Malwareless Web Analytics Pollution (MWAP) Tools Analysis

Introduction

The history of Web analytics dates back to the early days of the World Wide Web itself. In the early stages, web analytics was mostly used for diagnosing, troubleshooting and to encounter various errors. However, as the various firms and businesses started to heavily rely on the Web, Web analytics became an integral part of the online world which gives important insights about how an online business is performing.

The generated analytics data is in most cases a crucial part of the overall data which marketing teams base their decisions on. This latter fact has been always at the interest of malicious hackers. The ultimate goal of an attacker is harming the overall performance of a firm or business and polluting the Web analytics data is one of the simplest ways to achieve this goal.

Previous Work

The key characteristics of the MWAP attack are that it is very easy to execute, as it does not involve the use of any malware [1]. A Web Analytics Pollution attack is successful when the Web requests generated by the attacker are ultimately labelled as genuine human requests by the Web analytics tool deployed on the target site.

There are two main approaches to pollute the analytical data generated by Web analytics tools: Malware-based and Malwareless-based approaches. In this project, we have analyzed the second approach as it is much easier to conduct and in the real world, such attacks are more preferable due to the lesser cost and complexity.

In her previous work, Professor N. Vlajic has described and deployed an experimental framework to study the effectiveness of MWAP attacks. In her paper, Professor N. Vlajic has also provided a brief summary of the key findings which show a clear performance superiority of the Web analytics tools that utilize JavaScript page tags as a means of identifying genuine human-generated Web-requests. However, even such Web analytics tools are not completely prone to MWAP attacks [1]. In fact, Professor N. Vlajic's work shows the potential of abusing some smarter tools to narrow the adversaries' approach and perform successful MWAP attacks.

In this project, we have evaluated the overall feasibility and effectiveness of conducting successful MWAP attacks. To make our analysis more realistic, we have used free and readily available tools traditionally deployed for application layer testing and/or DDoS attacks.

Narrowing Down the Analysis

We have focused on two of the most famous Web analytical tools; AWStats and Google Analytics are representatives of two different categories of Web analytics tools. AWStats is a self-hosted solution that deploys access logs only. On the other hand, Google Analytics is a hosted (Software-as-a-Service) solution that deploys embedded page tags (JavaScript snippets) only.

In particular, our project is analyzing the overall feasibility and effectiveness of six free and publicly available tools for generating Web/HTTP traffic. The following tools have been described as "second" and "third" generation of Web bots by [2] and [3].

The second-generation bots/tools that have been evaluated in our study include the following three 'classical' application-level DDoS tools [4], [5]:

- HULK (HTTP Unbearable Load King)
- DDOSIM-Layer 7 DDoS Simulator
- GoldenEye HTTP Denial of Service Tool

The third-generation bots/tools that have been evaluated in our study include the following three popular headless browsers [6], [7], [8]:

- Headless Chrome
- Firefox Headless Mode
- HtmlUnit

The Plan

To be able to evaluate the effectiveness of the examined tools we need to have full control of the target Web site and its installed Web analytics tool. Therefore, we have developed and deployed one victim Web site and deployed it on the AWS, which is using the two chosen Web analytics tools.

After studying the six listed traffic generating tools above, we have performed the MWAP attack using each of them i.e. generating HTTP(s) requests towards the "target" Web sites and analyze the labelling of these requests by the installed Web analytics tool. As mentioned earlier, the requests that end up being categorized as genuine "human" implies a successful WAP attack.

The Experiment

The six tools that have been examined resulted in successfully fooling both of the web analytics engines AWSTATS and Google Analytics. The first three tools HULK (HTTP Unbearable Load King), DDOSIM (Layer Seven DDoS Simulator) and GoldenEye (HTTP Denial of Service Tool) could theoretically ruin the result of AWSTATS engine since all of them make the web server (in our case Apache) to log the http requests. Since AWSTATS only look at these access logs, it counts all of them as valid visits. However, only the attack using HULK was successful and all the requests made using DDOSIM and GoldenEye were instantly dropped by the webserver (by using default settings on the webserver). In addition, they were not able to fool Google Analytics since they simply can not fetch the whole page and run the JavaScript segment attached to the page header.

However, all of the last three tools Headless Chrome, Firefox Headless and HtmlUnit as well as fooling AWSTATS, were also successful in making fake visits that Google Analytics counts legitimate. In the following report, I get to details of each tool as well as the results obtained by using each them.

The target web site is hosted on an AWS Ubuntu 18.04 EC2 instance which uses an Apache service to serve the website. The website is accessible at the following link:

http://eloki.tk

The repository for this project has been published on https://github.com/saaniaki/eecs4480.

HULK (HTTP Unbearable Load King)

HULK is a web server denial of service tool written for research purposes. It is designed to generate volumes of unique and obfuscated traffic at a webserver, bypassing caching engines and therefore hitting the server's direct resource pool.

The version that has been used in this research was not the original python script; the version used was ported to Go language from Python. The main difference from Python version layered in Golang architecture is that it can handle concurrency; the “goroutines”. “hulk.py” runs a new thread for each connection in the connection pool so it uses hundreds and thousands of threads. “hulk.go” just uses lightweight goroutines that used only tens of threads (commonly Golang runtime started one thread for CPU core and several service threads). This architecture allows Golang version better consume resources and got much higher connection pool on the same hardware than Python version can [9].

