Anomaly-Based Intrusion Detection for Web Servers

A Technical Report
presented to the faculty of the
School of Engineering and Applied Science
University of Virginia

by

Roman Bohuk
May 4, 2020

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments.

Signed: _____ Date: _____
Roman Bohuk

Approved: _____ Date: _____
Jack Davidson, Department of Computer Science

Table of Contents

Glossary of Technical Terms

Cookie – A small piece of data associated with a website and stored locally in a browser. It is generally sent by the browser with every request to the web server. Its main uses include session management, activity tracking, and storage of settings and preferences.

GET Request – A method of requesting data from a web server by encoding data in the URL.

Hypertext Transfer Protocol (HTTP) – A high-level internet communication protocol designed to support data transfer and communication between web servers and web clients.

Hypertext Markup Language (HTML) – A language used to define the structure and the content of a webpage. When accessing a website, the browser retrieves the HTML page from the web server and renders it visually to the user.

HTTP Header – Information passed along with the HTTP request or response containing some additional details about the message such as its content type, a browser or a server identification tag, and connection details.

Intrusion Detection System – A device or an application that monitors traffic to identify malicious activity

JavaScript – A client-side programming language commonly used in web development. Unlike a server-side language, JavaScript runs in the client's web browser. The language allows to augment the user experience by adding dynamic and interactive functionalities to the site.

Man-in-the-Middle (MITM) Attack – An attack executed by intercepting and possibly manipulating a communication between two systems.

Parameter – a named variable passed in an HTTP request

Proxy – an application that sits between a web client and a web server that intercepts requests from the client and forwards them to the server.

POST Request – A method of sending data to a web server by including it in the message body.

Request – Messages initiated by the client (such as a browser) to a server. The messages sent by the server as replies are called responses.

Session – A method for web servers to uniquely identify users of the application and track their interactions in between separate requests.

Signature – A pattern unique to a specific exploit or a piece of malware that is used to detect other instances of the same attack.

Status Code – A numeric indicator describing whether an HTTP request completed successfully in a specific manner or encountered an error.

Uniform Resource Locator (URL) – an address of a specific webpage or a file on the internet. It generally consists of the protocol prefix, the domain name of the site, and the path to the resource.

Web Client – The party that initiates the communication with a web server via HTTP requests. It is usually a web browser on a user's machine.

Web Server – A software application that listens for HTTP requests and fulfils them. It can also refer to a dedicated computer that stores and runs web server software.

Sources include MDN Web Docs (https://developer.mozilla.org/), OWASP Foundation (https://owasp.org/), TehcTerms (https://techterms.com/definition/javascript), and Webopedia (https://www.webopedia.com/)

Abstract

Web applications are ubiquitous. They store and process sensitive user data and serve as entry points to corporate networks. Despite increased emphasis on security, vulnerabilities in web applications are still common. An explosion of traffic volume has created a need for automated means to stop attacks. Rule-based intrusion detection systems require large, up-to-date signature databases, and they are powerless against new attacks. A non-intrusive anomaly detection system for intercepting and identifying malicious HTTP traffic in real-time is proposed. The tool uses a reverse HTTP proxy to intercept traffic between the client and the server to create a baseline profile of benign traffic. The profile is created by observing various characteristics of web requests and their expected responses. The tool then measures and compares new requests to the baseline to detect and stop attacks without the use of signatures.

I. Introduction

According to the Internet Society, organizations have collectively lost over $45 billion due to breaches of confidentiality, integrity, and availability of their data in 2018. To gain unauthorized access to a network, a malicious actor has to examine the available attack surface to find a viable vector to get in, and web applications are frequently targeted first. An attacker can manually craft a malicious request to perform an unintended action on the web server in an attempt to steal data or disrupt the service. The more complex a website is, the more likely it is for a hacker to find a flaw.

A study by Positive Technologies in 2018 found that 19 percent of tested web applications have vulnerabilities that could enable an attacker to take control of the server. The average number of vulnerabilities jumped dramatically from the year prior as well (Positive Technologies, 2018). These findings are surprising especially since the majority of those applications store and process user data. The number of web applications increases as internet usage grows and companies ramp up their infrastructure. This causes an explosion of traffic volume that the security teams have to analyze and process. Combined with a workforce shortage (in almost every position within cybersecurity), these cyber threats necessitate more automated solutions (Crumpler & Lewis, 2019).

