Protecting Web Applications from Web Scraping

Baftjar Tabaku and Maaruf Ali(✉)

Epoka University, Rruga Tiranë-Rinas, Km 12, 1032 Vorë, Tiranë, Albania
maaruf@ieee.org

Abstract. Automated software programs called, "bots", are being used extensively to extract, collect and harvest information from the Internet across any website that can be accessed. This unsolicited and intrusive gathering of information, knowns as "web scraping" is then subsequently used in a way that can lead to damaging the integrity, authenticity and reputation of the victim website. The problems of web scraping are identified, along with the methods used to prevent the web scraping are explored and explained. Prevention using rate limit detection, identification of automated traffic and known malicious entities are described including the use of honey-potting. The success of using these techniques is concluded by the positive outcome of the results.

Keywords: Web scraping · Web data extraction · Web harvesting · Web crawling · Information extraction · Google cloud services · Data integration · Interoperability

1 Introduction

1.1 Background

The Internet always is considered a data centre where an outstanding amount of information has become easily accessible to be used through a web-browser. Web-scraping is considered an internet-based data collection technique where a script of code will make continuous automated requests to different webpages and then store the data taken from these webpages for further offline processing.

There is an increasing demand for new and fresh data, however, this cannot be done manually. Manual extraction of data by an end user is almost impossible because of the sheer required volume of information to be collected. This web collection of data has evolved into an automated processes most often using bots, this new concept is known as "Web Scraping".

Without any protection will leave web-applications vulnerable to web-scraping that will compromise the security of the users' data. This will consequently allow third parties to use this data in a way that the original data owners never intended nor permitted. This treatise describes a few techniques to prevent web-scraping. To implement this, Google Cloud Platform was utilised.

M. H. Miraz et al. (Eds.): iCETiC 2021, LNICST 395, pp. 56–70, 2021.
https://doi.org/10.1007/978-3-030-90016-8_4

1.2 Web Scraping

Web scraping is a widely used method of collecting internet data where a code script will be executed repeatedly making requests across different websites and then storing their responses, the data, for offline processing. By leaving websites or web-applications unprotected and vulnerable, it will lead to compromising the security, integrity and reputation of the website. This is because the sensitive and valuable data that has now been extracted without the original owner's permission - can be used by the third parties in ways not originally intended. Figure 1, below, show a typical web scraping scenario.

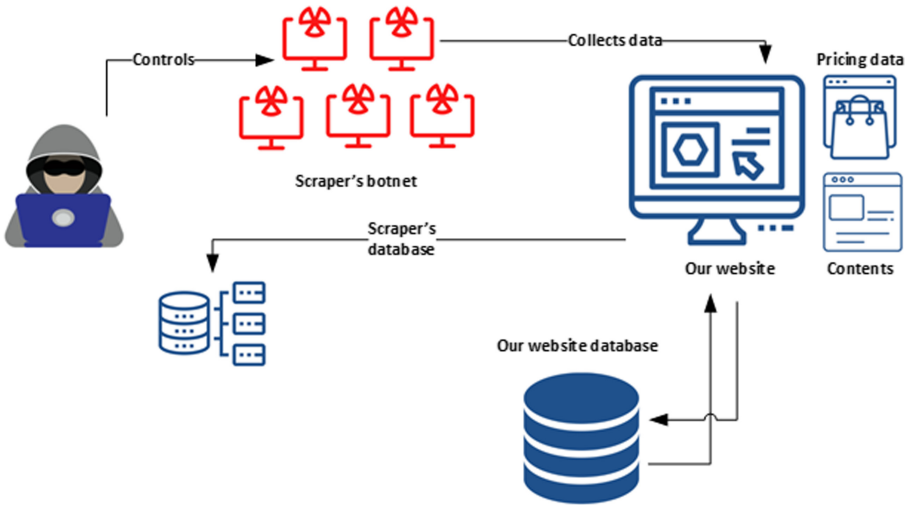

Fig. 1. General website content scraping schema.