This repository [9] has Docker file which makes using the tool quite easy and straight forward:

```
$ sudo apt install docker.io
$ git clone https://github.com/grafov/hulk
$ cd docker/
$ sudo docker build -t hulk .
$ sudo docker run -it hulk -site http://ec2-35-183-239-72.ca-central-1.compute.amazonaws.com
```

As you can see, after installing Docker on the Ubuntu server and cloning the repository, one can simply run the Docker image and then run the HULK program.

HULK generates some nicely crafted unique Http requests, one after the another, generating a fair load on a webserver, eventually exhausting it of resources. This can be optimized much further, but as a proof of concept and generic guidance it does its job [10].

However, as the source of my research explains and demonstrates, HULK defines a certain order of request headers using the "request.add_header()" function call; the actual order of the headers at run time is dictated by the dictionary key ordering of the python implementation. This ordering is a unique fingerprint for this tool as no other legitimate web clients have this header ordering [10]. Thus, making this tool easily detectable.

Using HULK enables an adversary to fool AWSTAST engine to mark all the requests hitting the webserver as legitimate visits in mass scale. However, this tool can not fool Google Analytics Engine.

DDOSIM-Layer 7 DDoS Simulator

DDOSIM is a tool that can be used in a laboratory environment to simulate a distributed denial of service (DDOS) attack against a target server. The test showed the capacity of the server to handle application specific DDOS attacks. DDOSIM simulates several zombie hosts (having random IP addresses) which create full TCP connections to the target server. After completing the connection, DDOSIM starts the conversation with the listening application (e.g. HTTP server) [11].

DDOSIM is written in C++ and runs on Linux. Its current functionalities include:

- HTTP DDoS with valid requests
- HTTP DDoS with invalid requests (similar to a DC++ attack)
- SMTP DDoS
- TCP connection flood on random port

In order to install and use this tool, first the Debian package needs to be downloaded. Then after installing some of its dependencies, the package can be successfully compiled and installed:

```
$ sudo apt install libpcap-dev
$ wget http://launchpadlibrarian.net/1315573/libnet0-dev_1.0.2a-7_amd64.deb
$ wget http://launchpadlibrarian.net/1315572/libnet0_1.0.2a-7_amd64.deb
$ sudo dpkg -i libnet0_1.0.2a-7_amd64.deb
$ sudo dpkg -i libnet0-dev_1.0.2a-7_amd64.deb
$ git clone git@github.com:krsumeet/Attack.git
$ cd Attack/ddosim/
$ chmod +x ./configure
$ ./configure
$ make
$ sudo make install
$ host ec2-35-183-239-72.ca-central-1.compute.amazonaws.com
$ ifconfig
$ sudo ./ddosim -d 35.183.239.72 -p 80 -c 0 -w 0 -t 10 -r HTTP_VALID -i wlx00c0ca964887
```

Note that this tool does not accept a domain as input, and it needs the IP address of the target. DDOSIM also makes it easy to use multiple threads to be used in order to make the process faster. All the requests made by this tool were dropped by the webserver instantly and therefore the attack was unsuccessful.

GoldenEye HTTP Denial of Service Tool

GoldenEye is a HTTP DoS Test Tool written in python3. The attack vectors that it exploits are “HTTP Keep Alive” and “No Cache” [12]. This tool also makes the user able to use concurrent TCP sockets and pick what type of HTTP request should be made.

After installing python3, the repository can be cloned, and the script can be run easily:

```
$ git clone git@github.com:jseidl/GoldenEye.git
$ cd GoldenEye/
$ ./goldeneye.py http://ec2-35-183-239-72.ca-central-1.compute.amazonaws.com
```

Same as previous tool examined, all the requests made by GoldenEye were dropped by the webserver and made the attack unsuccessful.

Headless Chrome

Headless Chrome is a great tool for automated testing and server environments where you don't need a visible UI shell. However, an adversary can abuse this tool easily to ruin web analytics data of a target website. Puppeteer is a Node library developed by the Chrome team. It provides a high-level API to control headless (or full) Chrome. It's similar to other automated testing libraries like Phantom and NightmareJS, but it only works with the latest versions of Chrome [13].

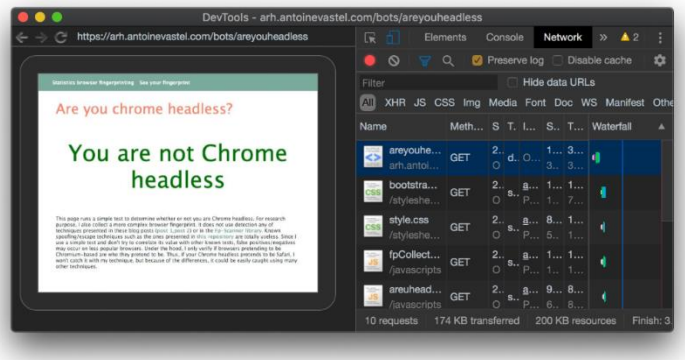
Using Puppeteer, I was able to create a minimal program which fully loads the target website and thus fakes a real user visit:

```
const puppeteer = require('puppeteer');
const TARGET = "http://ec2-35-183-239-72.ca-central-1.compute.amazonaws.com/p1.html";

function sleep(ms) {
    return new Promise(resolve => setTimeout(resolve, ms));
}

(async () => {


    for (let i = 1; i <= 100; i++) {
        const browser = await puppeteer.launch();
        const page = await browser.newPage();
        var response = await page.goto(TARGET, { waitUntil: 'networkidle0' });
        console.log(response.status());
        await page.waitForFunction('typeof gtag === "function"');
        console.log(i);
        await browser.close();
    }


})().catch((e) => {
    console.log(e);
});
```

The first couple of requests were counted as legitimate by Google Analytics, however, after repeating the request many times, it seemed that Google Analytics stopped counting the requests as legit. This pattern has been also seen in the next two headless browsers as well but using the other two tools was much easier and enabled me to show how an adversary can fool Google Analytics as well as AWSTATS.