One method to detect these exploits programmatically is to implement functions that inspect each request for signs of known malicious activity. This approach is known as rule-based detection, and it requires extensive signature databases that must stay up-to-date. Thus, it is powerless against new attacks and zero-day exploits. The alternative is anomaly detection. The security system would observe benign traffic to form statistical models and raise flags if it sees something out of the ordinary.

A web request passes through several layers of server logic that are all part of the same web server. From a high-level point of view, the network card on the computer first receives the binary message and sends it to the operating system, which then routes the message to a web server process running software such as Apache, Nginx, or Python Flask. The process then decrypts the message if necessary, parses it, and finds the requested resource or a script. Static resources such as images, JavaScript files, CSS stylesheets, and other forms of documents and media can be sent directly back to the client. Dynamic requests may require the web server to execute custom code and return the output. This custom code might perform simple calculations, parse data, or even interact with other systems like database servers. The attacker can target and exploit any of these layers of the server. The OWASP Foundation maintains a comprehensive list of web attacks that can be used as a reference.

There exists a large variety of exploits that target different parts of the web server, and there are multiple ways for attackers to inject payloads in requests. In some cases, anomaly detection is not necessary to stop certain kinds of attacks. For example, some exploits work by bypassing server logic in unexpected ways. A bug known as Heartbleed allowed attackers to steal sensitive data from servers using OpenSSL by sending an information header that was inconsistent with the request (Fruhlinger, 2017). A manually crafted message could claim that it had 50KB of data but only contain 5KB. A special kind of request would prompt the server to echo back the requested 5KB of data but read out of bounds and send the full 50KB back. These kinds of attacks can be mitigated simply by filtering out invalid requests that do not strictly conform to protocol standards. At the application layer of the OSI model where we parse the HTTP message itself, some of these exploits are no longer relevant.

This is what an example request to google.com looks like:

Request Headers:

:authority: www.google.com
:method: GET
:path: /search?q=test
:scheme: https
accept: */*
accept-encoding: gzip, deflate, br
accept-language: en-US,en;q=0.7
cache-control: no-cache
referer: https://www.google.com/
user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36

Body:

<empty>

In this example, we are sending a GET request to the `/search` page of `google.com` with `q=test` as the query parameter. The HTTP request above shows what the request would look like. Note that the body is empty because the query is sent via the URL. If we were to send a POST request, the parameters would be sent as part of the body. The response to this request has a similar structure, except the page content is sent back in the body.

The goal of this project was to create and evaluate a web application intrusion detection system while exploring anomaly detection as an approach for web application security. Attacks targeting the hardware, the operating system, or the web server software were not in scope. The requests were intercepted at the HTTP protocol level. These attacks are meant to target the web application code itself. An optimal system and a possible future iteration of this project would combine rule-based filters with anomaly detection to identify even more attacks.

II. Related Work

The concept of anomaly detection in security has existed for at least 30 years, but most of the research has been done in general network security rather than web security specifically (Winkler & Page, 1989; Cho & Cha, 2004; Reunanen et al., 2019).

In 2003, Kruegel and Vigna claimed to present the "first anomaly detection system specifically tailored to the detection of web-based attacks." They used six statistical models to analyze the character sequences in the request parameters and to come up with an anomaly score. In 2004, Wang and Stolfo attempted to solve a similar problem, but used different techniques to analyze the character sequences. Their research was limited however, in that it only analyzed a handful of GET parameters without looking at any other request data.

Since then, multiple papers have been published focusing on various distinct parts of a web request and increasingly utilizing machine learning models (Wen, Guo, & Yu, 2013; Zhang, et al., 2015; Zhang, Lu, & Xu, 2017).

