An Overview on Detection and Prevention of Application Layer DDoS Attacks

Samuel Black
Computer Science Department
University of Nevada Las Vegas
Las Vegas, Nevada
samuel.black@unlv.edu

Yoohwan Kim
Computer Science Department
University of Nevada Las Vegas
Las Vegas, Nevada
yoohwan.kim@unlv.edu

2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC) | 978-1-6654-8303-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CCWC54503.2022.9720741

***Abstract*—Distributed Denial-of-Service (DDoS) attacks aim to cause downtime or a lack of responsiveness for web services. DDoS attacks targeting the application layer are amongst the hardest to catch as they generally appear legitimate at lower layers and attempt to take advantage of common application functionality or aspects of the HTTP protocol, rather than simply send large amounts of traffic like with volumetric flooding. Attacks can focus on functionality such as database operations, file retrieval, or just general backend code. In this paper, we examine common forms of application layer attacks, preventative and detection measures, and take a closer look specifically at HTTP Flooding attacks by the High Orbit Ion Cannon (HOIC) and "low and slow" attacks through slowloris.**

Keywords—DDoS, Attacks, Application Layer, Layer 7, HTTP Flooding, HOIC, Low and Slow attacks, Slowloris

I. Introduction

Application layer Distributed Denial-of-Service (DDoS) attacks aim to stop or slow services by using application layer protocols. Attackers can specifically send requests that are legitimate at other layers, but can cause issues once they get to the application layer. These kinds of attacks have the potential to be quite dangerous under certain circumstances, taking advantage of application functionality to wreak havoc rather than relying on sheer power of the attacking machine(s).

Recognizing application layer attacks is often more reactive than proactive. Knowing that a specific request will consume excessive resources is not always clear until that request starts executing. Attacking traffic patterns also require some requests to be received before it becomes apparent that the traffic is malicious. It is important to catch an attack promptly, preferably as the request is first received. Detection and defense in response to an attack is hard enough, and preventing attacks entirely is nearly impossible. The goal then is to stop attacks quickly and prevent any future attacks where possible.

Unlike with lower layers, application layer attacks are not as obvious based on the traffic itself; the attackers can send relatively small amounts of traffic that still generate serious side effects to the victim and do not necessarily have to spoof any of the header information of lower layers. Instead, attacks often rely on the contents of the payload of the request or on vulnerabilities in the protocol itself. The attacks have the potential to be especially effective since not only is monitoring attacking traffic more difficult, but it also means that the resource barrier for launching an attack is lower.

Attackers do not often work using a single machine, instead opting for a collection of infected machines, known as a botnet, to launch attacks and overwhelm victim servers. A botnet is generally formed from a single attacker or small pool of attackers that hijack other users' machines and force them to execute malicious code. Once an attacker has access to many machines, they can quickly increase the stress of an attack by launching simultaneous requests targeting a server, as seen in ***Fig. 1***. Botnets also increase the complexity of the attack, creating a variety of IPs from which attacking traffic originates and making it harder to track the actual attacker's identity. Botnet traffic may also increase the complexity of the distribution of attacking requests, making detection more difficult.

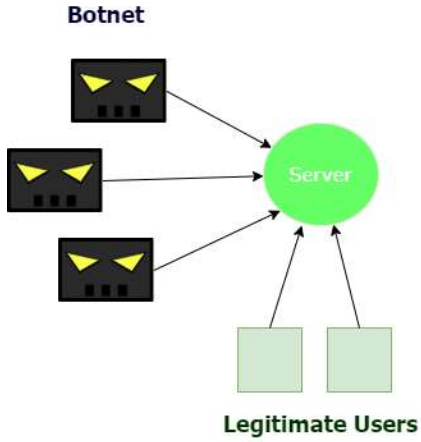

Fig. 1 A botnet is a collection of machines that autonomously act on a server. Sending HTTP requests alongside legitimate users can result in stress on the server that causes slowdown or down time

Effective attacks will maximize the ratio of victim to attacker resource usage [1]. The majority of DDoS attacks in general are "low-end"; that is, the bitrate at which packets are sent is considerably lower than the highest recorded attacks and the threshold rate for attacks that common mitigation services can handle. One such mitigation service, CloudFlare, reports that

978-1-6654-8303-2/22/$31.00 ©2022 IEEE

almost 89% of attacks that they experienced were under 50Kb/s and lasted less than an hour [2]. Some of the largest reported attacks CloudFlare claims to have handled have had bitrates exceeding 1Tb/s. These large-scale attacks, however, primarily target the network layer as opposed to the application layer. This could be due to the fact that attacks like TCP flooding are simpler to launch and require less knowledge of the victim's infrastructure [3] [4]. That being said, the absolute number of DDoS attacks has been trending upwards recently, and even more so as the average user traffic has increased, with more users staying home from the ongoing COVID pandemic. An increase in users not only means more potential attackers, but also more stress on the servers from legitimate users that can make the jobs of attackers easier. As application layer attacks are harder to detect and as the attacks get more sophisticated in taking advantage of vulnerabilities in applications themselves, the bitrate required for a successful attack becomes lower.

Common examples of application layer attacks include HTTP flooding and "low and slow" attacks. In this paper, we will examine these two types of attacks in more detail and preventative measures for general application layer attacks. We will also examine the effects they cause in a controlled environment using DDoS attack tools. Attacks will be simulated using High Orbit Ion Cannon (HOIC), which can send rapid amounts of HTTP requests, as well as a slowloris script for "low and slow" attacks. They will be tested using a low-end machine, which should allow the side effects of the attacks to be noticeable given the tools available. Note that while this paper may discuss the effects that these attacks could have on larger servers, the resources available are limited and would require further testing on a larger scale. More detailed information about system specifications and the attacks will be available in the ***Attack Simulations*** section (Section IV). Botnets are practical for application layer attacks, however due to limitations, a single attacking machine will be used against a single target in the following simulations. This will mean that effects seen will be milder than in realistic attacks.

