Investigating The Robustness of Synchronous and Asynchronous Software Systems with Buffer Overflow Attacks

Cyrus Majd **HAR Period 8** **Apr. 12, 2019**

Abstract

Web servers and micro services constitute the backbone of modern IT service infrastructure, and such software systems can be accessed either synchronously or asynchronously. The asynchronous architecture is typically preferred due to its non-blocking and scalable nature and hence widely deployed in large datacenters. Because of the extensive use of this architecture over more traditional synchronous architecture, there is a need to investigate the robustness of asynchronous client-server architecture with respect to cyberattacks.

In this research project, the extent of damage from denial of service (DoS) buffer overflow cyberattacks on synchronous and asynchronous client-server software systems is quantitatively measured and analyzed. The results demonstrate that while asynchronous architecture is more scalable due to its non-blocking nature, it is more vulnerable to request-based buffer overflow attacks, which could effectively render critical services inaccessible to users.

Literature Review

Synchronous softwares are services that are optimized for reactive systems. These systems can see use cases in websites, peer-to-peer networks, and general internet traffic, as it provides a dynamic send-receive verification method to ensure safe data transfer between two computers.

However, new communication methods, which operate on asynchronous platforms (such as AJAX), offer greater flexibility and shorter response times for webapps. These systems are often installed on high priority checkpoints in data centers, such as switches and routers. They can also be installed on virtual machines that act as proxies for a network's traffic to pass through. Supratik Mukhopadhyay et. al. These data points are analyzed within a few seconds and a judgement made by the Anomaly Based system is returned as to whether or not to allow the network traffic to continue. These systems can help find live attacks and mitigate them while they occur, as opposed to current statically configured security systems which can only block known exploits and not detect any new ones.

These systems have their errors, however. While Anomaly Based Intrusion Detection Systems may identify and mitigate attacks while they occur, their reliance on machine learning constantly puts them at risk for an error in judgement. In other words, it is always possible for an Anomaly Based Intrusion Detection System to wrongly identify regular user traffic as a malicious cyber attack, if too many "red flags" are alarmed. This could put regular admins and datacenter employees at risk, since they constantly run penetration tests on the datacenters for research and security enhancement purposes.

In my project next year, I want to take advantage of the dynamic nature of Anomaly Based Intrusion Detection Systems and utilize the method of DoS attack mitigation I invented this year to create a dynamic, full datacenter mitigation tool. This tool could potentially eliminate the need for classic, statically configured security defenses such as load balancers, IDS/IPS systems, and restrictive firewall rules.

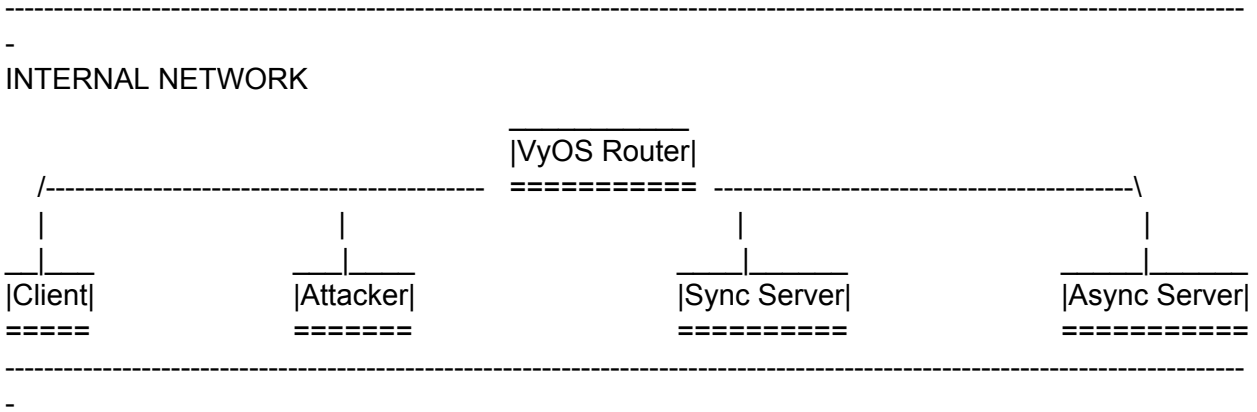
Experimental Design

This marking period consisted of setting up the virtual experimental environment and writing the programs necessary to host a functioning synchronous and asynchronous webserver.