2 Preventing Web Scraping

Web scraping is a content extracting process using bots to extract valuable data from one or more websites. It is different from "screen scraping" where only copies of the pixels that are displayed on a screen are made. Web scraping in contrast will only extract the HTML source code and its associated data. After that the scraper can replicate the entire source website source anywhere else, e.g. on a different e-commerce website.

To prevent web-scraping four methods are mentioned here:

1. **Rate limiting:** this limits the rate of requests on a website made from a client to a server that provides the service. For example, if the website service is configured to only 50 requests per second for a client (depending on the service provider capacity), then if this number of requests is exceeded, an error will occur and this client cannot then generate any more requests according to the limit threshold.
2. **Identifying the automated traffic:** this can be identified by detecting non-human behaviour in web traffic information exchange. One of the traffic indicators which

can be utilised is the volume of traffic request, (e.g. making more than 10,000 requests on a given day on a server, even if the rate limit is never exceeded). This will be done automatically on a periodic time per minute or 24-h daily.

3. **Known malicious identifiers:** one of the best practices to protect web applications is to reject different requests from known malicious entities and end users. There are online services and websites that maintain lists of banned or black IPs which can be used to deny their access for the web applications by blocking their IPs. The scraping endpoint user can also be identified by the user-agent. If there is someone trying to scrape the website then its user-agent will contain at least one of these terms like 01h4x.com, 360Spider, 404checker, 404enemy, 80legs, Abonti, Aboundex, etc.
4. **Honey Potting:** this is another technique that protects web applications from being scraped. In this technique, links or buttons to the webpage are placed or planted strategically to make them appealing so that only the bots can latch onto them and access them. These buttons or links are not displayed in the browsing page, so they cannot be used by simple users, if someone clicks them, that means that a bot is operating in the web application or website.

2.1 Rate Limit

Rate limiting will be used to control the incoming and outgoing traffic to a network. For example, if a particular API (application programming interface) service is being used, then that will be configured to allow 1000 requests/minute. If the limit of 1000 requests are exceeded, then an error message will be triggered in the user browser window.

Implementing a rate limit is important for better data flow and increased security from DDoS (distributed denial-of-service) attacks. The rate limit becomes critically useful, if a user on a certain network makes a mistake by sending unlimited requests to a server and retrieving gigabytes or unreasonably higher amounts of information that will also overload the network for everyone. With rate limiting, these attacks would be easily caught and managed by termination (by blocking their requests). Figure 2, below, shows a bar graph display of exceeding the rate limit where the attack condition is detected, managed by blocking and the subsequent reduction of the traffic requests with time.

Types of Rate Limiting

1. *User Rate Limiting.*
   The most popular rate limiting is the user rate limiting where the number of requests that a user will make to an API key or an IP (depends on the user agreed preferences) address. If the user exceeds the request number limit, then any further request for this user will be denied until they reach out to the developer and the API request time limit is reset. Figure 3, below, shows an example where the policy configuration is set to five requests within a ten second window with one retry permitted with a 500 ms delay.

2. *Geographic Rate Limit.*
   To increase the security in certain geographic locations, a regional rate limit can be set with an added temporal or period of operation as well. For example, if in a

Fig. 2. Rate limit threshold detection and application of rate limiting [1].

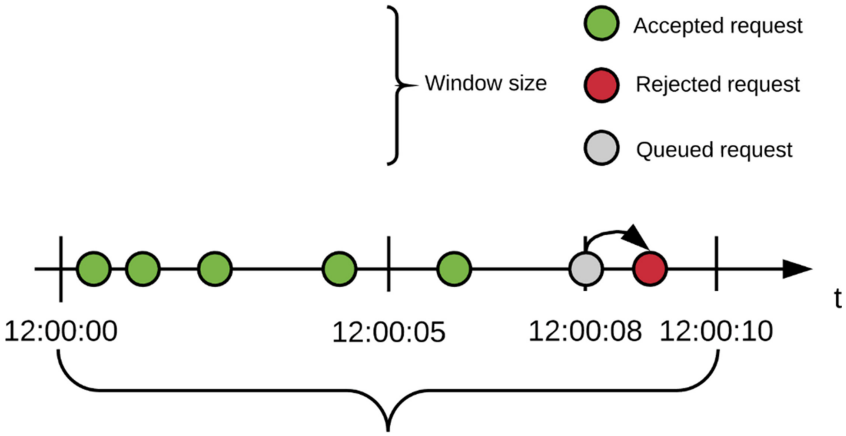

Fig. 3. User rate limiting and throttling [2].

