the attack.

In summary, while data mining detection of a sustained attack is theoretically possible, we believe (a) a far better protection is to prevent amplified malicious requests and/or provide enough data to subscribers to let them perform their own site-specific detection (see Section 6), and (b) content delivery networks and their subscribers must be aware of this dangerous attack regardless, to make sure they are protected.

5 Implication for CDN Security

Although this paper focuses on the threat to CDN customers, the vulnerabilities we describe also pose security issues for the CDN itself. We demonstrated in Section 3.3 that edge servers view each URL with an appended random string as a unique URL, and cache it independently. Thus, by requesting an object with multiple random strings, the attacker can consume cache space multiple times. Furthermore, by overriding CDN's edge server selection (Section 3.2), the attacker can employ a botnet to both target strategically selected edge servers and to complicate the detection. Constructing its requests from several base URLs can further complicate the detection of this attack.

In principle, the attacker can attempt to pollute the CDN cache even without the random strings, simply by requesting a large number of distinct CDN-accelerated URLs. However, unlike forward caches, edge servers only accelerate a well-defined set of content which belongs to their customers, limiting the degree

of cache pollution that could be done with legitimate URLs. The random string vulnerability removes this limit.

Detailed evaluation of this attack is complicated and is outside the scope of this paper. We only note that the countermeasure described in Section 6.1 will protect against this threat as well.

6 Mitigation

The described attack involves several vulnerabilities, and different measures can target different vulnerabilities. In this section, we describe a range of measures that can be taken by content providers and by CDNs to protect or mitigate our attack. However, we view our most important contribution to be in identifying the attack. Even the simple heuristic of dropping URLs in which query strings follow file extensions that indicate static files, such as ".html", ".gif", ".pdf", would go a long way towards reducing the applicability of our attack. Indeed, these URLs should not require query strings.

6.1 Defense by Content Provider

Our attack crucially relies on the random string vulnerability, which allows the attacker to penetrate the protective shield of edge servers and reach the origin. Content providers can effectively protect themselves against this vulnerability by changing the setup of their CDN service as described below. We will also see that some types of CDN services are not amenable to this change; in these cases, the content provider cannot protect itself unilaterally and must either forgo these services or rely on CDN's mitigation described in the next subsection.

To protect against the random string vulnerability, a content provider can setup its CDN service so that only URLs without argument strings are accelerated by the CDN. Then, it can configure the origin server to always return an error to *any* request from an edge server that contains an argument string. Returning the static error message is done from main memory and consumes few resources from both the server and network. In fact, some CDNs customize how their URLs are processed by edge servers. In particular, Akamai allows a customer to specify URL patterns to be dropped or ignored [12]. The content provider could use this feature to configure edge servers to drop any requests with argument strings, thus eliminating our attack entirely. The only exception could be for query strings with a small fixed set of legitimate values which could be enumerated at edge servers. We refer to this approach of setting up a CDN service as "no-strings-attached".

The details how no-strings-attached could be implemented depend on the individual Web sites. To illustrate the general idea, consider a Web site, foo.com, that has some dynamic URLs that do require seemingly random parameters. A possible setup involves concentrating the objects whose delivery is outsourced to CDN in one sub-domain, say, outsourced.foo.com, and objects requiring argument strings in another, such as self.foo.com. Referring back to Figure 1,

foo.com's DNS server would return a CNAME record pointing to the CDN network only to queries for the former hostname and respond directly with the origin's IP address to queries for the latter hostname.

Note that the no-strings-attached approach stipulates a so-called "origin-first" CDN setup [14] and eliminates the option of the popular "CDN-first" setup. Thus, the no-strings-attached approach clearly limits the flexibility of the CDN setup but allows content providers to implement the definitive protection against our attack.

6.2 Mitigation by CDN

Although the no-strings-attached approach protects against our attack, it limits the flexibility of a CDN setup. Moreover, some CDN services are not amenable to the no-strings-attached approach. For example, Akamai offers content providers an *edge-side includes* (ESI) service, which assembles HTTP responses at the edge servers from dynamic and static fragments [6]. ESI reduces bandwidth consumption at the origin servers, which transmit to edge servers only the dynamic fragments rather than entire responses. However, requests for these objects usually do contain parameters, and thus no-strings-attached does not apply. In the absence of the no-strings-attached, a CDN can take the following steps to mitigate our attack.

To prevent the attacker from hiding behind a CDN, the edge server can pass the client's IP address to the origin server any time it forwards a request to the origin. This can be done by adding an optional HTTP header into the request. This information will facilitate the identification of, and refusal of service to, attacking hosts at the origin server. Of course, the attacker can still attempt to hide by coming through its own intermediaries, such as a botnet, or public Web proxies. However, our suggestion will remove the additional CDN-facilitated means of hiding. Coral CDN already provides this information in its x-codemux-client header. We believe every CDN must follow this practice.

Further, the CDN can prevent being used for an attack amplification by throttling its file transfer from the origin server depending on the progress of its own file transfer to the client. At the very least, the edge servers can adopt so-called *abort forwarding* [7], that is, stop its file download from the origin whenever the client closes its connection. This would prevent the most aggressive attack amplification we demonstrated in this paper, although still allow the attacker to achieve significant amplification by slowing down its transfer. More elaborate connection throttling is not such a clear-cut recommendation at this point. On one hand, it would minimize the attack amplification with respect to bandwidth consumption. On the other hand, it would tie other server resources (e.g., server memory, process or thread, etc.) for the duration of the download and delay the availability of the file to future requests. We leave a full investigation of connection throttling implications for future work.