Further improving on the experimental design and foundations set in the previous marking period, a testing environment was developed to simulate a real, isolated datacenter network. The testbed was created with the following parameters.

a. All computers that were used were VMs. The VMs run independently and are identical in technical specifications (same RAM, storage, CPU power/cores, etc.)
b. All VMs were connected to an *internal network interface*. This means that the VMs can communicate with each other, but cannot communicate to any outside source (the internet, other computers in the LAN, etc.)
c. All of these VMs were also configured to connect to a VyOS virtual router. Without this router, the VMs cannot establish any connections that require an IP address, such as FTP, HTTP services, and almost all other methods of communication, such as TCP.

To illustrate, the topology of the network looks like as follows:



*everything is connected to the VyOS router and these five machines are the only devices on this *isolated* internal network

The virtual machines were all hosted on Oracle VM Virtualbox, and were all given 3 gigabytes of RAM, 1 logical processor, and 10 gigabytes of storage. All VMs other than the VyOS router machine are running the latest version of Ubuntu Linux.

Now that each machine can communicate with each other, it was time to write the code for the network connections. A webapp was developed in order to simulate a real interaction between a client and a synchronous/asynchronous webserver. This webapp was coded in java, and the code was split into four parts; a code for the client (normal user), a code for the attacker

(someone who attempts a buffer overflow attack on the webserver or exploits process threads), codes for the synchronous server, and codes for the asynchronous server.

These codes were then packed into executable file formats (.jar) and sent to the VMs to be run on them. The VMs also had to have their path variables set so they could run the programs. Detailed descriptions of the two main codes can be seen below.

```
1  package com.cyrus.har.attackcode;
2
3  import java.util.Scanner;
15
16 public class AttackClient {
17     public static void main(String[] args) throws JSONException {
18
19         final Client client = ClientBuilder.newClient(); // Client is an interface. ClientBuilder implements it
20         int attackPeriod;
21         final ScheduledExecutorService executorService = Executors.newSingleThreadScheduledExecutor();
22
23         Scanner scanner = new Scanner(System.in);
24         System.out.println("Type in the attack period in miliseconds: \n");
25         attackPeriod = scanner.nextInt();
26         scanner.close();
27
28
29         executorService.scheduleAtFixedRate(new Runnable() {
30
31             @Override
32             public void run() {
33
34                 // http://localhost:8082/attackcode/webapi/server/1 -> Synchronous Server
35                 String response = client.target("http://10.0.0.48:8080/attackcode/webapi/attackedserver/1")
36                         .request(MediaType.TEXT_PLAIN)
37                         .get(String.class);
38
39                 System.out.println("Respnse to client: " + response);
40             }
41         }, 0, attackPeriod, TimeUnit.MILLISECONDS);
42     }
43 }
44
```

Figure 1. This is the attacker's code. The attacker sends as many http "get" requests to the targeted server as possible. The targeting can be observed in the highlighted line, which indicates the IP address of the target. The following two lines indicate that the http packet being sent to that IP is an http get request that is looking for a plaintext response (a "string"). The regular user

client code nearly identical to this code, except for the fact that the client only sends requests when a response from the previous request is received.

```
2
3  import java.util.concurrent.TimeUnit;
14
15 @Path("/storage/{attackCount}")
16 @Consumes(MediaType.TEXT_PLAIN)
17 @Produces(MediaType.TEXT_PLAIN)
18
19 public class Storage {
20
21     int returnValue;
22     int randomNumber;
23     JSONObject jsonObject = new JSONObject();
24
25     // ============Method called for attacking the Server==========
26
27     @GET
28     @Produces(MediaType.TEXT_PLAIN)
29
30     public String getJson(@PathParam("attackCount") int attackCount) throws JSONException, InterruptedException {
31         Long storageTime_in = System.currentTimeMillis();
32
33         // ----------------------Querying Server---------------------------------
34
35         for (int i = 0; i <= 5000000; i++) {
36             randomNumber = (int) (Math.random() * 50 + 1);
37         }
38
39         // ----------------------End of Query----------------------------
40
41         Long storageTime_out = System.currentTimeMillis();
42
43         jsonObject.put("attackCountValue", attackCount)
44                 .put("storageReturnValue", randomNumber)
45                 .put("storageStartTimeStamp", storageTime_in)
46                 .put("storageEndTimeStamp", storageTime_out)
47                 .put("cumulativeServerDelay", (storageTime_out - storageTime_in)); // needs correction
48
49         System.out.println();
50         System.out.println("Storage Json: " + jsonObject);
51         return jsonObject.toString();
52
53     }
54
55     // ================================================================
56
57 }
58
```