II. Types of Application Layer Attacks

A. Overview

HTTP flooding is a strategy in which the attacker sends legitimate HTTP requests that target the functionality of the application. This method is notable in that it can target vulnerabilities in the actual backend code written by developers rather than just network protocols.

This is done by sending requests for tasks that consume a high number of resources on the server's end, generally more than a normal user would need [5]. Botnets attempt to create as many computationally expensive requests as possible on the server, consuming as much CPU time and bandwidth as possible. Successful attacks can cause the server to go down or at the very least increase the response time for handling requests. Attacking botnets try to tie up as many threads as possible, with each bot creating threads either frequently or that take a long time to finish, making the server unable to handle as many legitimate users, if any.

There are application layer attacks that require less resources to launch compared to attacks at other layers in that requests can be sent at relatively reasonable rates, but handling the requests takes a long time. More and more queued requests will build up until the attacker requests are done being processed assuming those requests take longer to handle than to receive new incoming ones. This creates a feedback loop; the worse the effects get on the server, the more they compound as attacking requests build up or requests are outright rejected, including legitimate ones.

It should also be considered that for each attacking machine, a TCP/IP connection must be initiated and maintained, which will also require some computation time for the handshake. Only a finite number of connections may be maintained. In particular, Apache allows a maximum of 150 concurrent connections by default, although this can be changed and is expected to be much larger for servers hosting many users.

HTTP flooding attacks can be more difficult to detect than attacks that target other layers. Unlike a volumetric attack like TCP flooding, which aims to simply send great amounts of traffic to overwhelm the server with no extra computation, HTTP flooding attempts to cause slowdown using the application functionality. HTTP flooding attacks are then more difficult to detect due to their possible complexity as well as lower bandwidth necessity for the attacker. In order to reach the application layer in the first place, the attacking traffic must seem legitimate at all other layers. The sending rate is not the chief contributor of negative effects, but rather the payload of the request or the operations that the request results in. Rather than try to fool a firewall or IDS, the requests often do very little to spoof information that would be relevant lower than layer 7. In the simulations later, measures will be taken to avoid Apache's Web IDS, which filters out some traffic/connections from being handled. This will depend on the tool being used, but generally user-agent headers (e.g. web browser used) and similar information will be spoofed.

There are several types of HTTP request methods, POST and GET being two of the most common. A POST request is one in which any parameters being sent by the requesting client are housed within the body of the request. POST requests are generally used when accessing scripts on a server that perform operations and return data. GET requests on the other hand keep any parameters being passed as part of the URL string itself, and are thus more easily monitored by web traffic sniffers. GET requests are ideal for data retrieval, where sensitive parameters are not often sent, such as getting a file from a server like a webpage or a piece of media.

B. POST Flooding

POST flooding is a specific type of HTTP flooding in which an attacker sends a POST request generally meant to tie up the server with computationally expensive operations. This can vary wildly depending on the target application, but common attacks include sending large payloads and requests that require expensive database operations. These attacks don't necessarily have to be limited to POST requests, but POST is the common method for these kinds of operations.

A straightforward approach involves sending a large payload that takes a long time to process. It is quite common

for applications to send serialized data (e.g., objects and arrays in XML or JSON) to a server for it to then parse and process, and return more serialized data. A large payload will take a relatively long time to receive, parse and deserialize, and process, especially if it contains some sort of array structure. Consider it similar to providing a large input file to a program locally. In the event that a botnet sends multiple instances of these kinds of attacks, threads on the server will need to consume more resources, both computation time and memory, which can cause slowdown for other requests being handled.

POST flooding can have limitations, however. Firstly, the attacker has to have some prior knowledge of the workings of the victim server. They would have to know which scripts take what format of data, both in terms of serialization scheme and in terms of the kind of data that it takes as input, such as the parameter types and names. On top of this, since the payload is large, the attacker will need to use more resources sending requests relative to other types of HTTP flooding attacks. This method is also easier to detect then some others; a large payload could be suspicious, but if a user is continuously sending large payloads, then they can be flagged and temporarily blacklisted. The same would go if there is a sudden influx from multiple senders of large payloads. A simple detection method would be to set a threshold on the size of the payload that can be received.

Aside from only sending large payloads, attackers may take advantage of operations that are inherently expensive. Some database queries can take a long time to execute despite being simple to request. Just to name a few: join operations, inefficient array insertions, and selections with many matching rows are expensive, yet generating queries that use them are not [6]. Again, some knowledge of the victim's infrastructure is necessary to exploit these, but once an expensive operation is identified, the attacker gains a way to generate requests that are disproportionately more stressful on the victim's side than on their machine(s).

Large payloads can generate many expensive queries; the attacker is not limited just to sending huge payloads to see results. Taking advantage of database operations can not only reduce the size of requests required for a successful attack to be launched, but also removes the large payload method of detection. Without large payloads, it is not as obvious when an illegitimate request is made, and can be harder to detect without first executing the query. While many relational database languages can optimize inefficient queries, there are some cases in which attackers can exploit vulnerabilities.

SQL injection is often associated with security breaches, but can also be used for attacks in this context. SQL injection is when an attacker inserts a string into a form submission on the frontend of an app or via the POST parameters in such a way that the string forms a well-formed SQL query on the server side with a different intention than the original query. This can be used to generate queries that receive data from unintended tables or include extra columns, but also can be used to increase the complexity of a query. It is possible to inject expensive, meaningless queries with the intention of forcing the server to spend more computation time. This can be effective in that the attacker does not have to download or send a large amount of data (just a small string) that yields computationally expensive requests only on the server side.