As mentioned before, Headless Chrome makes it very easy to fool AWSTATS. One interesting observation was that by using Puppeteer, all requests agents were set to “Headless Chrome” and this suspects me that Google Analytics filters these requests quite easily.

However, there are various ways to overcome this problem. One way is to use “puppeteer-extra-plugin-stealth” to prevent detection [14].



Firefox Headless

To automate Firefox in Headless mode, one of the easiest ways is to use Node.js and Selenium [15]. Using “selenium-webdriver” and “geckodriver”, I was able to create a small program that demonstrates the abelites of this tool in fooling the analytics engines:

```
const TARGET = "http://ec2-35-183-239-72.ca-central-1.compute.amazonaws.com/p1.html";
var webdriver = require('selenium-webdriver'),
    By = webdriver.By,
    until = webdriver.until;

var firefox = require('selenium-webdriver/firefox');

var options = new firefox.Options();
options.addArguments("-headless");

(async () => {
    for (let i = 1; i <= 20; i++) {
        var driver = new webdriver.Builder()
            .forBrowser('firefox')
            .setFirefoxOptions(options)
            .build();
        await driver.get(TARGET);
        await driver.quit();
        console.log(i);
    }
})();
```

After making “geckodriver” available on the PATH variable:

```
$ wget https://github.com/mozilla/geckodriver/releases/download/v0.24.0/geckodriver-v0.24.0-linux64.tar.gz
$ tar -xvzf geckodriver*
$ chmod +x geckodriver
$ export PATH=$PATH:/path-to-extracted-file/.
```

I was able to run my code and observe an increment in the number of online users reported by Google Analytics followed by the number of page visits. Once again, this code easily fools AWSTATS. The problem with both of Headless Chrome and Firefox Headless was that they needed too much extra work to make a fast program and they don’t have easy to find documentation and working examples. The next tool, however, makes the situation very much suitable for a successful attack in large scale.

HtmlUnit

A java GUI-Less browser, which allows high-level manipulation of web pages, such as filling forms and clicking links; just “getPage(url)”, find a hyperlink, “click()” and you have all the HTML, JavaScript, and Ajax automatically processed. HtmlUnit provides excellent JavaScript support, simulating the behavior of the configured browser (Chrome, Firefox or Internet Explorer). It uses the Rhino JavaScript engine for the core language (plus workarounds for some Rhino bugs) and provides the implementation for the objects specific to execution in a browser [16].

This tool is the most robust and easy to adopt tool among all the tools that have been examined. HtmlUnit is well documented and has a great community around it. With one simple Maven project, I was able to completely fool both engines in mass scale (the full Maven project is available on the research repository, the following only shows the main logic):

```
final WebClient webClient = new WebClient(BrowserVersion.FIREFOX_68);
Random rand = new Random();
HtmlPage page1 = webClient.getPage(TARGET + "/p" + (rand.nextInt(3) + 1) + ".html");
// blocks till GA tag is fully initialized
page1.executeJavaScript("while(dataLayer[1][1] != 'UA-157513426-1') {}; var x = true; x;");
HtmlAnchor htmlAnchor = page1.getAnchorByHref("./index.html");
Thread.sleep(1000); // To make the GA confused about online users
HtmlPage page2 = htmlAnchor.click();
page2.executeJavaScript("while(dataLayer[1][1] != 'UA-157513426-1') {}; var x = true; x;");
webClient.close();
```