7 Related Work

Most prior work considering security issues in CDNs focused on the vulnerabilities and protection of the CDN infrastructure itself and on the level of protection it affords to its customers [20, 17, 10, 9]. In particular, Wang et al consider the how to protect edge servers against break-ins [20] and Su and Kuzmanovich discover vulnerabilities in Akamai's streaming infrastructure [17]. Our attack targets not the CDN but its customer Web sites.

Lee et al. propose a mechanism to improve the resiliency of edge servers to SYN floods, which in particular prevents a client from sending requests to unintended edge servers [10]. Thus, it would in principle offer some mitigation against our attack (at least in terms of detection avoidance) because it would disallow the attacking host to connect to more than one edge server. Unfortunately, this mechanism requires the CDN to know the client IP address when it selects the edge server, the information that is not available in DNS-level redirection.

Jung et al. investigated the degree of CDN's protection of a Web site against a flash crowd and found that cache misses from a large number of edge servers at the onset of the flash event can overload the origin site [9]. Their solution – dynamic formation of caching hierarchies – will not help with our attack as our attack penetrates caching. Andersen [3] mentions a possibility of a DoS attack that includes the amplification aspect but otherwise is the same as flash crowds considered in [9] (since repeated requests do not penetrate CDN caches); thus the solution from [9] applies to this attack also. We experimentally confirm the amplification threat and make it immune to this solution by equipping it with the ability to penetrate CDN caches.

The amplification aspect of our attack takes advantage of the fact that HTTP responses are much larger than requests. The similar property in the DNS protocol has been exploited for DNS-based amplification attacks [19, 15].

Some of the measures we suggest as mitigation, namely, abort forwarding and connection throttling have been previously suggested in the context of improving benefits of forward Web proxies [7]. We show that these techniques can be useful for the edge servers as well.

8 Conclusion

This paper describes a denial of service attack against Web sites that utilize a content delivery network (CDN). We show that not only a CDN may not protect its subscribers from a DoS attack, but can actually be recruited to amplify the attack. We demonstrate this attack by using the Coral CDN to attack our own web site with an order of magnitude attack amplification. While we could not replicate this experiment on commercial CDNs without launching an actual attack, we showed that two leading commercial CDNs, Akamai and Limelight, both exhibit all the vulnerabilities required for this attack. In particular, we showed how an attacker can (a) send a request to an arbitrary edge server within the CDN platform, overriding CDN's server selection, (b) penetrate CDN

caching to reach the origin site with each request, and (c) use an edge server to consume full bandwidth required for processing a request from the origin site while expending hardly any bandwidth of its own. We describe practical steps that CDNs and their subscribers can employ to protect against our attack.

Content delivery networks play a critical role in the modern Web infrastructure. The number of CDN vendors is growing rapidly, with most of them being young firms. We hope that our work will be helpful to these CDNs and their subscribers in avoiding a serious security pitfall.

Acknowledgements: We thank Mark Allman for an interesting discussion of the ideas presented here. He in particular pointed out the cache pollution implication of our attack. This work was supported by the National Science Foundation under Grants CNS-0615190, CNS-0721890, and CNS-0551603.

References

1. Akamai Technologies. http://www.akamai.com/html/technology/index.html.
2. Akamai Technologies. http://www.akamai.com/html/perspectives/index.html.
3. D. G. Andersen. Mayday: Distributed Filtering for Internet Services. In *4th Usenix Symp. on Internet Technologies and Sys.*, Seattle, WA, March 2003.
4. The Coral content distribution network. http://www.coralcdn.org/.
5. Dipzoom: Deep internet performance zoom. http://dipzoom.case.edu.
6. ESI Language Specification 1.0. http://www.w3.org/TR/esi-lang, August 2001.
7. A. Feldmann, R. Cáceres, F. Douglis, G. Glass, and M. Rabinovich. Performance of web proxy caching in heterogeneous bandwidth environments. In *INFOCOM*, pages 107–116, 1999.
8. Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *NSDI*, pages 239–252, 2004.
9. J. Jung, B. Krishnamurthy, and M. Rabinovich. Flash crowds and denial of service attacks: characterization and implications for CDNs and web sites. In *WWW*, pages 293–304, 2002.
10. K.-W. Lee, S. Chari, A. Shaikh, S.Sahu, and P.-C. Cheng. Improving the resilience of content distribution networks to large scale distributed denial of service attacks. *Computer Networks*, 51(10):2753–2770, 2007.
11. Limelight networks. http://www.limelightnetworks.com/network.htm.
12. Bruce Maggs. Personal communication, 2008.
13. C. Partridge, T. Mendez, and W. Milliken. RFC 1546: Host anycasting service, November 1993.
14. M. Rabinovich and O. Spatscheck. *Web Caching and Replication.* Addison-Wesley, 2001.
15. F. Scalzo. Recent DNS reflector attacks. http://www.nanog.org/mtg-0606/pdf/frank-scalzo.pdf, 2006.
16. A.-J. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. Drafting behind akamai (travelocity-based detouring). In *SIGCOMM*, pages 435–446, 2006.
17. A.-J. Su and A. Kuzmanovic. Thinning Akamai. In *ACM IMC*, pages 29–42, 2008.
18. S. Triukose, Z. Wen, and M.Rabinovich. Content delivery networks: How big is big enough? (poster paper). In *ACM SIGMETRICS*, Seattle, WA, June 2009.
19. R. Vaughn and G. Evron. DNS amplification attacks. http://www.isotf.org/news/, 2006.
20. L. Wang, K. Park, R. Pang, V. S. Pai, and L. Peterson. Reliability and security in the CoDeeN content distribution network. In *USENIX*, pages 171–184, 2004.