III. Materials and Methodology

A. Vulnerable Application

A vulnerable website was created for testing. To simulate a realistic application, an existing content management system was taken as a base. It included a functioning login page, dozens of endpoints using both POST and GET, plenty of static content, and it was connected to a MySQL database. The following vulnerabilities were introduced:

- *Command Injection* was embedded a tool designed to perform whois, nslookup, and dig queries on domain names did not validate user input server-side allowing the attacker to run bash commands on the server
- *SQL injection* was possible through a vulnerable search interface enabling the attacker to inject malicious queries and steal data from the entire database
- *Cross-site scripting* on a misconfigured blog allowed creation of posts with XSS payloads that would then be displayed back to the users
- *Directory traversal* vulnerability in a PDF library was possible because the documents were fetched by filename without validating the file type
- *Buffer overflow* vulnerability was introduced in a script, so that it crashed and exposed sensitive data when the input length exceeded a threshold
- *Improper data validation* allowed a login page that used PHP's vulnerable strcmp function to authenticate users without a password. This function would bypass the login check if the API received an array instead of a string for a password
- *Sensitive data exposure* was possible by accessing the `.git` folder allowing the attacker to retrieve full information about the website's GIT repository and commits. A similar `.ftpconfig` file was also accessible.

B. Data Collection

Much previous research to identify malicious activity was limited to analysis of web server logs, which contained only minimal data. By default, such log data only includes information about the HTTP method, the requested URL, and the status code. Information such as cookies, headers, duration between request and response, and response data type is lost. Even the POST parameters and data used to send information to the server are not preserved. Frequently, this kind of analysis is also platform and web-server specific.

To avoid these issues and go a bit further, a reverse proxy was used to intercept traffic between the client and the server on-the-fly. Python mitmproxy module was a perfect fit for this, as it not only intercepts requests and responses but also supports analysis and modification of them in real-time with custom scripts. The proxy stood in between the client and the server, and it was setup to forward any requests sent to the vulnerable application hosted in a Docker container locally. This kind of setup with a proxy enables "hands-free" deployment since no changes had to be made to the web server configuration or the application code. There is no difference in what the user sees as well. Minor configuration changes will need to be done to enable SSL, but the proxy will automatically decrypt and encrypt the traffic as needed to make sure the data can be recorded.

Having a proxy allows us to collect much more data. To aid in analysis and creation of a profile later on, the requests were stored in json file. The following details about each transaction were recorded:

- *Request method, request path, and response status code* – these three metrics will be used together to separate every request into unique "endpoints" or bins. Analysis will be performed only on alike requests where all three of those values match.

- *Request duration* – time between the request start time and the response end time. Anomalously quick or long requests can indicate something malicious.
- *Request headers and request cookies* – many web applications implement custom session management modules that can be vulnerable to injected payloads delivered via the cookies. The request headers are mostly used by web servers, but some custom logging mechanisms rely on them. Cookies are included in the request headers, so storing them separately is a bit redundant, but it makes it easier to parse.
- *Request GET parameters* – data sent to the server in the URL. This is frequently used to retrieve certain pages by an identifier or for small queries. There is a limit to the URL length.
- *Request POST parameters* – data sent to the server in the message body. This mechanism is often used to submit data entered in forms or when many parameters are used. The proxy merges the multipart data with the url-encoded data.
- *Request content* – this is the same POST request data but encoded in the message body as a single string. This content can be useful for detect attempts to pass parameters with mismatching datatypes
- *Response content size and content type* – even though the response content is intercepted by the proxy, it is not stored in its entirety to avoid having to store a large json file. Instead, only the response content length is stored together with its MIME-type as determined by the Linux `file` command.

The traffic was generated manually by using the various features of the vulnerable website on behalf of different users with different permission levels. Traffic generation was done in two phases. First, only benign traffic was generated to produce a series of baseline requests. A

total of 10,418 requests were recorded. In the second phase, a new dataset of traffic was generated that contained both benign and malicious traffic. The malicious traffic was labeled manually with a help of an interactive tool that would summarize the request and propose label options. The second dataset contained 812 requests, 63 of which were malicious.

C. Profile Creation

The profile was generated in two phases. First, the data was pre-processed to identify the different endpoints and separate all requests into their own bins. An object consisting of the request's path, request method, and response status code was hashed to get a unique endpoint ID. A unique profile was created for each of those endpoints based on the other request with the same ID.