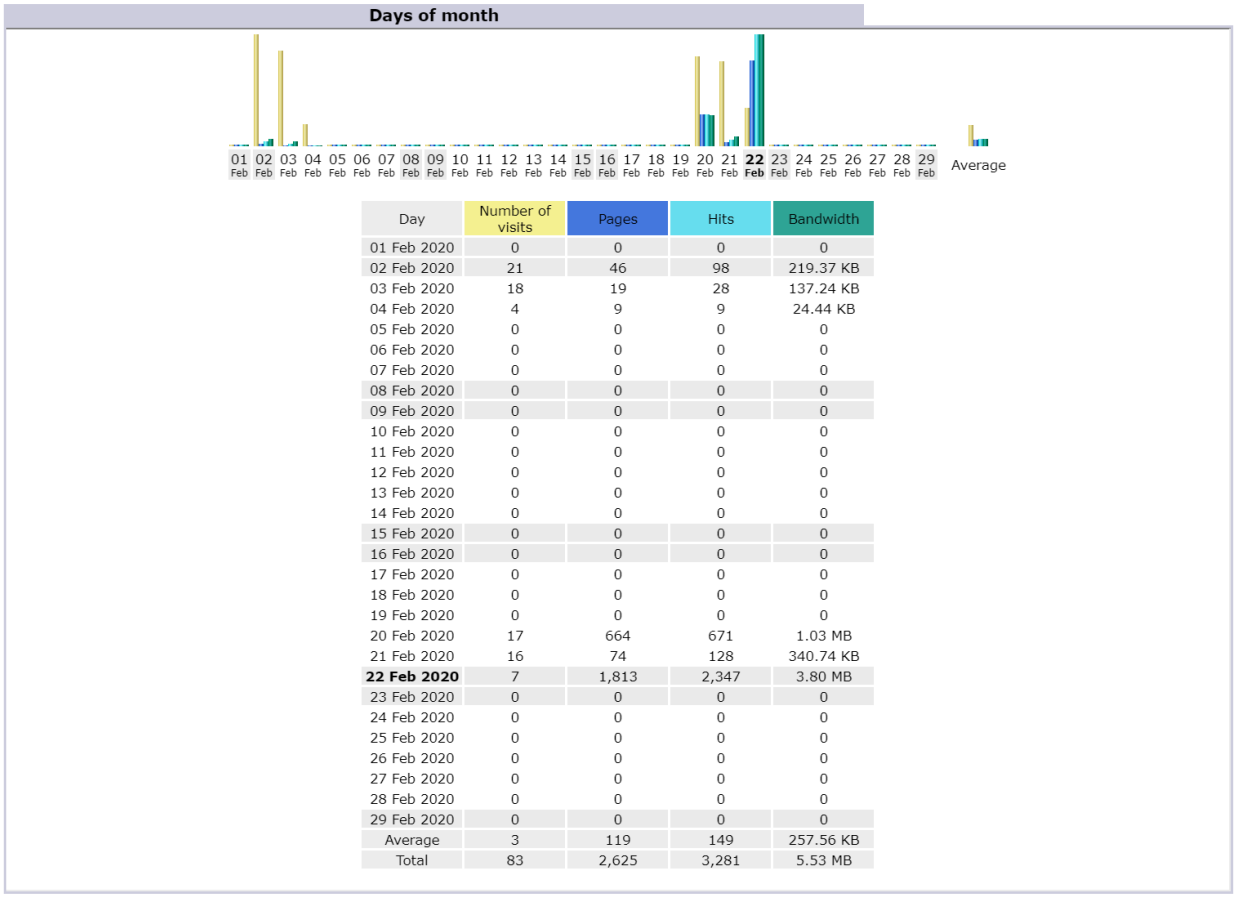
Running the above segment in a “Runnable” java class, made me able to achieve high speed and mass scale attack on both AWSTATS and Google Analytics. Note that the code tries to fake a real user behaviour by waiting for the page to fully load and execute the JavaScript, waiting a little bit more (faking a real user trying to find a link) and then clicking on a link and again waiting for the next page to fully load. The requests are also being made on randomly chosen pages.

The line “`executeJavaScript("while(dataLayer[1][1] != 'UA-157513426-1') {}; var x = true; x;")`” waits for the browser and makes sure that the Google Analytics key 'UA-157513426-1' has been registered in the JavaScript function that initializes Google Analytics on the client side. Also note that by running this code on different threads, we don’t need to wait for each of the browsers to finish and we can launch as many browsers as we want at the same time.

The very first line of this segment also runs on different threads, opening a new browser for each thread. By doing so, I have reduced missing attack requests to zero.

The following images show the result of the two engines after attacking via HtmlUnit:

AWSTATS:

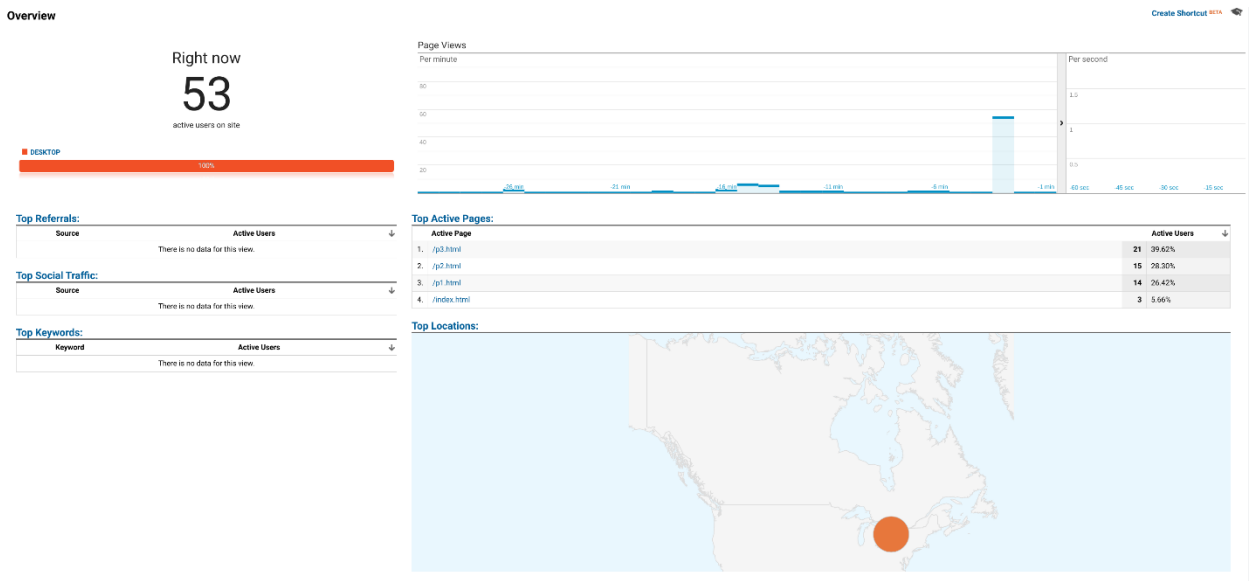


Days of month

01 Feb 02 Feb 03 Feb 04 Feb 05 Feb 06 Feb 07 Feb 08 Feb 09 Feb 10 Feb 11 Feb 12 Feb 13 Feb 14 Feb 15 Feb 16 Feb 17 Feb 18 Feb 19 Feb 20 Feb 21 Feb **22 Feb** 23 Feb 24 Feb 25 Feb 26 Feb 27 Feb 28 Feb 29 Feb Average

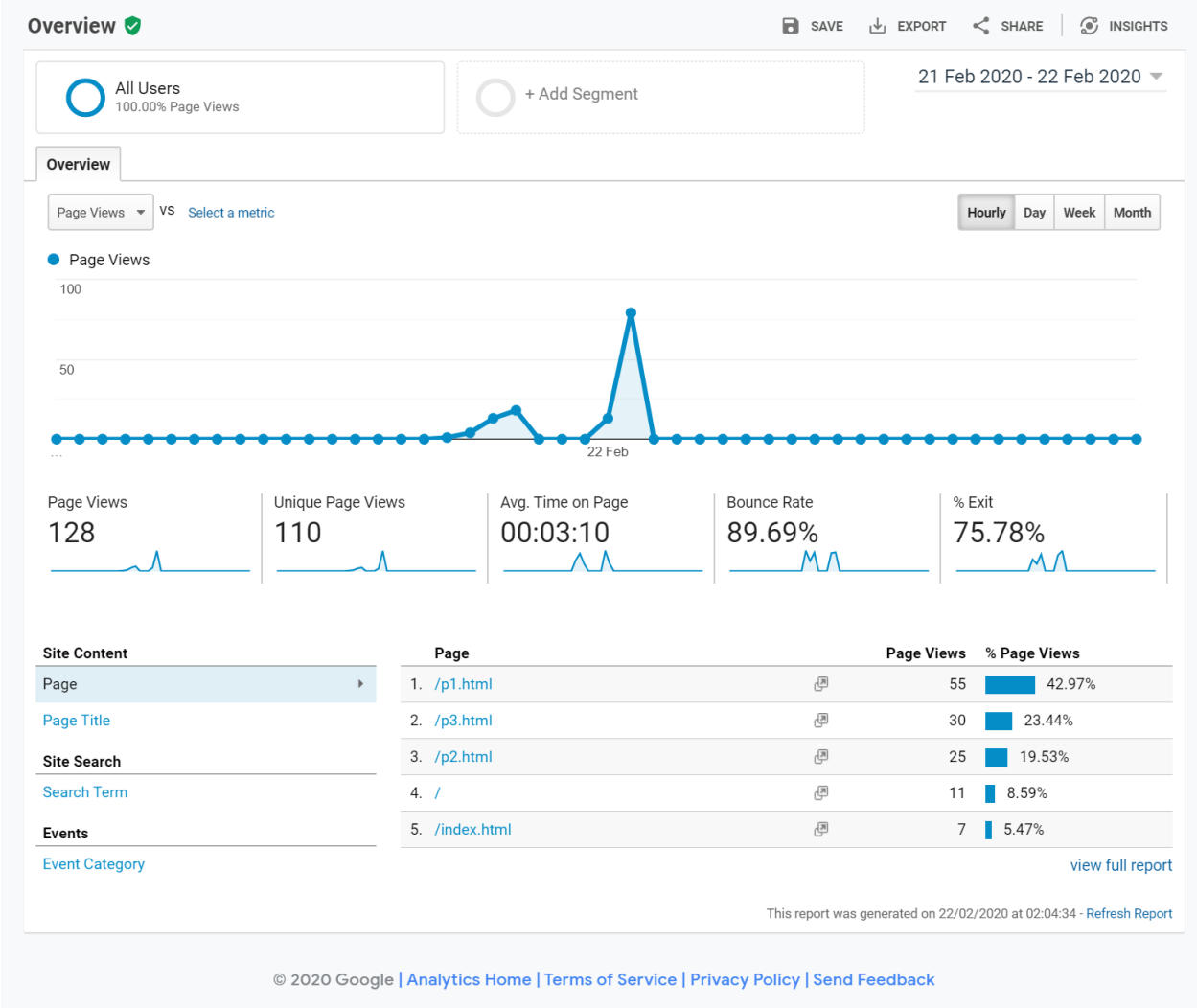
Day	Number of visits	Pages	Hits	Bandwidth
01 Feb 2020	0	0	0	0
02 Feb 2020	21	46	98	219.37 KB
03 Feb 2020	18	19	28	137.24 KB
04 Feb 2020	4	9	9	24.44 KB
05 Feb 2020	0	0	0	0
06 Feb 2020	0	0	0	0
07 Feb 2020	0	0	0	0
08 Feb 2020	0	0	0	0
09 Feb 2020	0	0	0	0
10 Feb 2020	0	0	0	0
11 Feb 2020	0	0	0	0
12 Feb 2020	0	0	0	0
13 Feb 2020	0	0	0	0
14 Feb 2020	0	0	0	0
15 Feb 2020	0	0	0	0
16 Feb 2020	0	0	0	0
17 Feb 2020	0	0	0	0
18 Feb 2020	0	0	0	0
19 Feb 2020	0	0	0	0
20 Feb 2020	17	664	671	1.03 MB
21 Feb 2020	16	74	128	340.74 KB
22 Feb 2020	7	1,813	2,347	3.80 MB
23 Feb 2020	0	0	0	0
24 Feb 2020	0	0	0	0
25 Feb 2020	0	0	0	0
26 Feb 2020	0	0	0	0
27 Feb 2020	0	0	0	0
28 Feb 2020	0	0	0	0
29 Feb 2020	0	0	0	0
Average	3	119	149	257.56 KB
Total	83	2,625	3,281	5.53 MB