particular region at a certain time the developers expect a low rate of requests, they can set up a lower rate of requests per second limit. This action will prevent potential attacks and risks of suspicious activity for the targeted web-applications. An example of an access denied and the geographic blocking reason message is shown in Fig. 4, below.

3. *Server Rate Limit.*
Certain servers will handle certain aspects of their applications and their rate limit will be defined based on a server level-basis. This will allow developers finer control to give them the opportunity to increase or decrease the rate limits on a server-by-server basis. As shown in Fig. 5, below, if Server C exceeds its rate limit, then traffic may be diverted or load balanced to Server B. The servers send information about

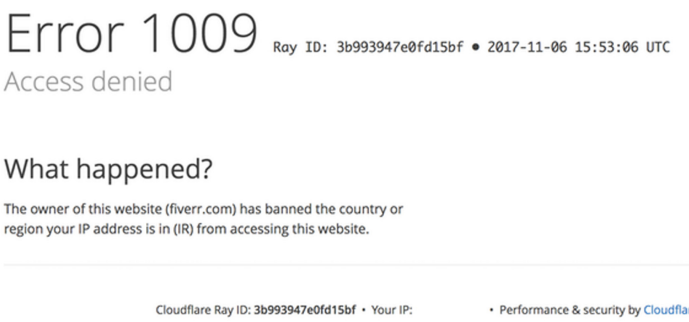

Fig. 4. IP address(es) banning through geographic location [3].

the request volumes and the rate-limiting service responds with the rate-limiting decisions.

The rate limit can be implemented using various methods at the server level using various programming languages. For this research, it can be implemented using a Nginx or Apache server.

2.2 Identifying the Automated Traffic

A bot, that is, the end-user generating the automated traffic – may submit queries for a variety of different reasons, most of which are benign and not overly monetisable. As an example, rank bots periodically scrape web pages to determine the current ranking for a <query,URL> pair. A Search Engine Optimization company (SEO) may employ a rank bot to evaluate the efficacy of its web page ranking optimisations for its clients. If a client's current rank is low, a user may need to generate many NEXT PAGE requests to find it in the search engine's results [5].

Figure 6, above, shows the query traffic for a typical bot over a 24-h period. Another way of protecting the web-applications from web-scraping is to identify automated traffic. A group of end-users scraping a webs-application or website would try to prevent extending the rate-limits by slowing down their rate request quota or even making them at random times. Doing this they are still detectable through their access patterns.

The bot end users can be detected by analysing their automated traffic on our applications and identifying entities (end-users, in this case bots) that does not behave as usual normal users when accessing our application.

[6] identified the patterns that indicate typical automated traffic. There are two main patterns in particular that identify the automated traffic which are:

1. The traffic volume, entities or end-users will usually send more than +500 requests to a web-server from a single IP address in a time interval that usual human users can never cannot complete in that same time period.
2. The event periodicity, where the end-users or bots send requests to a server that website or web-application is hosted on every minute on an exact time period from the previous one.