Separating by endpoints allows us reduce variance in the calculations that would otherwise not be necessary. Even though the same script might get requested twice, one of them might return a success status code 200, while the other one will redirect the user to the login page with a status code 304. The parameters could have been completely different for the two types of requests, so it does not make sense to merge them. Similarly, many webpages accept both GET and POST requests and perform different actions based on the request method. It makes sense to treat these as different endpoints. Using this method, a total of 94 unique endpoints were identified. This automatic detection of endpoints allows this tool to be applied to almost any service or system without having to manually specify what is located where.

Any endpoints not identified during the profiling phase will be treated as an anomaly by the detection script. This method did not have any issues with the vulnerable website described earlier, but it could have been a limitation if some data was embedded in the URL without the

use of a GET parameter (such as Python Flask's variable sections of the URL) because the code would have treated it as a separate endpoint.

We can go a step further and split those profiles more granularly by also including the specific GET and POST parameters in the hash because some pages may run different code based on the input. For example, there might be a parameter called "action" that accepts a variety of function names. Each of those might require additional GET or POST parameters. This would not work well with user-facing API's with a variety of parameters, many of which are optional. However, regular websites and JavaScript calls tend to always use the same parameters that are often in the same order, so this method would work well there.

The exception to this rule is that all cookies for all requests will be processed together. There is generally no difference in them from page to page. In addition, even though the headers are recorded in the proxy, they will be ignored for the scope of this project, as there is usually no data that is passed to the application directly.

In the second phase, expected values were calculated for each metric. The means and standard deviations were calculated for expected lengths of each request parameter value and response content. The parameter fields for all GET requests, POST requests, and Cookies were analyzed to calculate the probability of each one of them occurring in any given request. Also, the character distributions for all parameter values were calculated.

Two types of character distributions were calculated: one for every ASCII character from 0 to 255 and one with only 7 categories (uppercase letters, lowercase letters, digits, whitespace, non-printable characters, punctuation, other special characters). The expected frequency for each character/category and a standard deviation of all individual frequencies for that attribute were calculated and stored in the profile.

C. Intrusion Detection and Prevention

Finally, a combination of the scripts in parts A and B was used to analyze new requests. An mitmproxy module was written to do this both in real-time and with pre-recorded json data. In live mode, the requests were denied to the client with an "Access Denied" message if the request anomaly metric passed a threshold.

For metrics such as response content length that had a mean and a standard deviation, the Chebyshev inequality was used to calculate the upper bound on the probability of the difference between the length of a valid request and the mean being greater the distance between the length and the mean (Kruegel & Vigna, 2003; Ibe, 2013). We can calculate the anomaly score $a(n)$ by subtracting that probability from 1.

$$a(n) = 1 - \frac{\sigma^2}{(n-\mu)^2}$$

As proposed by Wang and Stolfo, a simplified Mahalanobis distance $d(n)$ was used to calculate how far the given character distribution was away from the baseline. In other words, it calculates the sum of the z-scores of each character group. In our case, *n* is either 255 (for ASCII values) or 7 (for character groups):

$$d(m) = \sum_{i=0}^{n-1} \frac{|m_i - \mu_i|}{\sigma_i}$$

Finally, the simple probability multiplication rule was used to calculate the anomaly score $b(n)$ of a specific combination of provided parameters. For example, if an endpoint expected 3 parameters `a`, `b`, and `c`, with probabilities of occurrence being 0.9, 0.7, and 0.2 respectively. If only the parameter `a` was actually sent, the anomaly score for this request would be $1 - \big((0.9) * (1 - 0.7) * (1 - 0.2)\big) = 0.784$.

IV. Results

Each request was compared to the baseline profile on a variety of metrics, some of which turned out to be more effective than others at detecting anomalies. All of the plots below depict benign requests in blue and malicious request in red. A plot of an effective anomaly metric would reveal a clear visual separation between red and blue dots.