Figure 2. This is the server code. In line 27, it is indicated that it listens for an http “get” request from a client. In lines 35 and 36, it randomly generates a number, which calls for a process thread. It is hypothesized that the attacker can exploit these process threads to overload the

server. The code then takes record of the time and returns the request in line 51 back to the originating IP address.

Data and Results

Results showed that the Asynchronous client-server software system was successful in conducting a denial of service attack and crashing the server as a result of a buffer overflow attack, but the Synchronous client-server software system was unable to crash the server. This is likely because the Synchronous system can only send one request at a time - and thus cannot fill up the server's buffer - while the Asynchronous system can send many thousands of requests at a time, which can fill up the server buffer very quickly if requests are send in faster than they can be processed.

Figures 2 and 3 further illustrate the results of the project.

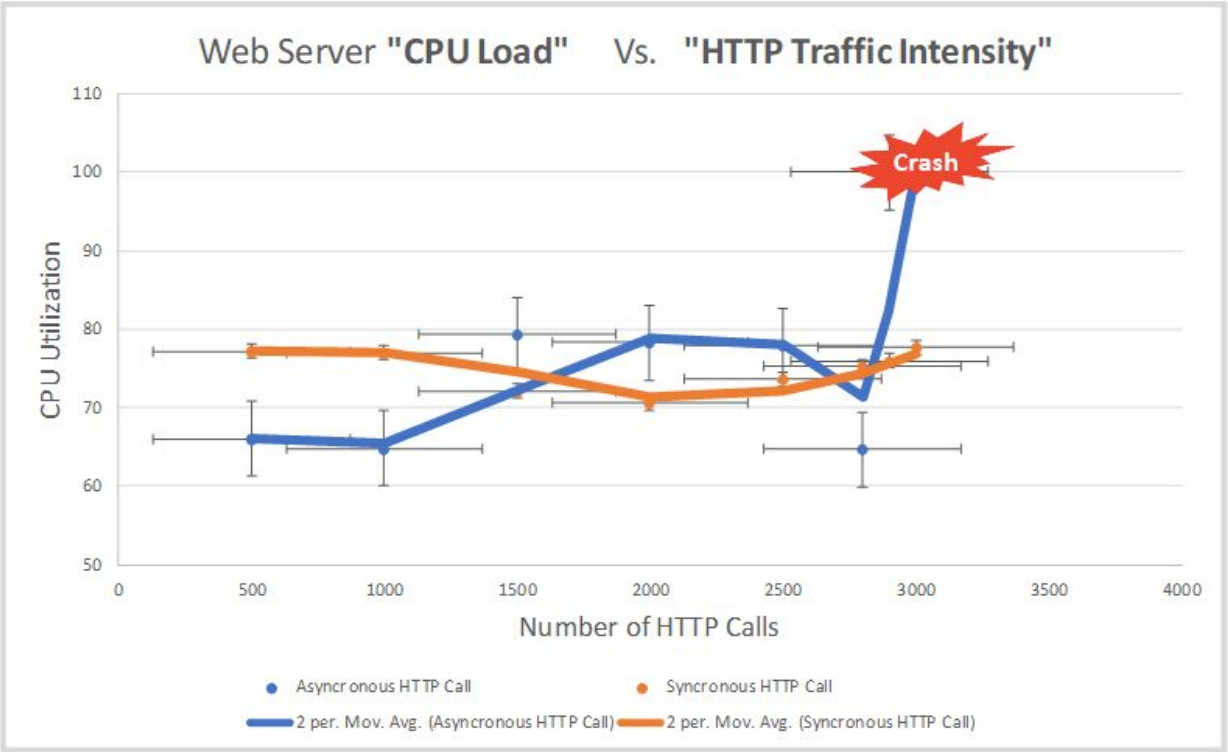