Google Analytics:



By putting the threads in sleep, I was also able to increase the average time on the page:

Google Analytics:

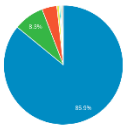


Loki

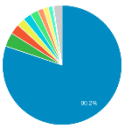
To complete our demonstration, we have developed a tool based on HtmlUnit to further show Google Analytics vulnerability to fake requests. Loki is a multi threaded Maven project written in Java and uses the results of our experiment phase. We have tried to make Loki flexible enough to later be used by other researchers who may want to do further development in this area.

In the Java project resources folder, there are couple of different text files which are being used as lists of different values for different parameters to be fed into the HtmlUnit browsers while making the fake requests. Each Loki thread opens a brand-new browser which guarantees the Google Analytics to count the browser as a brand-new user. While initiating the HtmlUnit browser, Loki picks a random browser agent from the provided list (almost 9K famous agents have been listed).

Operating System	Users	Users	contribution to total: Users
	2,229 % of Total: 100.54% (2,217)	2,229 % of Total: 100.54% (2,217)	
1. Windows	1,915	85.91%	
2. Linux	184	8.25%	
3. Macintosh	90	4.04%	
4. (not set)	14	0.63%	
5. Solaris	7	0.31%	
6. Android	6	0.27%	
7. iOS	5	0.22%	
8. SunOS	3	0.13%	
9. FreeBSD	2	0.09%	
10. OpenBSD	2	0.09%	



Browser	Users	Users	contribution to total: Users
	2,217 % of Total: 99.73% (2,223)	2,217 % of Total: 99.73% (2,223)	
1. Firefox	1,777	80.15%	
2. Mozilla	84	3.79%	
3. Chrome	61	2.75%	
4. Internet Explorer	49	2.21%	
5. Opera	49	2.21%	
6. Safari	48	2.17%	
7. Netscape	34	1.53%	
8. SeaMonkey	32	1.44%	
9. Konqueror	25	1.13%	
10. Iron	7	0.32%	



Next, it picks a random path from the provided “paths” list. Path list is the list of the paths of the targeted domain that we want the traffic to be directed at. Loki also sets some random keywords picked from the “keywords” list to fake the referee keyword of the requests (for now, we are setting the referee to Google.com). After opening the page and halting for a while, Loki picks a random link from the page, clicks on it and waits until the requested page has been completely initialised. The list of links on the page which we want to target is also provided in the “anchors” file and will be picked randomly.


Right now
201
active users on site

DESKTOP OTHER (1.99%)

Top Referrals:

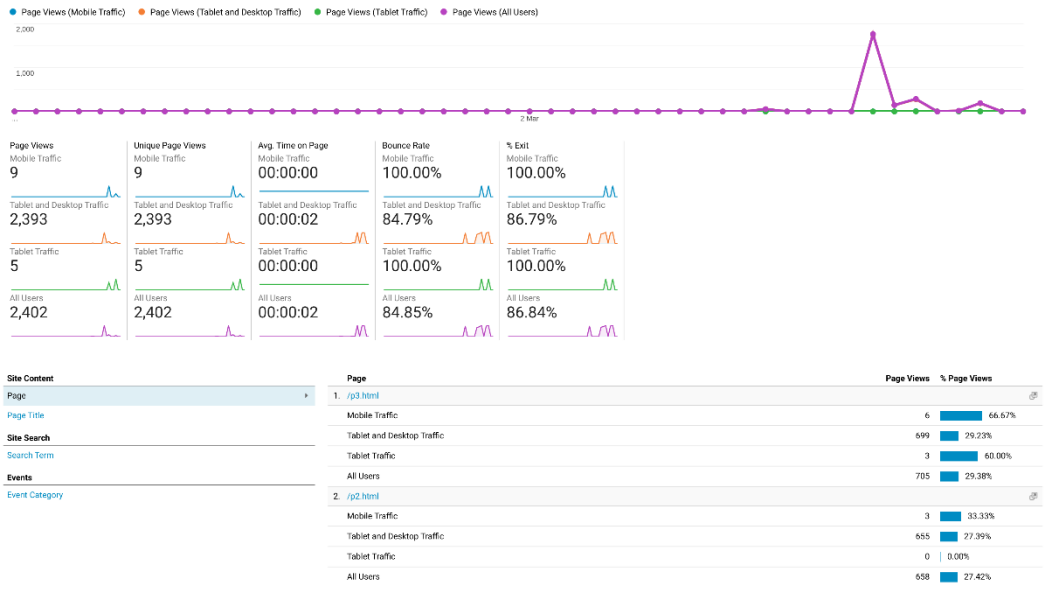
Source	Active Users
There is no data for this view.	