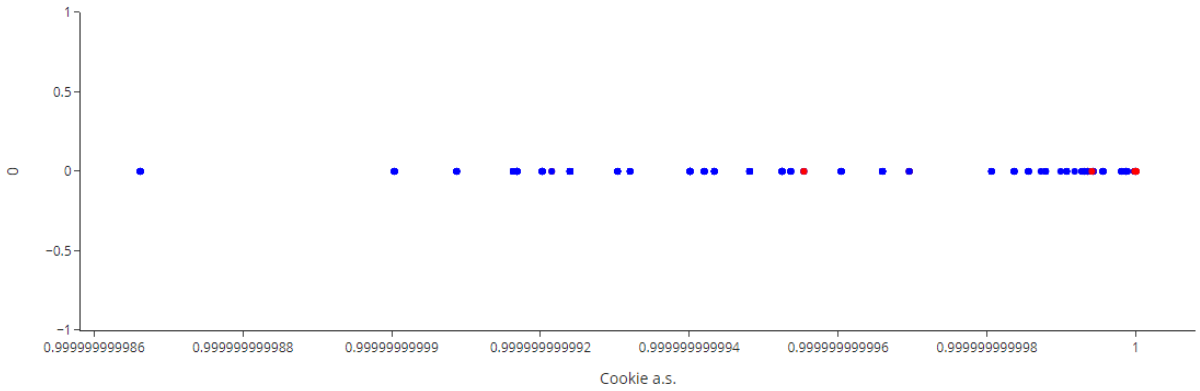

Figure 1: Cookie occurrence anomaly score

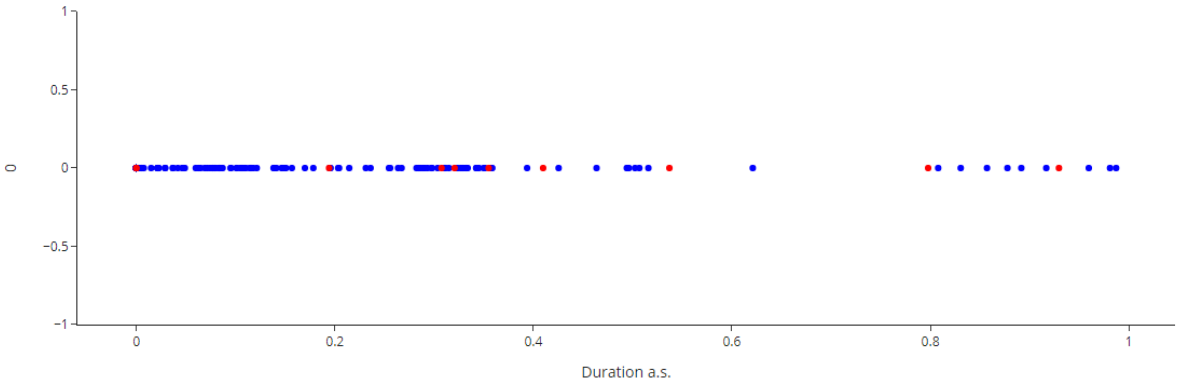

Figure 2: Request duration anomaly score

Figures 1 and 2 demonstrate the two metrics that have been the least effective at detecting malicious traffic. The anomaly scores for the cookies ranged from 0.99999999998662 to 1.0. This result can be explained by a few different factors. The most obvious explanation is that in the collected data, there were no malicious requests that were injected via the cookies. Also,

unlike any other metric that used the probability multiplication rule, the tool identified 34 unique cookies, each of which had about a 50% chance of occurrence due to the differences in the users that were used to test the system and the different browsers. This means that regardless of which cookies were provided by the browser, the anomaly score calculated using the multiplication rule would have always been very close to $1 - 0.5^{34} \approx 1$. A different metric should be used to analyze request characteristics such as these. The lack of coherent results from the duration anomaly score (calculated using the Chebyshev inequality) occurs because of the limitations of mitmproxy. It can only handle a very small amount of traffic at a time, throttling connections at random, and different datasets imposed different loads on the server.