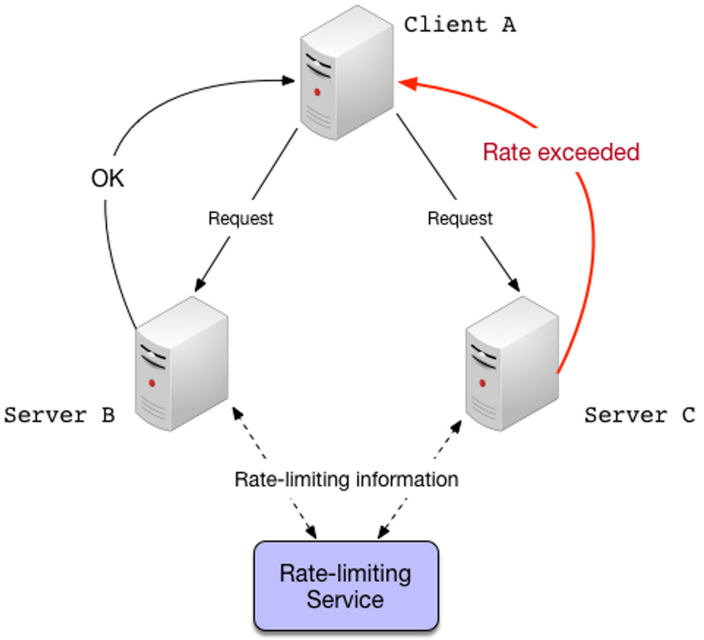

Fig. 5. Global rate-limiting with a central server [4].

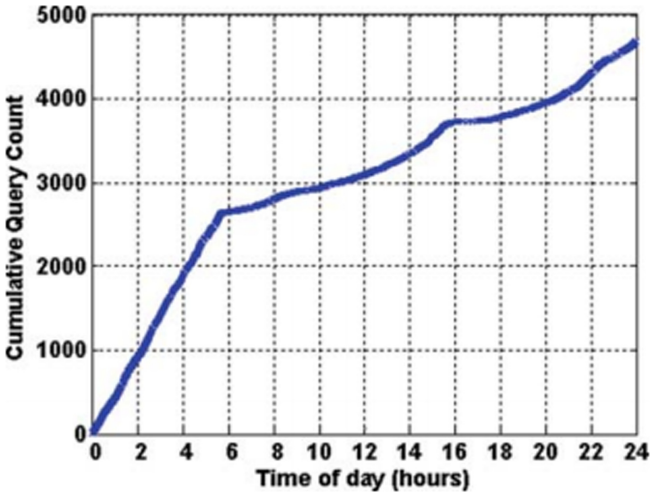

Fig. 6. A graph depicting the time of day versus aggregate queries for a typical bot [5].

2.3 Known Malicious Identifiers

An alternative way of protecting web-applications or websites from web scraping is to block access to the website or web-applications from these malicious entities or end-users (bots). This can be accomplished by checking the blacklisted IP addresses or

User-Agents that are based on popular web-scraping libraries. These IP addresses can then be simply blacklisted and those scrapping library User-Agents blocked. Figure 7, below, illustrates blocking at the ISP (internet service provider) level to an international website, in this example, to 198.51.100.126.

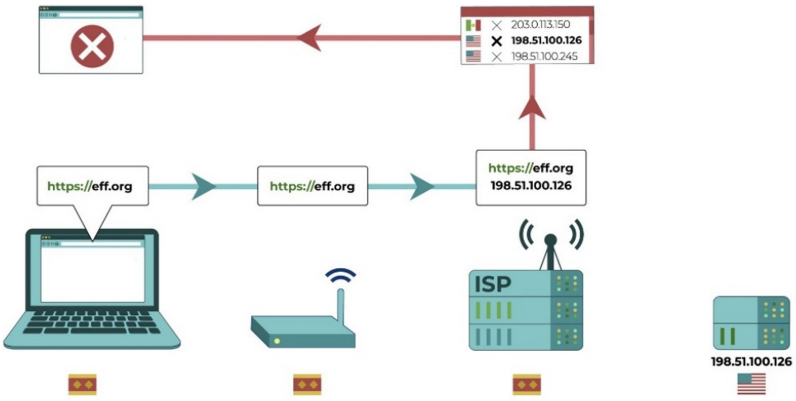

Fig. 7. Blocking of websites at the ISP level by using a black list of IP addresses [7].

2.4 Honey-Potting

This sneaky technique is used to protect web applications and websites from web-scraping. A honeypot is a mechanism used in computer security to detect illegal attempts at using information in an unauthorised manner. The way that honey-potting was used in this project was by adding links that were invisible on the web-browser screen and not clickable by end users. They could only be detected by bots and clicked only by them. By clicking on a honeypot, the bot exposed itself and then its access was blocked from further accessing the website. Figure 8, below, illustrates the deployment of a honeypot at the periphery and outside the enterprise internal network. The scanning attack is caught and logged by the honeypot network entity.