Top Social Traffic:

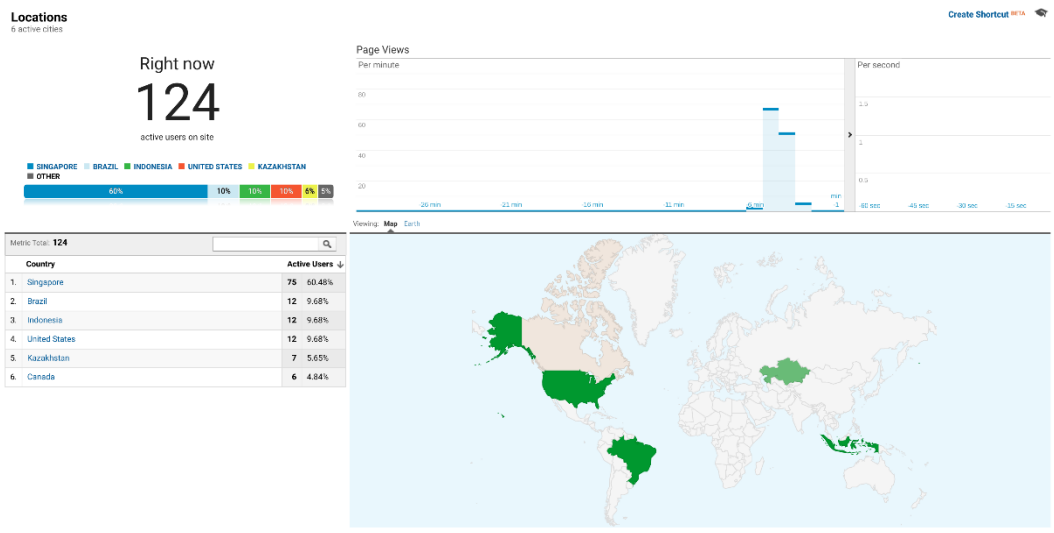
Source	Active Users
There is no data for this view.	

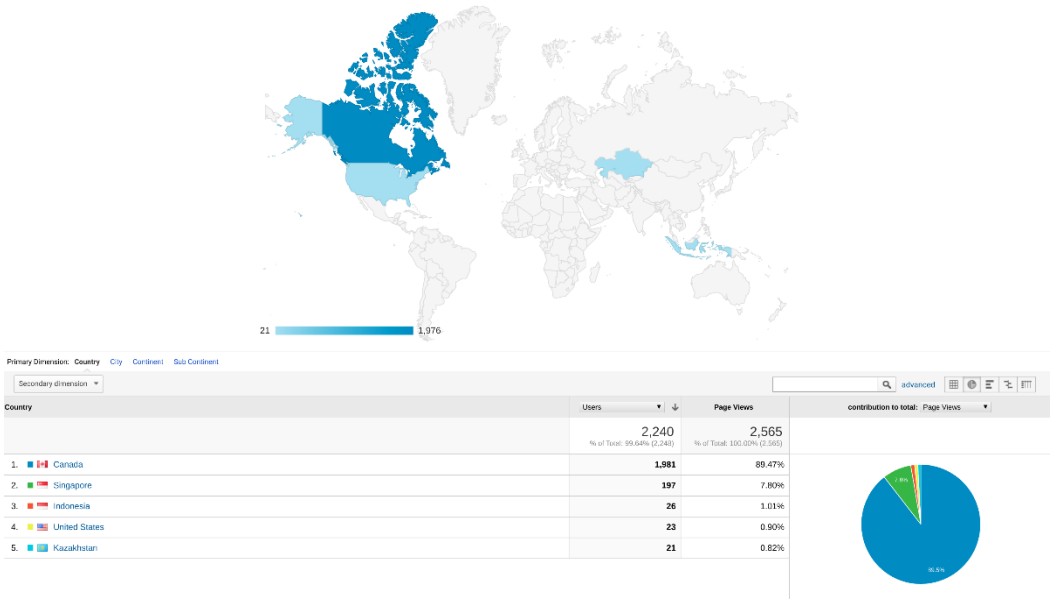
Top Keywords:

	Keyword	Active Users
1.	loki 4480	26
2.	testing tool	23
3.	york university	23
4.	course project	22
5.	saaskaki	22
6.	lab	21
7.	ali niaki	18
8.	ieee paper	18
9.	test google analytics	15
10.	computer security	12



To make the requests more legitimate and prevent the DoS mechanisms, we have set the HtmlUnit to use Tor's SOCKS V5 proxy. To be able to use Tor, the hosting machine simply needs to run the Tor browser and Loki picks up the proxy on port 9050. If the proxy is not available on the system which is running the program, Loki falls back to the default mode and uses the public IP adders of the host machine (not recommended). Tor has a nice feature which allows us to define the amount of time that each generated identity is valid. By setting the "MaxCircuitDirtiness" parameter of Tor's browser to one minute, we can benefit the most from Loki's request distribution feature.