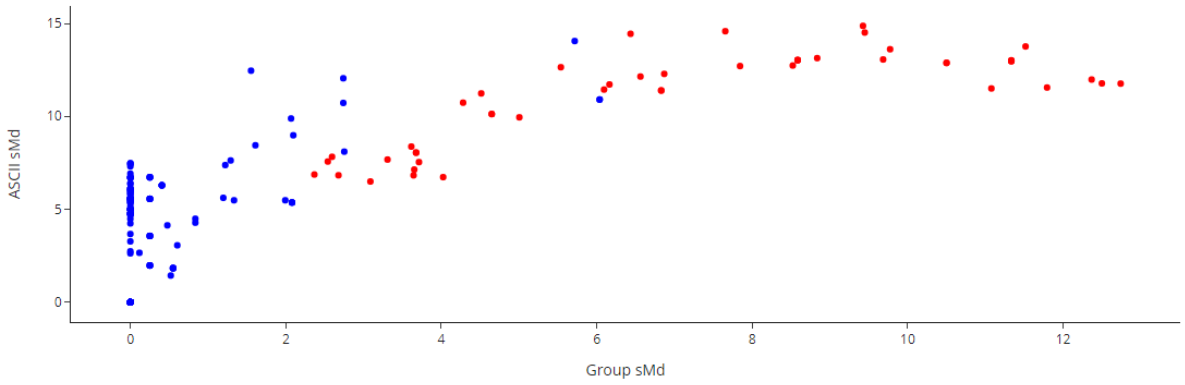

Figure 3: Simplified Mahalanobis distance anomaly scores for POST and GET requests using ASCII vs categorized character distributions

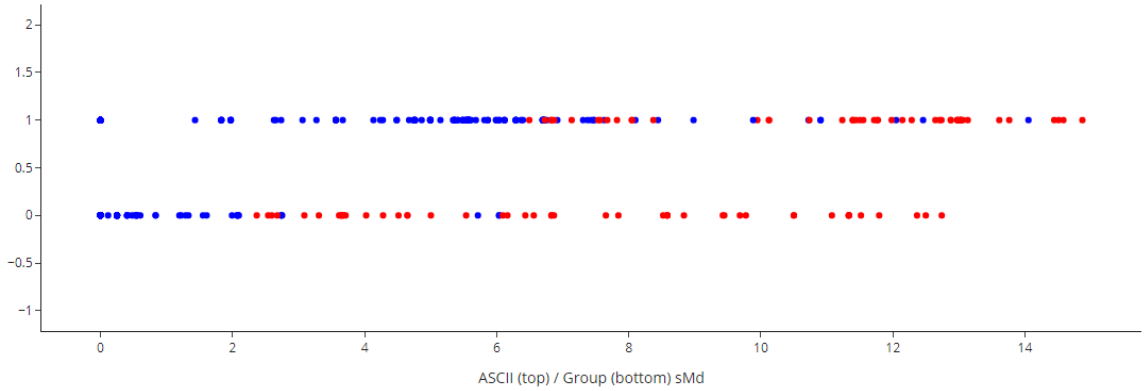

Figure 4: Simplified Mahalanobis distance anomaly scores for POST and GET requests using ASCII (top) and categorized (bottom) character fistributions

Figures 3 and 4 compare the maximum anomaly scores for parameters passed via POST and GET requests for two different character distributions. The graphs represent the same data but on different axes. The relative frequency of each ASCII character and a group of characters was calculated and compared to every request using the simplified Mahalanobis distance. The grouped character distribution seems to outperform the ASCII distribution. If we were to pick the smallest anomalous point (6.494 for ASCII, and 2.367 for group) and count the number of times a benign request would appear with a greater anomaly score, the ASCII distribution would have 26 benign requests misclassified while the group distribution would only have 6. This is because the ASCII distribution is much more likely to incorrectly mark a benign request as anomalous if some characters in the request have never been seen before, while the group distribution is more lenient.

