Automating Suricata Rule-Writing

by

Brian T. Carr

A Capstone Submitted to the Faculty of

Utica College

August 2021

in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Cybersecurity

© Copyright 2021 by Brian T. Carr

All Rights Reserved

**Abstract**

Network Security Monitoring (NSM) is a practice used to defend a network of resources against threats. To detect new threats, signature-based Intrusion Detection Systems (IDS) such as Suricata require rule development. Signature-based detection is the simplest method for detecting malicious activity. The verbosity of Suricata rules makes it hard for Suricata administrators to deploy new rules rapidly. The Open Information Security Foundation OISF (2021) acknowledges that most Suricata administrators use the default ruleset exclusively. Signature-based security solutions such as Suricata rely solely on their rulesets for their detection capabilities. The signatures contained within these rulesets define the anomalies which generate alerts. This research project provided a command-line interface Python script that outputs syntactically valid Suricata rules. The author intended that Suricata administrators and others would use the Python script developed during this research project to assist the rapid deployment of novel Suricata rules.

*Keywords:* Cybersecurity, Professor Carmen Mercado, Detection Engineering, Network Traffic Analysis, Malware, Cybercrime,

Table of Contents

[List of Illustrative Materials vi](#_Toc80574683)

[Automating Suricata Rule-Writing 1](#_Toc80574684)

[Purpose Statement 2](#_Toc80574685)

[Research Questions 3](#_Toc80574686)

[What is Network Security Monitoring? 3](#_Toc80574687)

[What is a Signature? 4](#_Toc80574688)

[Intrusion Detection Systems 5](#_Toc80574689)

[Signature-Based Network Security Monitoring Limitations 5](#_Toc80574690)

[Encrypted Network Traffic 6](#_Toc80574691)

[Value of Rulesets in Network Security Monitoring Solutions 7](#_Toc80574692)

[Scripting and Automation Potential 8](#_Toc80574693)

[Literature Review 8](#_Toc80574694)

[Network Security Monitoring Background and Definition 8](#_Toc80574695)

[The History of Network Security Monitoring 9](#_Toc80574696)

[Purpose and Value of Network Security Monitoring 10](#_Toc80574697)

[Intrusion Detection and Intrusion Prevention Solutions 11](#_Toc80574698)

[Intrusion Detection Systems 11](#_Toc80574699)

[Intrusion Prevention Systems 12](#_Toc80574700)

[Network-Based vs. Host-Based Solutions 12](#_Toc80574701)

[Intrusion Detection Systems 12](#_Toc80574702)

[Signature-Based Intrusion Detection Systems 13](#_Toc80574703)

[Anomaly-based Intrusion Detection Systems 13](#_Toc80574704)

[Suricata 14](#_Toc80574705)

[Suricata Rules 14](#_Toc80574706)

[Open Information Security Foundation 15](#_Toc80574707)

[Signatures and Signature-Based Security Solutions 15](#_Toc80574708)

[Network Traffic 16](#_Toc80574709)

[Data Encapsulation 19](#_Toc80574710)

[Developing Signatures for Encrypted Traffic 20](#_Toc80574711)

[The Value of Rulesets in Signature-Based Network Security Monitoring Solutions 21](#_Toc80574712)

[Default Rulesets 21](#_Toc80574713)

[The Shortcomings of Signature-Based Network Security Monitoring 22](#_Toc80574714)

[False Negative Events: Missed Alerts 23](#_Toc80574715)

[The Importance of Rule-Writing 23](#_Toc80574716)

[Discussion of Findings 24](#_Toc80574717)

[The Value of Network Security Monitoring 24](#_Toc80574718)

[The Value Rulesets of Signature-Based Intrusion Detection Systems 25](#_Toc80574719)

[Signatures for Encrypted Traffic 25](#_Toc80574720)

[Signature-Based Intrusion Detection System Limitations 26](#_Toc80574721)

[Simplifying the Rule Writing Process 26](#_Toc80574722)

[Generating Suricata Signatures with Suri-rule-gen.py 27](#_Toc80574723)

[Argparse Module Application 27](#_Toc80574724)

[Input Validation 29](#_Toc80574725)

[File Handling 30](#_Toc80574726)

[Suri-rule-gen.py Usage 31](#_Toc80574727)

[Future Research and Recommendations 32](#_Toc80574728)

[Conclusion 33](#_Toc80574729)

[References 35](#_Toc80574730)

[Appendix A 38](#_Toc80574731)

# List of Illustrative Materials

Figure 1 – Suricata Rule Syntax 4

Figure 2 – IPv4 Packet Structure 16

Figure 3 – Suricata Packet Processing using autofp Runmode and Multiple Capture Threads 18