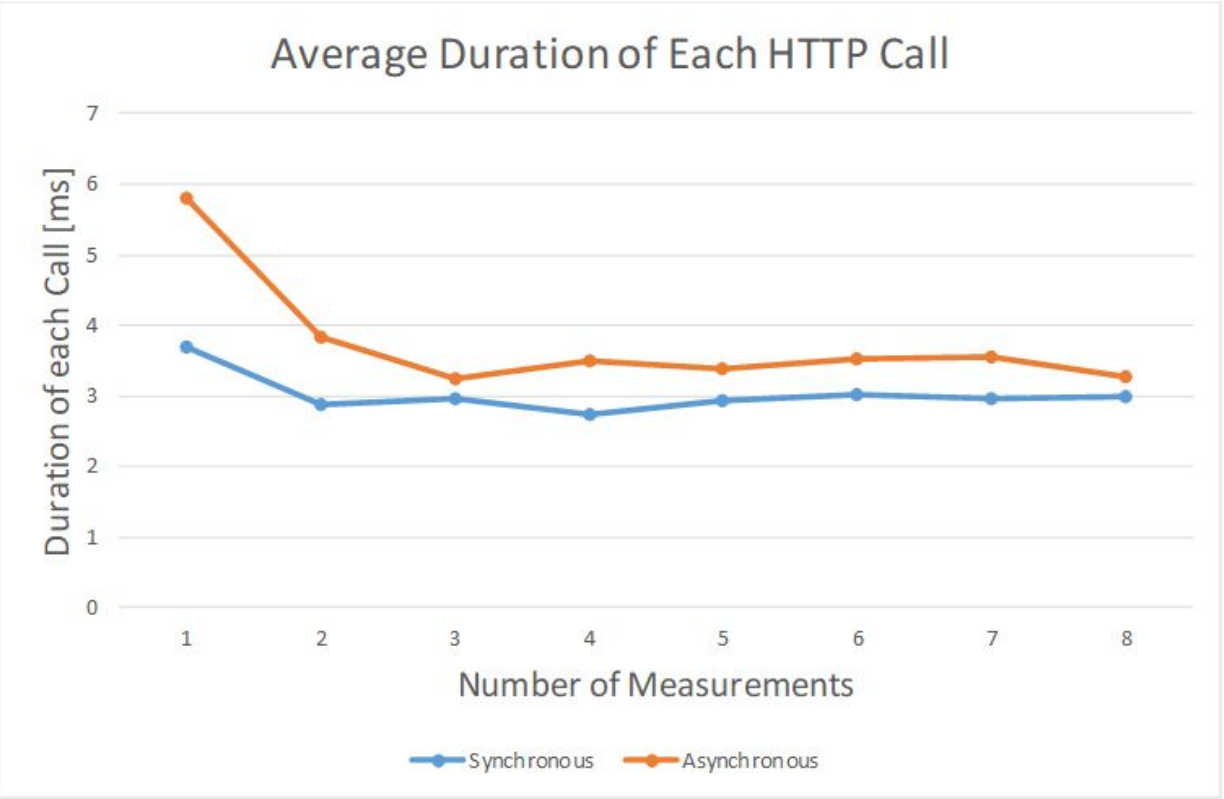


Figure 2 (above) Figure 3 (below):



After all of this was complete, some very basic statistical tests were conducted to determine factors such as standard deviation, which can also be identified in figures 2. The project was then printed and presented on a poster at JSSF.

References:

1. Pescatore, J. “Defining Intrusion Prevention.” *Umich*, 29 May 2013, www.bus.umich.edu/KresgePublic/Journals/Gartner/research/115200/115237/115237.pdf .
2. Thanh Van, Nguyen & Ngoc Thinh, Tran & Thanh Sach, Le. (2017). An anomaly-based network intrusion detection system using Deep learning. 210-214. 10.1109/ICSSE.2017.8030867.
3. Khalkhali, Iman, et al. “Host-Based Web Anomaly Intrusion Detection System, an Artificial Immune System Approach.” *IJCSI*, International Journal of Computer Science Issues, Sept. 2011, www.ijcsi.org/papers/IJCSI-8-5-2-14-24.pdf.
4. Aarseth, R. (2015, September). Security in cloud computing and virtual environments. Retrieved October 7, 2016, from http://bora.uib.no/bitstream/handle/1956/10695/138625252.pdf;jsessionid=F32AF4096B3F0FFEA838BFF5F278879F.bora-uib_worker?sequence=1