Ultimately, to further fake a real-world scenario, Loki distributes requests by putting random delays between them and overall, tries to increase the number of requests in a logarithmic pattern. Let's assume the number of the fake requests we want to make is "n", the number of threads is "t" and the number of requests made at each step is "r". Loki makes sure that the overall number of requests would not be more than "n". In each step, "MAX" is the maximum value of "r" and equals to "log2(n^3) * t". The minimum value of "r" is 75% of "MAX". This way, overtime, the number requests at each step increases logarithmically, but also has a somewhat random increasing pattern.

After each thread made "r" requests, the thread will wait for a random duration of time between "minDelay" and "maxDelay". As time goes on, the time that each thread starts to make the fake requests will not be aligned with any other threads which will confuse the Google Analytics even more. Combined with a short value set to Tor's "MaxCircuitDirtiness", our requests would not only be distributed in a timely fashion, but they will also be randomly distributed geographically.

All the variables defined above with some extra configuration (Targeted Google Analytics Token), are all adjustable via the Config instance that is being passed to Loki.

Defense Mechanisms again Loki

This project would not be completed without suggesting some defence mechanisms to combat MWAP attacks. Loki and tools alike, can not fully replicate the human interactions with webpages. The main area that they lack copying humans, is the mouse movement.

Although by leveraging headless browsers we can trigger mouse events, it will not be easy to fake a human like mouse movement completely. JavaScript engine in headless browsers make an adversary able to write a simple script which fakes mouse movement but by running adaptive risk analysis, we will be able to filter out these "simple" scripts. Google has recently developed and released reCaptcha v3 which does the exact same thing.

The reCAPTCHA v3 system is Google's attempt to remove the user interaction from the equation, using a variety of different signals in the background that combine to give site owners a score that assesses how likely a user is to be a bot. reCAPTCHA v3 runs adaptive risk analysis in the background to alert you of suspicious traffic while letting your human users enjoy a frictionless experience on your site [17].

Combining the result of such analysis with the current approaches can help Google Analytics to detect fake requests being made my simple tools such as Loki. Even to a degree, more advanced tools which blindly try to fake mouse movement or other interactions with Webpages would be detectable.

However, more advanced tools that use Artificial Intelligence to fake human behaviour would still not be detectable. Researchers have shown that Reinforcement Learning (RL) methodologies [18] are able to bypass the currently available adaptive risk analysis. On the other hand, other research shows that using the same method and by leveraging Artificial Intelligence techniques, it is possible to better detect more advanced AI generated interactions even on mobile devices [19].

Fully detecting MWAP attacks and automated bot generated web requests is not an easy task and further research is needed. However, it seems that the solution is probable to be found by leveraging analysis made by Artificial Intelligence software.

References

[1] N. Vlajic, S. Wilson Chow, X. Charles, "Malwareless Web-Analytics Pollution (MWAP): A Very Simple Yet Invincible Attack", 20th IEEE Conference on Business Informatics (CBI), Vienna, Jun 2018.

[2] "Meet the Four Generations of Bots",
Available: https://blog.radware.com/security/2019/09/meet-the-four-generations-of-bots/

[3] "Bot detection & Bot protection: how to identify bot traffic to your website, mobile app & API?",
Available: https://datadome.co/bot-management-protection/bot-detection-how-to-identify-bot-traffic-to-your-website/

[4] "Best DoS Attacks and Free DoS Attacking Tools (for 2019)",
Available: https://resources.infosecinstitute.com/dos-attacks-free-dos-attacking-tools/

[5] "DoS attacks: What are the popular DoS attacking tools?",
Available: https://www.greycampus.com/blog/information-security/dos-attacks-tools-and-protection

[6] "Popular Headless Browsers for Web Testing",
Available: https://www.keycdn.com/blog/headless-browsers

[7] "What is Headless Browser? Is Headless Browser Good for Web Scraping?",
Available: https://www.proxyrack.com/what-is-headless-browser-is-headless-browser-is-good-for-web-scraping/

[8] "The new DDoS Landscape",
Available: https://readyspace.com.hk/the-new-ddos-landscape/

[9] https://github.com/grafov/hulk

[10] https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/hulk-vs-thor-application-dos-smackdown/
[11] https://stormsecurity.wordpress.com/2009/03/03/application-layer-ddos-simulator/
[12] github.com:jseidl/GoldenEye.git
[13] https://developers.google.com/web/updates/2017/04/headless-chrome
[14] https://www.npmjs.com/package/puppeteer-extra-plugin-stealth
[15] https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Headless_mode
[16] http://htmlunit.sourceforge.net/
[17] https://duo.com/decipher/google-recaptcha-v3-finds-bots-with-no-user-interaction

[18] Akrout, I.; Feriani, A.; and Akrout. M. 2019. Hacking Google reCAPTCHA v3 using Reinforcement Learning. arXiv:1903.01003

[19] Acien, A., Morales, A., Fiérrez, J., Vera-Rodríguez, R., & Bartolome, I. (2020). Be-CAPTCHA: Detecting Human Behavior in Smartphone Interaction using Multiple Inbuilt Sensors. ArXiv, abs/2002.00918.