Figure 4 – OSI Model Showing Data Transmission 19

Figure 5 – Packet Encapsulation Visualization 20

Figure 6 – Use of Argparse Python Module in suri-rule-gen.py 28

Figure 7 – Lists and Regular Expressions used for Input Validation 29

Figure 8 – Input Validation Performed on User-Provided Destination IP 30

Figure 9 – File Handling from suri-rule-gen.py 30

Figure 10 – Writing a Suricata rule with suri-rule-gen.py 31

Figure 11 – Command-line switches in suri-rule-gen.py 32

# Automating Suricata Rule-Writing

"The current state of security for Internet-connected systems makes me think of the Wild West" (Sanders & Smith, 2014, p. 1). The Internet Crime Complaint Center (IC3) reported that from 2016-2020 victims in the United States experienced a combined total of $13.3 billion in total losses from cybercrime (IC3, 2021). All organizations that are networked to the internet are likely to experience a breach (Bejtlich, 2013). Many security solutions defend against cybercrime and other malicious activity. At the network level, Intrusion Detection Systems (IDS) can identify and alert on desired activity. Signature-based detection is the simplest method of detecting malicious activity. Signature-based detection solutions match known malicious indicators to newly supplied data (Scarfone & Mell, 2007). One popular signature-based network security tool is Suricata. Suricata is a network-based IDS, Intrusion Prevention System (IPS), and Network Security Monitoring (NSM) solution that protects many different networks (OISF, 2021).

The goal behind this research was to develop a tool that automated Suricata rule-writing process. The author's intention for the tool developed in this research project was to assist Suricata administrators in rapidly deploying Suricata rules. The resulting Python script named suri-rule-gen.py helps inexperienced rule-writers who may encounter issues with Suricata's rule syntax. This project contains information from scholarly articles, website blog posts, textbooks, and official Suricata documentation and training resources. The tool developed was intended for Security Analysts, Network Administrators, Detection Engineers, Malware Analysts, Incident Response Analysts, and all other Suricata administrators. Additionally, the information comprising this project is relevant to all NSM administrators.

Computers have become embedded in all facets of every industry. From hospitals to federal government agencies, every organization has found a reason to utilize information technology systems in their daily activities. These computers and the networks they comprise are valuable. Although, they are not arbitrarily secure. While firewalls attempt to block unwanted connections, they have limited alerting mechanisms if something malicious has occurred. NSM solutions address this problem by generating alerts based upon the rulesets they utilize. Alhomoud et al. (2011) explained that NSM technologies such as network-based IDS are necessary for detecting malicious activity at the network level. The 13th control from the Center for Internet Security (CIS) CIS Controls V8 (2021) is the ability to operate processes and tooling to practice network monitoring for security threats across an organization's network. Organizations can implement this control through the organizational practice of NSM.

## Purpose Statement

The purpose of this research project was to develop tools that automate steps in the Suricata rule-writing process. The general problem is that signature-based NSM solutions such as Suricata require administrators to write rules to identify new anomalies (Einay et al., 2021). The specific problem is that many Suricata administrators and analysts dealing with NSM solutions find that the rules utilized by the Suricata open-source NSM solution are challenging to develop. The Open Information Security Foundation (OISF) (2021) acknowledged that most Suricata administrators exclusively use the default rulesets. It is possible that if Suricata rules became easier to write, more Suricata administrators would develop custom rules.

Additionally, signature-based NSM solutions heavily rely on their rulesets for functionality (Einay et al., 2021). Sanders (2014) stated that one issue plaguing NSM is that it requires multiple personal with specialized expertise to implement effectively. While less experienced junior analysts can practice NSM, guidance from a senior analyst is required (Sanders, 2014). Additionally, rule-developers currently require a wide range of skills to write valuable signatures which cover attack variations while still being efficiently utilized by the IDS engine (Shipulin, 2018). The CIS Controls V8 also stated that through security operation, organizations would locate tactics, techniques, and procedures of attackers in addition to IOCs that are useful in the detection process of network monitoring (CIS, 2021). The tool developed during this research project assists in the rapid deployment of Suricata signatures.

## Research Questions

The research contained within this project attempted to answer the questions: 1) Why is NSM valuable? 2) How necessary are signatures in signature-based NSM solutions? Furthermore, 3) How can rule-writing for Suricata be simplified for Suricata administrators?

## What is Network Security Monitoring?

Bejtlich (2013) described NSM as a threat-centric approach to detecting and responding to specified anomalies. NSM methodology and NSM solutions are valuable in any network environment. In most NSM solutions, an IDS generates alerts when it detects an anomaly. NSM centers on the idea that prevention eventually will fail; subsequently, NSM involves detection methods and not prevention methods (Sanders & Smith, 2014). Network-based NSM solutions lie between the network segment which it is monitoring and the network egress point. The NSM solution receives network traffic via a port mirroring solution (sometimes referred to as SPAN ports) or a network tap (hardware or software device used to duplicate network traffic). The IDS engine can compare the traffic to the ruleset rules and match the desired anomalies by obtaining this traffic. While NSM is a methodology that goes beyond solutions such as Suricata, the NSM solution handles the core functionality of detecting and alerting upon specified network events (Bejtlich, 2013).