Depending on what measures are in place to stop SQL injection, the severity of these attacks can differ. In the event that there is no checking of strings being sent for queries, a query of arbitrary length and operations can be performed.

As an example, consider an application that needs to do a simple lookup of a user from their first name. An example of a possible SQL query may look like

```
"SELECT * from users WHERE fname = '<name_param>'"
```

In this example, the ***<name_param>*** may be a field that is entered by a user via some frontend application, like a text entry box, which is then concatenated to the rest of the string in a backend script. An attacker is then limited to providing a string that replaces ***<name_param>*** and still forms a syntactically valid query. One example of an attack string that could be entered for the ***<name_param>*** in this case might be

```
"x' OR CONVERT(varchar(10), (SELECT COUNT(column) FROM (SELECT column FROM users JOIN other_table ON id))) = 3 OR fname ='hello"
```

In taking this string and substituting it in for the ***<name_param>,*** a syntactically valid SQL query is generated that performs an unnecessary join operation on two tables and does evaluations of useless expressions.

There is some expectation that measures will be in place to limit SQL injections. On the front end for text fields, there are often simple ways to limit injection, such as character limits. Web application firewalls, services that allow firewall rules for incoming requests, can also be used to limit the way SQL injection attacks can be launched; strings can be limited to certain characters or fitting regular expressions. However, it is still possible for a clever attacker to find ways to generate valid queries such as this that serve to waste the server's execution time [7].

C. GET Flooding

GET requests are another type of HTTP request that can be used for attacks. While POST is generally used to pass parameters to server-side scripts, GET requests are often used for retrieving data.

File hosting servers are the prime target for GET requests. An attacker can send many requests to download large files, such as images, and tie up threads on the server as they send the data. A drawback to this attack strategy is that the attacker will need attacking machines to maintain a connection for the duration of the attack, thus limiting the number of requests that can be sent. The connection will also need to be maintained during the attack so as to download the whole file from the victim. The length of the attack can also be extended by purposefully downloading the data at a slow rate. While the initial request is small and not intensive for the attacker, the actual downloading portion will mean that the client and server

must both communicate any large files. This is yet another reason as to why the attacker will realistically require a botnet to launch a successful attack; a single or few machines would not be able to handle an attack that a server could not.

Many file hosting sites have measures in place to prevent or discourage large influxes of requests. For example, the popular file hosting site, Mega, limits users to 5GB of downloads within a 24-hour period unless they pay for a premium account [8]. Other popular media sites have similar measures, limiting how frequently requests can be sent and the amount of data that can be retrieved within a set amount of time. This again burdens the attacker, requiring them to have a sizable botnet to launch a meaningful attack, as well as concentrate all attacks within a time window before being locked out.

D. "Low and Slow" Attacks

"Low and Slow" attacks are a specific type of application layer attack in which a successful connection is made to a victim server, which is then kept open for as long as possible, with each connection requiring a thread on the server to be handled. With enough attackers, the server may have too many threads doing essentially nothing, causing serious side effects. Threads created to handle requests of legitimate users become starved for computation time as they make up a smaller and smaller proportion of all threads being run on the server.

These types of attacks are notable in that they do not require sending massive amounts of requests nor any real data, and instead rely on sending relatively small amounts of traffic that keep connections open. Not much data is required to stall and keep a connection open, thus low and slow attacks consume less bandwidth than other types of attacks, meaning there is a lower resource barrier.

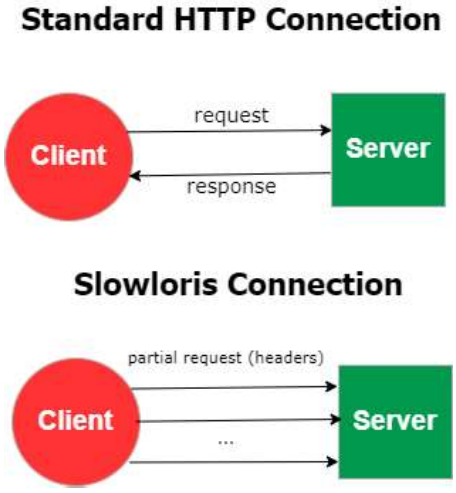

Fig. 2 Low and Slow attacks keep connections open without completing the request, thus keeping server threads busy waiting for nothing and never sending the final response

In the case of slowloris, connections are opened by sending legitimate HTTP requests followed with periodically sending HTTP headers to keep the connection open. The server is never able to give a response to the initial request and stalls indefinitely (***Fig. 2***). The script used for the sake of this paper allowed the number of web sockets maintaining connections with the server to be set alongside the number of seconds to sleep in between sending headers on each connected socket. In the event that the server closed a connection with a socket, a new socket is made in place of each closed one and is added to the pool of open sockets. The number of total sockets will affect the maximum number of threads created on the victim server, which will be maintained as long as possible. The seconds of sleep time should affect the likelihood that a socket will be kept open by the server, as if it does not get a message within a certain amount of time, or suspects an attack, the socket will be closed and will need to be reopened. Shrinking the time should increase the likelihood the thread is kept open, but should also decrease the amount of time that a closed socket is left without being replaced. The Python script used can be found here https://github.com/gkbrk/slowloris [9].

III. General Defense Measures

A. Basic Prevention

Many of the defense measures discussed so far require that an attack is already in progress before we are able to handle it. On top of this, users with poor or inconsistent connections are likely to be mistaken for attackers. It is preferrable then that we turn our focus to preventing the attacks from starting in the first place.

There are some simple-to-implement methods that can act as a first line of defense where possible. Having log-ins for authenticated users is a straightforward feature that can mitigate bot attacks. Assuming that making an account requires verification and that users cannot be logged in from more than a few IP addresses at a time, this adds a layer of tedium on the attacker's part. Attackers are forced to create multiple accounts, which can be difficult to do in bulk, given that account creation has anti-AI measures. While it may not be a method to completely block out attacks, it can act as a deterrent.