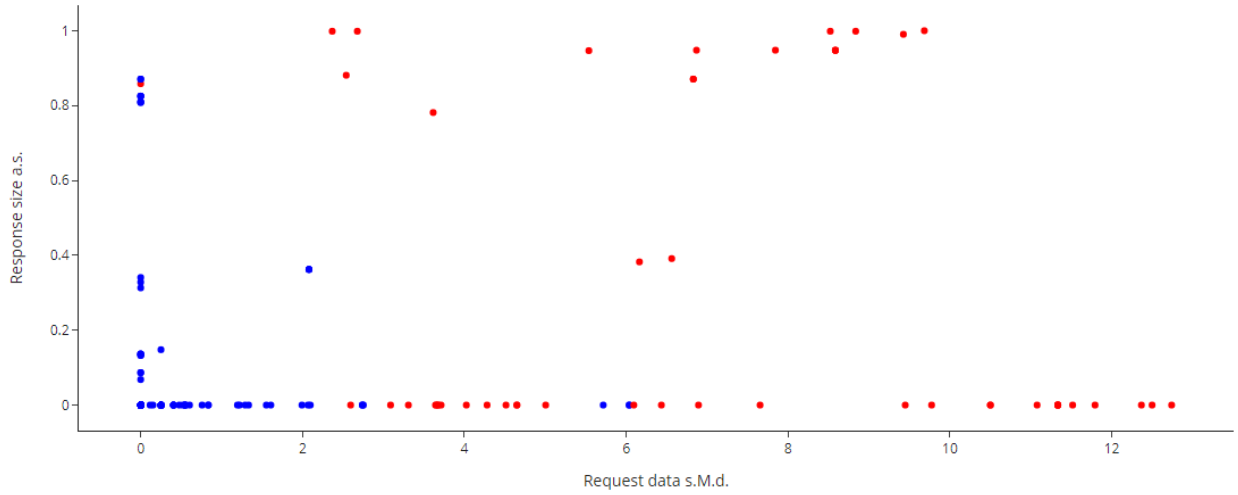

Figure 5: Response size vs request parameters anomaly scores

Taking this a step further, we can combine multiple metrics to come up with a better overall indicator of an anomalous request. Different vulnerabilities manifest in different ways. For example, a directory traversal attack may not look too different from a regular request, but

the response size and data type might be completely different. On the other hand, a SQL injection attack might include strings and special characters in a number-only field, but the response size will not be different at all. Figure 5 above combines these two metrics. As a result, some previously ambiguous requests can now be clearly distinguished as good or bad

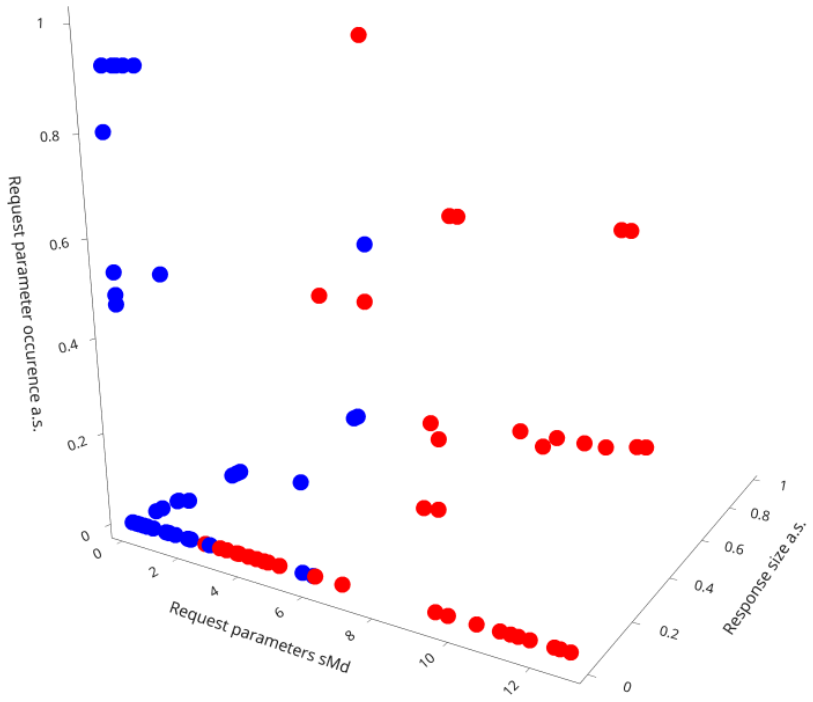

Figure 6: Request parameter occurrence vs request parameters vs response size anomaly scores

Finally, Figure 6 demonstrates the use of 3 different metrics at once to identify anomalies. Areas with benign and malicious requests can be easily distinguished. The additional, third metric (request parameter occurrence anomaly score) improved the detection only slightly. Even though only a single malicious request that would have otherwise been marked benign was identified, this plot demonstrates that some exploits can only be caught by certain metrics.