3 Methods and Tools Employed

The hardware used for this research is listed below:

- Device name DESKTOP-F7RINFF
- Processor Intel® Core™ i7-7700HQ CPU @ 2.80 GHz 2.81 GHz
- Installed RAM 16.0 GB (15.9 GB usable)
- Device ID A3604B8D-083F-44CC-AEBD-E10B0FCEF4B6
- Product ID 00331-10000-00001-AA770
- System type 64-bit operating system, ×64-based processor
- Pen and touch No pen or touch input is available for this display.

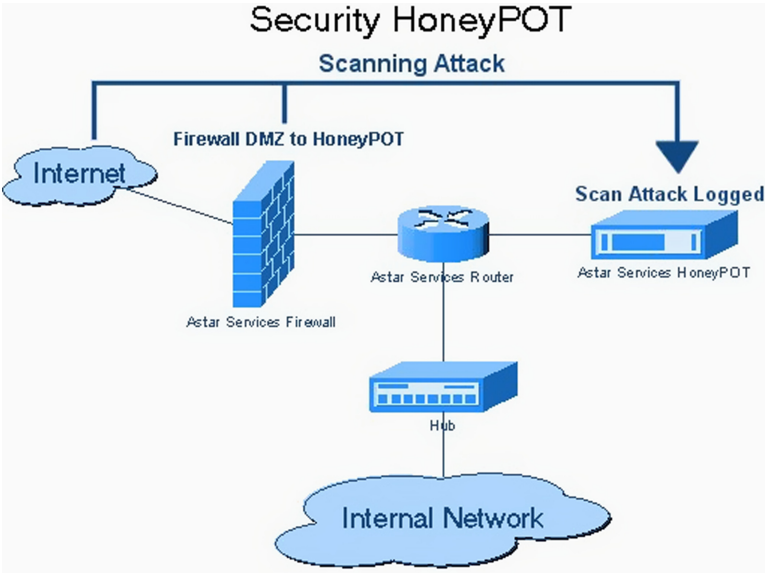

Fig. 8. Deployment of an information system honeypot [8].

- Edition Windows 10 Pro
- Version 20H2
- Installed on 1/23/2021
- OS build 19042.1083
- Experience Windows Feature Experience Pack 120.2212.3530.0

3.1 Experimental Part

There are different methods used to prevent web-applications from web-scraping, limiting the rate of the API was implemented.

API Rate Limiting

Even if there are different methods used to limit the rate of the API, in this experiment, a java servlet filter was used that was setup to limit the request rate using the Bucket4J library. This limited the incoming requests based on a token-bucket algorithm.

The example used here had one endpoint, where the users were taken from the database. Without rate-limiting the users' data at this endpoint would mean that their data could be scrapped and collected. However, by implementing the rate limiting method, it prevented the scraping part.

It was implemented using the Rate Limiting Filter class based on Bucket4J [9]. By using this filter, the rate limit that an IP can send and receive requests would be limited to 50 requests/second. For a better result, the IP addresses are stored to a hash-map where the IP are the keys and the buckets are the values.