For certain web apps, metadata about the connection can also be used to prevent connections. For example, HTTP headers contain information about the user-agent of the sender. If this field is a common browser (Chrome, Firefox, Safari, Opera, etc.), then the request can be accepted. This does limit the users that can access a web app and might not be desired in some cases, particularly if some bot users are allowed, but can still stop some attacks. The user-agent related header fields can be spoofed; it is then even more crucial that we lean into different preventative measures.

In the case of POST attacks targeting computationally intensive tasks, a straightforward deterrent would be to simply limit the time that a thread is allowed to execute on the server, or to monitor the number of requests made in an average single user session [10]. This is subject to affecting normal users, though, as traffic conditions or general scheduling problems may lead to false positives. A good request scheduler does have the potential to mitigate issues from attacks. A simple blacklist can suffice, as well.

Low and slow attacks can be handled directly by simply setting a maximum amount of time since the initial request to complete before killing it. Using mitigation services such as CloudFlare as a middleman between clients and a server can also stop these attacks entirely, as the service can pass only requests

that have been completely received to the final destination server [11]. The service itself should still try to detect these attacks for their own sake.

B. Prevention with and against Machine Learning

In a realistic setting, an attacker will not be able to produce noticeable effects by themselves; they will need to have a botnet in order to launch a successful attack.

Adding the step of automation to the attack makes the attacker's job easier, but it also provides a shortcoming in that we can use present weaknesses of AI. Certain defense mechanisms such as blacklisting IPs may prove ineffective when an attacker has access to a wide array of attacking machines. However, posing difficult challenges to bots can also act to stop harmful connections. Computer vision and human behavior tasks are frequently used to differentiate human users from bots [12].

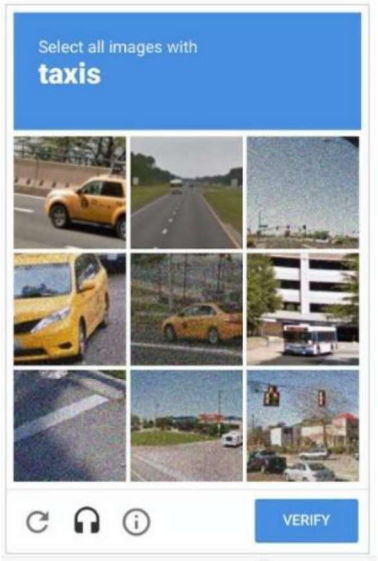

Fig. 3 Modern CAPTCHA Challenge. Notice that several of the images have noise applied. Image retrieved from article [13]

The most notable preventative measure would have to be CAPTCHA. CAPTCHA refers to a broad set of challenges that attempt to differentiate a bot from a human. They became common in the early 2000s by issuing challenges that were relatively easy for humans to solve, but hard for computers to solve. The most common example of this at that time would have been the text CAPTCHA with distortions applied to the text. CAPTCHA challenges have since evolved to keep pace with innovations in computer vision. Current iterations that use image recognition tests will often display a set of images with a label and ask the user to identify all images that match that label. Similar to the old text challenges, distortions are applied to images in contemporary CAPTCHA challenges in the form of noise or splitting key parts of the image on boundary lines, such as in ***Fig. 3***.

Modern computer vision models are highly susceptible to noise; however, it is possible that in the future this weakness may be overcome. There are new models that aim to reduce the amount of noise in images before making a classification [14]. CAPTCHA challenges will need to continue to evolve if bots are to be kept out. Either through further weaknesses in computer vision, or through other tasks that are currently challenging for computers, like natural language.

C. Mitigation Services

In a business setting, DDoS attacks can cause major problems; server downtime causes loss not only in revenue, but also in reputation. Attacks can have many complex parts and require a software suite with a wide range of mitigation techniques for different scenarios. Mitigation services aim to offer these software suites for businesses as well as other quality of life features. While the loss as a result of downtime is dependent on a number of factors such as online presence and size, it is estimated by *Imperva* that an hour of downtime can cost anywhere from $50-300K for a small business [15]. The estimate only considers the amount needed to be provided to engineers that would fix the problem, but online storefronts could also see a loss of sales during this time, so this estimate is potentially low. It is also important to consider that over 50% of attacks last less than 15 minutes, so the average cost per attack could be lower than the hourly wage of engineers, depending on the exact attack length [16].

Several commercial options exist for DDoS prevention and attack detection. Imperva and Cloudflare are two such services that offer software suites to detect, prevent, and notify users about attacks. While the inner workings of these sites are private, there are some general descriptions of services provided on their websites. The services screen incoming requests and classify them as malicious using only the traffic itself. Imperva avoids using extra authentication such as CAPTCHA to stop bots from accessing sites for the sake of user convenience. *CloudFlare* opts in to this however, posing challenges to potentially harmful bots in exchange for mild inconvenience to users. These services act as a middle man between users and the actual host server (***Fig. 4***), meaning that the IP associated with a site's domain will belong to one of these services rather than the actual site itself, and that none of the computation done to detect attacks has to be performed by the actual destination server itself. With mitigation tools, massive servers can withstand upwards of 90 Gb/s of incoming requests according to a claim by *Imperva* [16]. CloudFlare boasts prevention of some of the largest measured attacks of up to 1Tb/s, although all of the details about such attacks are not made public.

Fig. 4 Diagram of Imperva service in web app protection. Their service acts filters requests before they arrive at the destination server [17].

The services themselves have price ranges that differ on the size of the site protection is being offered for, but *CloudFlare* offers rates of about $200 per month for small businesses and *Imperva* roughly $300 per month.