The system can be improved by adding some filtering. For example, requests with no cookies are likely to be either automated or account for the first times when the user visited the website. Such endpoints should be limited to only a few. Requests with unknown parameters, or lack of expected parameters could be flagged automatically as well.

Another observation was that mitmproxy tends to be very slow, and it is therefore not suitable for protecting production websites. Perhaps Nginx can be used as a reverse proxy instead, though this might require the current Python scripts to be rewritten in C.

V. Conclusion

Despite the simplicity of the calculations, the tool was successful at separating malicious traffic from benign. Additional testing and more data would allow us to calculate a more precise threshold for classifying traffic as anomalous. A machine learning approach can help automatically pair various web exploits with the metrics they affect and generate appropriate thresholds for identifying them.

In addition to detecting suspicious traffic, the tool can also block it. Insufficient baseline data would lead to many false positives, which can negatively affect the usability of the site. Lowering the anomaly threshold could provide some protection. Even weak security mechanisms and barriers can delay the attackers and give the incident response team some time to learn more about them.

VI. References

Cho, S., & Cha, S. (2004). SAD: web session anomaly detection based on parameter estimation. *Computers & Security*, 23(4), 312–319. doi: 10.1016/j.cose.2004.01.006

Crumpler, W., & Lewis, J. A. (2019, January 29). The cybersecurity workforce gap. *Center for Strategic and International Studies*. https://csis-prod.s3.amazonaws.com/s3fs-public/publication/190129_Crumpler_Cybersecurity_FINAL.pdf

Fruhlinger, J. (2017, September 13). What is the Heartbleed bug, how does it work and how was it fixed? *CSO Online.* https://www.csoonline.com/article/3223203/what-is-the-heartbleed-bug-how-does-it-work-and-how-was-it-fixed.html

Ibe, O. (2013). *Markov processes for stochastic modeling.* Newnes.

Internet Society. (2019). 2018 cyber incident & breach trends report. *Internet Society's Online Trust Alliance*. https://www.internetsociety.org/wp-content/uploads/2019/07/OTA-Incident-Breach-Trends-Report_2019.pdf

Kruegel, C., & Vigna, G. (2003). Anomaly detection of web-based attacks. *Proceedings of the 10th ACM conference on Computer and communication security*. doi:10.1145/948143.948144

OWASP Foundation. (2017). The ten most critical web application security risks. *The Open Web Application Security Project Foundation.* https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

Positive Technologies. (2019, March 5). Web application vulnerabilities: statistics for 2018. *Positive Technologies*. https://www.ptsecurity.com/ww-en/analytics/web-application-vulnerabilities-statistics-2019/

Reunanen, N., Räty, T., Jokinen, J. J., Hoyt, T., & Culler, D. (2019). Unsupervised online detection and prediction of outliers in streams of sensor data. *International Journal of Data Science and Analytics, 9*(3), 285-314. https://doi.org/10.1007/s41060-019-00191-3

Shieh S., & Gligor V. (1992). Pattern-oriented intrusion-detection system and method. United States Patent US5278901A. https://patents.google.com/patent/US5278901A/en

Wang, K., & Stolfo, S. J. (2004). Anomalous payload-based network intrusion detection. *Columbia University*.

http://www1.cs.columbia.edu/~locasto/projects/candidacy/papers/print/kewang2004payl.pdf

Wen, K., Guo, F., & Yu, M. (2013). Adaptive anomaly detection method of web-based attacks. *Journal of Computer Applications*, 32(7), 2003-2006. doi:10.3724/sp.j.1087.2012.02003

Winkler, J., & Page, W. (1989). Intrusion and anomaly detection in trusted systems. *[1989 Proceedings] Fifth Annual Computer Security Applications Conference*. doi:10.1109/csac.1989.81023

Zhang, M., Lu, S., & Xu, B. (2017). An Anomaly Detection Method Based on Multi-models to Detect Web Attacks. *2017 10th International Symposium on Computational Intelligence and Design (ISCID).* doi:10.1109/iscid.2017.223

Zhang, S., Li, B., Li, J., Zhang, M., & Chen, Y. (2015). A Novel anomaly detection approach for mitigating web-based attacks against clouds. *2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing*. doi:10.1109/cscloud.2015.46