The Java code utilised for this project is given next.

```
import io.github.bucket4j.Bandwidth;
import io.github.bucket4j.Bucket;
import io.github.bucket4j.Bucket4j;
import utils.RequestParser;
import utils.ScrapingEnforcer;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.time.Duration;
import java.util.HashMap;
import java.util.Map;
import java.util.logging.Filter;

public class RateLimitingFilter implements Filter {

    private Map<String, Bucket> bucketMap;
    private Bucket createNewBucket() {
        Bandwidth limit = Bandwidth.simple(1, Dura-
tion.ofSeconds(1));
        return Bucket4j.builder().ad-
dLimit(limit).build();
    }


    @Override
    public void init(FilterConfig filterConfig) {
        bucketMap = new HashMap<>();
    }

    private Bucket getBucket(String ip) {
        if (!bucketMap.containsKey(ip)) {
            bucketMap.put(ip, createNewBucket());
        }
        return bucketMap.get(ip);
    }

    @Override
    public void doFilter(ServletRequest servletRequest,
ServletResponse servletResponse,
                         FilterChain filterChain) throws
IOException, ServletException {
        HttpServletRequest httpRequest = (HttpS-
ervletRequest) servletRequest;
        String ip = RequestParser.getClientIp(httpRe-
```

```
quest);
        Bucket bucket = getBucket(ip);


        // tryConsume returns false immediately if no to-
kens available with the bucket
        if (bucket.tryConsume(1)) {
            // the limit is not exceeded
            filterChain.doFilter(servletRequest,
servletResponse);
        } else {
            // limit is exceeded
            ScrapingEnforcer.enforce(servletResponse);
        }
    }
}
```

This filter will constantly check and ensure if an IP address has yet exceeded its rate limit. If the rate limit is exceeded, then a 429-error status code that says "Too many requests" will be shown.

The next segment of code shows the client or UserServlet.java [9].

```
import com.fasterxml.jackson.databind.ObjectMapper;
import models.EventConstants;
import models.User;
import repositories.EventLogRepository;
import repositories.UserRepository;
import utils.RequestParser;
import utils.RequestType;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.sql.SQLException;
import java.sql.Timestamp;
import java.util.List;

@WebServlet(name = "servlets.UserServlet")
public class UserServlet extends HttpServlet {

    private final UserRepository userRepository = UserRe-
pository.getRepo();
    private final EventLogRepository eventLogRepository =
EventLogRepository.getRepo();

    private void preProcessResponse(HttpServletResponse
response) {
        response.setContentType("application/json");
        response.setHeader("Access-Control-Allow-Origin",
"*");
    }

    private void handleInvalidRequest(HttpServletResponse
resp) throws IOException {
        resp.setStatus(HttpS-
ervletResponse.SC_BAD_REQUEST);
        resp.getWriter().write("Invalid Request");
    }

    private void logGetUsersByPage(HttpServletRequest re-
quest) {
        try {
            eventLogRepository.insertEventLog(
                    RequestParser.getClientIp(request),
                    EventConstants.GET_USERS_BY_PAGE,
                    RequestParser.getPageNumber(re-
```

```
quest).toString(),
                    new Timestamp(System.currentTimeMil-
lis())
            );
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
    }

    private void handleGetUsersByPage(HttpServletRequest
request, HttpServletResponse resp) throws IOException {
        ObjectMapper mapper = new ObjectMapper();
        resp.setStatus(HttpServletResponse.SC_OK);
        List<User> users = userRepository.getTweetsBy-
Page(RequestParser.getPageNumber(request));
        resp.getWriter().write(mapper.writeVal-
ueAsString(users));
        logGetUsersByPage(request);
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpS-
ervletResponse resp) throws IOException {
        preProcessResponse(resp);
        RequestType requestType = RequestParser.parse-
Request(req);
        switch (requestType) {
            case INVALID:
                handleInvalidRequest(resp);
                break;
            case GET_USERS_BY_PAGE:
                handleGetUsersByPage(req, resp);
                break;
            default:
                throw new IllegalStateException("Unex-
pected value: " + requestType);
        }
    }
}
```

It was also necessary to include some URL specifications that were recorded through an xml file [9]:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-in-
stance"
        xsi:schemaLoca-
tion="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
        version="4.0">

<filter>
<filter-name>RateLimitingFilter</filter-name>
<filter-class>servlets.RateLimitingFilter</filter-class>
</filter>

<filter-mapping>
<filter-name>RateLimitingFilter</filter-name>
<url-pattern>/users/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>HighVolumeFilter</filter-name>
        <filter-class>servlets.HighVolumeFilter</filter-
class>
    </filter>

    <filter-mapping>
        <filter-name>HighVolumeFilter</filter-name>
        <url-pattern>/users/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>EventPeriodicityFilter</filter-name>
        <filter-class>servlets.EventPeriodicityFil-
ter</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>EventPeriodicityFilter</filter-name>
        <url-pattern>/users/*</url-pattern>
    </filter-mapping>