D. Detection

Instead of keeping users out, it is also possible to monitor incoming requests and handle suspicious behavior. Some developers may opt to avoid user tests such as CAPTCHA for the sake of user experience, and may choose to react to incoming malicious traffic, instead. Detecting attacks as opposed to adding preventative measures can improve end-user quality, although it can also run the risk of letting attackers get their foot in the door. Application layer attacks generally try to appear legitimate and can be difficult to distinguish from legitimate users. A straightforward and common detection method involves simply monitoring how many incoming requests are being sent from a single user and limiting the number that can be sent over a specified time period. This prevents both attackers and users who make frequent requests. Casual users to sites (using browsers) are likely not to be affected by this, but developers accessing a server with an API may experience lock outs. It is important then when using this method that the threshold for classifying requests as attacks is picked based on reasonable use traffic.

Machine learning is becoming increasingly popular for finding underlining factors in different mediums, and network security is no exception [18]. It is possible to train machine learning models on traffic generated by attacks as well as normal traffic so that it can discriminate between the two. ML models can take into consideration features such as distribution of requests over a given time period, header information, and even underlying factors that are not human comprehendible.

There has been success with machine learning models to predict whether a given request is malicious. Research done by Pokhrel et al. and a second study by Umarani and Sharmila showed good results in classifying static data using K-Nearest Neighbor and Naïve Bayes Classifiers using the traffic records and corresponding header fields as input over time periods [19] [20]. Another study by Sales de Lima Filho et al took a look at accuracy in several models and showed that several models, Random Forests and Decision Tree classifiers in particular, performed well in online environments where traffic was constantly fed in and classified [4]. Similar on the accuracy of Random Forests are also corroborated by Kousar et al and Nandi et al [21] [22]. It should be noted that the studies used flooding attack datasets, and while the research may provide helpful in classifying attacks that focus on rapid sending, they do not take into consideration the payload of the requests, which is where the danger of some application layer attacks comes from. These studies showed relatively low accuracy for MLPs/ANNs, despite their recent popularity. Further research may be needed into these models to determine if they still perform comparatively poorly with different architectures. In particular, neural networks with a generative component are good for learning distributions, which would seem to apply to traffic over time intervals.

More recently, the DDosNet by Elsayed et al has shown promise using neural networks, specifically a Recurrent Neural Network with an autoencoder prior to the network [23]. The DDosNet performed simple binary classification on whether or not a given piece of traffic belonged to an attack or not. The intuition behind using an RNN vs a plain ANN is that network traffic is best represented as a time series. That is, each individual packet received by the server has a temporal order; the amount of time between packets has a certain time it arrives at and the order of arrival matters, as well as the amount of time between packets. RNNs excel at time series, as they are capable of taking individual time step data from a given time series and feeding them into the network within the same feedforward pass. Context from a previous timestep is used when evaluating the next timestep in the data. In the DDoSNet, each timestep being a single piece of traffic received by the server. This research is also notable in that it specifically addresses application layer attacks, whereas many existing papers focus only on detection of volumetric flooding, where the focus of the attack is to simply send a lot of traffic in a short period of time. The DDoSNet in that paper showed up to 98.8% accuracy on the CICDDoS2019 data set.

IV. Attack Simulations

A. Preliminary Information

A series of attack simulations were run using the High Orbit Ion Cannon program. ***Fig. 5*** shows its user interface. Originally designed by the well-known hacker group, *Anonymous*, the program is intended to perform HTTP flooding attacks. In this experiment, simple HTTP requests were sent to retrieve the default Apache welcome html file for 10 minutes and server response was noted.

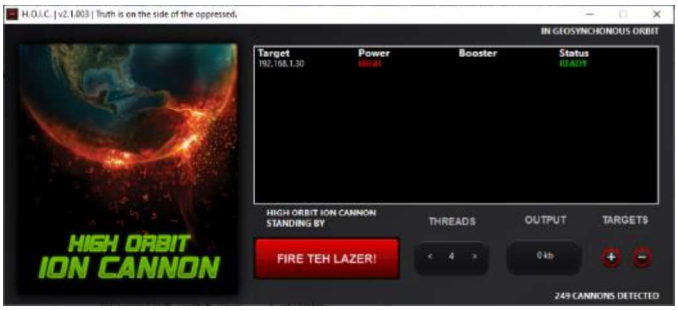

Fig. 5 HOIC Program GUI

Several simulations were run having the HOIC program send requests to a Raspberry Pi Model 3 B+ (Revision 1.3) hosting an Apache2 server. The communication path can be seen in ***Fig. 6***. The Pi has an ARMv7 Processor with 4 single threaded cores; it has approximately 900 MB of primary storage. Requests were sent to the Pi while running no other applications outside of standard daemons to measure the level of stress it could take before experiencing downtime. A Raspberry Pi was picked both out of availability and because it is expected to have far less stress tolerance than a traditional server and thus more easily observed. Attacks were launched from a single Windows machine.

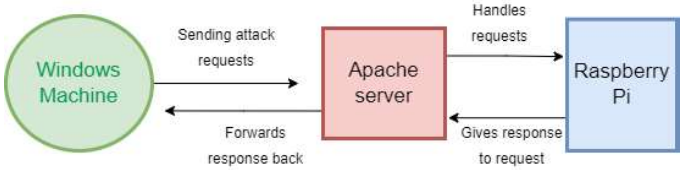

Fig. 6 Basic overview of server set up

The program allows some level of customization in the way that attacks are launched; other stats were measured to the best of my ability. The application lacks a logging feature and some of the parameters are non-specific. The parameters that can be set within HOIC include the host address to target, the sending "power" (low, medium, and high), and the number of threads to launch attacks. The power corresponds to the number of requests launched every second, with low being two requests per second, medium being 4 per second, and high being eight per second. The program also allows the option to import custom scripts that modify the header fields of the attacks to prevent detection and set parameters to be sent for POST requests. Initially, runs were done with differing power levels and with and without booster scripts. The runs not at high power and not using the generic script has negligible sending rates, and so their values are excluded.

Initially, tests were run in an attempt to cause server downtime and determine at what rate of transmission the different attacks would cause the server to go down. While attacks were in progress, the server was periodically pinged to see if it was still running and measure the response time, requests were sent infrequently via a browser, and the status of the host was checked within the HOIC window. Effects were recorded and can be observed in the provided tables.

B. HOIC Results

Please note that some data has been omitted for the sake of brevity; specifically runs with medium and low power as well as runs with no script applied. The sending rate was comparatively very slow (< 1Mb/s) and no effects to the server were observed for these runs.

Also note that due to the previously mentioned lack of a logging feature, the sending rate was estimated and may be slightly inaccurate. HOIC only specifies high, medium, and low power, but does not provide the target rate at which data is sent on threads for these powers. The sending rate was estimated by observing the amount of data sent over a 10 seconds interval, and then finding the average sending rate by taking the amount of data that had been sent over this time span by dividing by 10 to find the rate. There is also a brief period in which the sending rate is low. The measured rates were taken after waiting 30 seconds for the program to reach a stabilized state. The result is shown in ***Table 1***. It indicates that the server was not affected by the attack. In case of 2nd case (with 4 threads and a 1 Mb/s rate), the server crashed, but it is not clear whether the attack caused it.

TABLE I. RESULTS FROM GENERIC SCRIPT ATTACK ON INDEX.HTML HOME PAGE

Power	Threads	Avg. Rate (Mb/s)	Down?
High	2	0.3	None
High	4	1.0	Once
High	6	1.2	None
High	8	1.7	None
High	10	2.0	None
High	16	5.0	None
High	20	1.0	None

TABLE II. RESULTS FROM GENERIC SCRIPT ATTACK ON PHP BACKEND SCRIPT WITH DB OPERATIONS

Threads	Avg. Rate (Mb/s)	Down?	Ping (ms) avg	Ping (ms) stdev
Baseline	-	-	9.2	1.48
4	1.0	None	9.2	3.19
6	1.2	None	6.8	3.56
8	1.7	None	15.0	23.49
10	2.0	None	10.0	2.92
16	5.0	None	12.0	3.67

Given no apparent or consistent effects, the experiment was run a second time with extra server-side computation in hopes to see more noticeable effects. Instead of making a request to the default Apache index.html, requests were sent to a simple PHP script that made database queries. A sample database with a single table containing 30,000 rows was generated in MariaDB. Each row contained a random number on the interval [0, 1024] and a random string. On each request to the script a simple query was made that retrieved all rows whose random number column was below a certain threshold, arbitrarily picked to be 350. The query returned roughly 7000 rows.

In an attempt to better see effects, the attacks were always run with high power and the generic script applied. Pings were made to the server periodically while HOIC was running to gauge the response time. Included with these attacks is a control run where no attacks were underway, but the ping was still measured. The result is shown in ***Table 2***.

Finally, one more set of attacks was run, this time performing a retrieval of a file from the server with GET requests. A 3MB image was hosted on the server and retrieved during the attacks. Again, the ping was measured and the effects were recorded, as shown in ***Table 3***.

TABLE III. RESULTS FROM GENERIC SCRIPT IMAGE RETRIEVAL

Threads	Avg. Rate (Mb/s)	Down?	Ping (ms) avg	Ping (ms) stdev
Baseline	-	-	21.8	17.3
4	0.7	HOIC	324.6	4.34
6	1.7	HOIC	415.2	169.67
8	1.4	HOIC	363.0	86.26
10	0.8	HOIC	398.4	220.55

In the image retrieval runs, the server did not experience downtime, however HOIC did crash partway through for several runs. It crashed quicker as the number of threads were increased, which is why this table stops at 10 threads as opposed to the others.

C. Slowloris Results

A slowloris Python script was run and the effects were observed while the attack was underway. Several variations of attack were run with the number of sockets and the sleep time in between sending headers. The default suggested values of 150 sockets and 15 seconds between sending headers were used, and then further expanded upon. A flag was also set to

randomize user-agent header fields to make detection by the server more difficult. The sleep time field won't affect how many threads are created on the server; however, it will affect the likelihood that a connection will not be closed by the server in between sending headers. The server is more likely to close a connection if it has not heard from the client after a longer period of time. Connections that remain open consistently should affect the server more.

TABLE IV. RESULTS FROM SLOWLORIS SCRIPT ATTACKS ON INDEX.HTML PAGE

Sockets	Sleep Time (s)	Response Time (s)
Baseline	-	0.048
150	15	10.36
200	15	10.66
300	15	23.64
150	10	14.79
200	10	5.37
300	10	22.65
150	5	15.52
200	5	15.22
300	5	31.0

Attacks were sent to the same Raspberry Pi hosting an Apache server as described previously. HTTP requests were sent to retrieve the default index.html page in a Chrome browser and the response time was recorded via Chrome's network logging feature as seen in ***Table 4***. A control run was done with no attack underway to record the baseline response time. Pings were also sent to the server while attacks were underway, however no variation was noticed in the recorded ping between different attacks and when no attack was underway. In between each request to the server, the browser's cache was cleared to make sure that it would fully retrieve the document each time.