    <filter>
        <filter-name>MaliciousIdentifierFilter</filter-
name>
        <filter-class>servlets.MaliciousIdentifierFil-
ter</filter-class>
```

```
    </filter>

    <filter-mapping>
        <filter-name>MaliciousIdentifierFilter</filter-
name>
        <url-pattern>/users/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>UserServlet</servlet-name>
        <servlet-class>servlets.UserServlet</servlet-
class>
    </servlet>

    <servlet-mapping>
        <servlet-name>UserServlet</servlet-name>
        <url-pattern>/users/*</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>HoneyPotServlet</servlet-name>
        <servlet-class>servlets.HoneyPotServlet</servlet-
class>
    </servlet>

    <servlet-mapping>
        <servlet-name>HoneyPotServlet</servlet-name>
        <url-pattern>/honeypot/*</url-pattern>
    </servlet-mapping>

</web-app>
```

Rate limiting is not the best nor safest way to protect an application from being scraped, because the correct rules must be defined to set the right value for the rate-limit. Since IP addresses are used, a malicious user can define or change its IP proxy several times and then it can still scrape the web-application without limits. This action can also ban or prevent users that share one single IP address from using the web-application. Since there are too many users with the same IP address, then they will all be banned even if they are not doing anything. Due to these issues and reasons, it is important that this technique should be configured properly to give protection.

4 Results and Discussion

There is a plethora of penetration tools: illegal, legal, customised and Metasploit. The Metasploit tool that was used for testing was very comprehensive, effective and powerful. It offers a lot of options and possibilities to use it both manually or automatically according to the needs of the user.

By doing these experiments manually, it gave more possibilities to control the process flow and finer opportunities and control over exploiting a certain process - even if it meant that more time was required. Every IT system needs continuous work to always be one step ahead from the malicious attackers.

5 Conclusions

Response Rate limiting can be a great method to help fight against infrastructure attacks as well as block other types of suspicious activity like bots or different malicious activity in the attacked web applications or web pages.

There are various methods that can be utilised to implement rate limiting whether it be at the server level or user level. This can be implemented by blocking the access of the various IP addresses that are in the blacklist or by geographical location banning.

If an HTTP error code 429 Too Many Requests error is being experienced for a particular API, the developers should still be reachable to inform them to check their documents to verify what the rate limit is currently configured to be. The developers need to be instructed to modify their usage to fit within those limits according to the app-usage or requests to the website or web-application.

References

1. https://www.keycdn.com/img/support/rate-limiting-md@2x.webp. Accessed 26 July 2021
2. https://docs.mulesoft.com/api-manager/2.x/_images/throttling-rejected-request.png. Accessed 26 July 2021
3. McDonald, A.: Why geoblocking means the web is no longer worldwide, The Conversation, 11 December 2018. https://assets.weforum.org/editor/large_C_uKFkQFhmf1jdl_nGv-PBZrkpdLGrkDvMt6T-xcM40.png. Accessed 26 July2021
4. https://engineering.grab.com/img/beyond-retries-part-1/image1.png. Accessed 26 July 2021
5. King, I., Baeza-Yates, R. (eds.): Weaving Services and People on the World Wide Web, 1st edn. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00570-1
6. Buehrer, G., Stokes, J.W., Chellapilla, K., Platt, J.C.: Classification of Automated Web Traffic, Microsoft, pp. 1–27 (2009). https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/ClassAutoSearchTraffic.pdf. Accessed 26 July 2021
7. https://ssd.eff.org/files/2020/04/25/circumvention-networkshutdown.jpeg. Accessed 26 July 2021
8. https://upload.wikimedia.org/wikipedia/commons/7/76/Honeypot_diagram.jpg. Accessed 26 July 2021
9. Bukhtoyarov, V.: bucket4j. https://github.com/vladimir-bukhtoyarov/bucket4j/blob/master/doc-pages/basic-usage.md. Accessed 26 July 2021