V. DISCUSSION

A. HOIC attacks on simple webpage/ query script

In the original tests where a simple html file was accessed seen in ***Table 1***, only one instance of downtime occurred. However, it was inconsistent and unable to be repeated. The one attack with effects was the run with 4 threads and high power. On a single run, the HOIC program reported the server as down and a ping reported 100% lost packets. A subsequent ping while the attack was still running showed that the server was back up. The server did not suffer any other downtime for the duration of that attack, and was repeated in an attempt to replicate the loss of service, however, the server did not have any noticeable side effects. The downtime was likely a fluke, given the other runs with higher sending rates did not experience any noticeable effects, let alone down time.

In hindsight, it should not be surprising that simply flooding the server with requests did not cause slowdown or downtime, as this is simply the goal of TCP flooding, and application layer attacks will inherently have a lesser or equal incoming rate than attacks than simply spam packets. With further protection against incoming attacks on top of the default Apache settings, the number of attack requests that actually make it to the server to be handled would be even further reduced.

One issue that can be observed is that the sending rate decreased after a certain number of threads was introduced. The peak sending rate was around 14-16 threads, with 20 threads yielding a lower sending rate. This can potentially be because of the hardware available that the attacks were launched with. This issue was present in later attack sets, as well.

While the original goal of these experiments was to find how much stress would be required to cause server downtime, which was no achieved, we can still make some observations from the data. The highest observed sending rate for both POST flooding type attacks was 5.0 Mb/s, which seems to be a reasonable peak sending speed for a single attacking target. Bots in a botnet are likely to be around this range depending on the machine's bandwidth and general computational power; the number of attackers needed to cause damage with HOIC POST attacks can be estimated from a server's estimated threshold it can withstand. Despite the hardware the server was hosted on being considerably low end, it did not experience any nonnegligible crashes or loss of service. This makes it further clear that botnets or large numbers of simultaneous users are necessary to launch a successful attack. If even a Raspberry Pi can handle these requests coming from a much more powerful machine, then a dedicated server will certainly require an even larger attack, even if it may have more background processes.

This data also helps to demonstrate the importance of having HTTP flooding detection. While no additional security software was added, Apache2 does offer some built in protection in recent versions. The data that was excluded for having a low sending rate was done so without using the provided generic HOIC script, which spoofed user-agent header fields. The data was excluded since the sending rate never exceeded 1 Mb/s and were much lower than the provided generic script data. While it cannot be said for certain why this is, it can be assumed that Apache's HTTP flooding security measures were able to cut down the amount of incoming data considerably when attacking without spoofed HTTP headers.

Even with the higher send rate, the requests to the query-making script did not have noticeable effects on the server with regards to either response time or loss of service. The queries may have been too simple to and the database too small. Given the time, I would have liked to have tried with a few different scripts with different kinds of queries; a script that performs several queries and one that does expensive operations like join.

B. HOIC Image retrieval attacks

The one set of attacks that did yield promising results were the attacks that retrieved an image. These attacks took a noticeable toll on the server. The average ping was much longer than the control or the previous database operation requests seen in ***Table 2***, showing noticeable slowdown as a result of the attacks. The ping standard deviation was also considerably higher during the image attacks, meaning that the quality of service was also fluctuating as a result.

HOIC also experienced prominent slowdown, the average sending rate decreased significantly and the program itself was not as responsive as the previous runs. HOIC's crashes may have to do with the image retrieval requiring the client to do about as much work as the server, thus causing both considerable stress (It may even be worse for the client since it needs to allocate space to store the response). Regardless, the fact that HOIC consistently crashed when attacking from a single machine while the server remained active demonstrates the importance of having a botnet for attacks; the server would require more stress relative to the attacker(s).

Further research would need to be made regarding the efficacy of GET flooding compared to POST flooding. While the image retrieval set yielded the attacks with the most obvious side effects, the experiments may not be sufficient to compare the two types of attacks. Ideally, the POST attacks should be simulated with a more realistic script and database; a sandbox environment hosting an application would be better for comparison. Given more time, this should be possible.

C. *Slowloris attacks*

The slowloris attacks showed an increase in response time from both a decrease in sleep time in between sending header data and from increasing the number of sockets being used in the attack. The number of sockets caused a more noticeable increase in the response time. The number of sockets should cause more threads to be created on the server side, meaning that legitimate requests make up a smaller percentage of threads being run, and thus get less computation time, causing slow down. The increase in response time correlated with sleep time can likely be attributed to an increase in the likelihood that a connection remains open between sending headers. Preventative measures are in place within Apache to close connections that are idle for too long. Refreshing by sending more frequent header requests seems to cause sockets to be closed less often, and thus their threads remain active for longer without needing to create a new thread. The amount of time between a socket being closed and being opened is also decreased as the sleep time decreases in the slowloris script, so attacking threads should spend more time on the server.

While the server did not experience any outages, the response time with even just the default 150 sockets and 15 second sleep times was noticeable when compared to a normal connection to the server. This seems to be a fair point to expect slowdown given our setup, since Apache's default settings support up to 150 concurrent connections. Most of the possible connections were likely occupied during the duration of the attack. It should be noted that the slowloris script does mention when it needs to re-open a connection, and that it regularly had to reconnect with new sockets in between sleeps. So, while the number of sockets being used were >= 150 across all runs, it is likely that not all were active on the server consistently and that some of the threads associated with these connections may have been closed automatically. Still, they were intrusive enough to cause noticeable slowdown for a user in-browser.

VI. Conclusions and Future Research

This paper aimed to demonstrate the effects of HTTP flooding and slowloris attacks in a controlled environment and show how severe they can be using low-end hardware and common server hosting software. Without using mitigation services, slowdown was observed when slowloris and HTTP GET attacks were launched. In the future, it would be nice to scale the research to more realistic server, using multiple machines to attack and test the stress on a dedicated server.

Further research could also be used into machine learning to detect attacking traffic and explore further models. RNN classification shows promise, and it is possible that a generative component could aid in identifying attack traffic. Research is still needed into specifically application layer attack detection, as most existing literature focuses specifically on volumetric flooding attacks. The payload of HTTP requests has not been heavily researched as a parameter in machine learning models.

References

[1] A. Praseed and P. S. Thilagam, "DDoS Attacks at the Application Layer: Challenges and Research Perspectives for Safeguarding Web Applications," in IEEE Communications Surveys & Tutorials, vol. 21, no. 1, pp. 661-685

[2] CloudFlare, "DDoS Attack Trends for 2021 Q3,"https://blog.cloudflare.com/ddos-attack-trends-for-2021-q3/.

[3] D. Warburton and E. Ojeda, "DDoS Attack Trends for 2020," F5, 2021.

[4] Filho, Francisco & Silveira, Frederico & Junior, Agostinho & vargas solar, Genoveva & Silveira, Luiz. (2019). "Smart Detection: An Online Approach for DoS/DDoS Attack Detection Using Machine Learning". Security and Communication Networks. 2019. pp 1-15.

[5] Mahadev, V. Kumar and K. Kumar, "Classification of DDoS attack tools and its handling techniques and strategy at application layer," 2016 2nd International Conference on Advances in Computing, Communication, & Automation (ICACCA) (Fall), 2016, pp. 1-6

[6] CockroachDB, "Insert Data," https://www.cockroachlabs.com/docs/v21.1/insert-data

[7] A. Makiou, Y. Begriche and A. Serhrouchni, "Improving Web Application Firewalls to detect advanced SQL injection attacks," 2014 10th International Conference on Information Assurance and Security, 2014, pp. 35-40

[8] B. Davis, "Mega Daily Download Link," 2 June 2021. https://www.mvorganizing.org/what-is-mega-daily-download-limit/#:~:text=As%20a%20free%20user%2C%20you,than%205%20GB%20of%20data.

[9] G. Yaltirakli, "Slowloris," Github.com, 2015.

[10] Devi, S and P. Yogesh, "Detection of Application Layer DDOS Attacks Using Information Theory Based Metrics", Computer Science & Information Technology, vol. 2, pp 217-223, 202

[11] CloudFlare, "What is a Slowloris DDoS attack?," https://www.wallarm.com/what/what-is-slowloris.

[12] S. Bravo and D. Mauricio, "DDoS attack detection mechanism in the application layer using user features," 2018 International Conference on Information and Computer Technologies (ICICT), 2018, pp. 97-100

[13] C. Thompson, "Why CAPTCHA Pictures Are So Unbearably Depressing," https://onezero.medium.com/why-captcha-pictures-are-so-unbearably-depressing-20679b8cf84a. [Accessed 9 November 2021].

[14] Y. Huang, J. Gornet, S. Dai, Z. Yu, T. Nguyen, D. Y. Tsao and A. Anandkumar, "Neural Networks with Recurrent Generative Feedback," 2020.

[15] P. Weaver, "Cheap and nasty: How for $100 low-skilled ransom DDoS extortionists can cripple your business," Imperva, 1 September 2021. https://www.imperva.com/blog/cheap-and-nasty-how-for-100-low-skilled-ransom-ddos-extortionists-can-cripple-your-business/.

[16] A. Nadav and K. Johnathan, "2019 Global DDoS Threat Landscape Report," 4 February 2020. https://www.imperva.com/blog/2019-global-ddos-threat-landscape-report/.

[17] Imperva, "DDoS Protection for Networks (Image)," https://www.imperva.com/products/infrastructure-ddos-protection-services/.

[18] W. -z. Lu and S. -z. Yu, "An HTTP Flooding Detection Method Based on Browser Behavior", 2006 International Conference on Computational Intelligence and Security, pp 1151-1154.

[19] Satish Pokhrel, Robert Abbas, and Bhulok Aryal, "IoT Security: Botnet detection in IoT using Machine learning", arXiv e-prints, 2021.

[20] Umarani, S. and D. Sharmila. "Predicting Application Layer DDoS Attacks Using Machine Learning Algorithms." World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering 8, pp. 1912-1917.

[21] H. Kousar, M. M. Mulla, P. Shettar and N. D. G., "DDoS Attack Detection System using Apache Spark," 2021 International Conference on Computer Communication and Informatics (ICCCI), 2021, pp. 1-5

[22] S. Nandi, S. Phadikar and K. Majumder, "Detection of DDoS Attack and Classification Using a Hybrid Approach," 2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP), 2020, pp. 41-47

[23] M. S. Elsayed, N. -A. Le-Khac, S. Dev and A. D. Jurcut, "DDoSNet: A Deep-Learning Model for Detecting Network Attacks," 2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM), 2020, pp. 391-396

[24] Lukaseder, Thomas & Ghosh, Shreya & Kargl, Frank. (2018). Mitigation of Flooding and Slow DDoS Attacks in a Software-Defined Network.

[25] Verma, Apurv & Xaxa, Deepak. (2016). A Survey on HTTP Flooding Attack Detection and Mitigating Methodologies. International Journal of Innovations & Advancement in Computer Science. 5.

[26] D. Lorenzi, E. Uzun, J. Vaidya, S. Sural and Vijayalakshmi, "Enhancing the Security of Image CAPTCHAs Through," HAL, pp. 354-368 10.1007/978-3-319-18467-8_24ff. ffhal01345127f.

[27] G. A. Jaafar, S. M. Abdullah and S. Ismail, "Review of Recent Detection Methods for HTTP DDoS Attack", Journal of Computer Networks and Communications, 2090-7141.

[28] M. A. Saleh and A. Abdul Manaf, "Optimal specifications for a protective framework against HTTP-based DoS and DDoS attacks," 2014 International Symposium on Biometrics and Security Technologies (ISBAST), 2014, pp. 263-267