Signature-based NSM solutions allow for entities to protect against the threats identified within their rulesets. These rulesets contain signatures for cyberthreats such as ransomware or trojan malware infections, phishing sites, scanning activity, application use, and much more. NSM solutions are valuable because they can detect threats related to the commission of cybercrime or other nefarious activities.

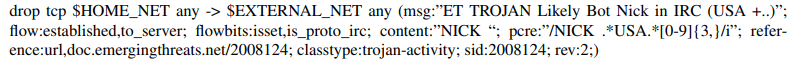
### What is a Signature?

Joshi et al. (2012) explained that a signature is a piece of data that can distinguish data. Signatures contain a combination of strings of characters, hex values, or various other static values. Joshi et al. stated that intrusion detection systems utilize signatures to locate security concerns in network traffic (2012).

Suricata uses rules to identify signatures. Rule files contain rules used by the Suricata engine, and the rule files have a .rules extension. These rule files contain the signatures designed for the Suricata engine to utilize when processing network traffic. Figure 1shows a sample Suricata rule. The sample Suricata rule contains an action, a header that defines the protocol, addresses, and ports involved in the rule, and the rule options that make the rule more specific (OISF, 2021).

**Figure 1**

*Suricata Rule Syntax*



*Note.* Image obtained from official Suricata documentation (OISF, 2021).

## Intrusion Detection Systems

Andress (2019) defined IDS as software or hardware solutions that monitor networks, hosts, or applications for unauthorized activity. The Suricata engine monitors network traffic, looking for specified events. The Suricata engine can be tuned to ignore certain events or even detect innocuous events by tuning the ruleset which it utilizes (OISF, 2021). Broucek and Turner (2004) described the two types of IDS which can monitor traffic at a network level. The two types of IDS are anomaly-based solutions and signature-based solutions. Anomaly-based IDS solutions function by detecting unusual or abnormal behavior by comparing current events to a baseline of the system's standard behavior. This method typically has a higher rate of false positives. On the other hand, signature-based methods compare traffic to a database of rules which contain signatures of previously identified malicious indicators (Broucek & Turner, 2004).

Sikorski and Honig (2012) explained that IDS solutions and other security appliances allow network administrators to employ signature-based defenses against malicious content. Commonly located Indicators of Compromise (IOC) include Uniform Resource Locators (URL) and Internet Protocol (IP) addresses. These IOCs frequently appear in network traffic and subsequently make for good NSM signatures. However, these signatures are often not valuable for very long because malicious actors regularly migrate infrastructure to evade security solutions (Sikorski & Honig, 2012). The short lifespans of these campaign-specific IOCs emphasize the need for organizations to develop and implement signatures for novel threats in a minimal amount of time.

## Signature-Based Network Security Monitoring Limitations

Signature-based NSM solutions have a significant shortcoming; They cannot detect anomalies for which they do not have signatures. Signature-based NSM solutions are as valuable as the signatures which they are utilizing in their rulesets. Additionally, while reputable and mature organizations develop default rulesets for IDS such as Suricata, they may be missing novel threats. Minimizing the time between IOC identification and signature deployment shrinks the risk associated with the threat that the signature identifies (Andress, 2019). The OISF (2021) reported that most Suricata administrators exclusively utilize preexisting rulesets. These entities could directly benefit from writing and implementing signatures for unaccounted threats that do not exist within the default rulesets. Entities that exclusively enable the default rulesets will be limited to detecting anomalies identified by these rule writing entities (OISF, 2021).

### Encrypted Network Traffic

Transport Layer Security (TLS) and Secure Sockets Layer (SSL) are protocols used to encrypt communications for everyday applications. Both legitimate applications and malware utilize encryptions protocols to encrypt their communications. To combat the threats associated with encrypted network traffic, researchers at Salesforce developed solutions that identify encrypted TLS communications by providing them with a unique value used as a fingerprint. These solutions were JA3 and JA3S. These solutions work by using the MD5 message-digest algorithm on a uniquely formatted string that contained values from the TLS Client Hello and TLS Server Hello packets to generate a hash value that can identify future TLS communications (Althouse, 2019).

Papadogiannaki and Ioannidis (2021) reported that over 75% of worldwide internet traffic is encrypted. When an encryption protocol such as SSL/TLS encrypts TCP level traffic, it does not encrypt the IP packet header. This encrypted traffic poses a problem for NSM solutions as many signatures identify malicious content within the packets of network traffic. Suppose the traffic under analysis by the IDS is encrypted. In that case, the content that these rules would match on is obfuscated, minimizing the effectiveness of content-based signatures in the face of encrypted traffic. Papadogiannaki and Ioannidis addressed this in their IDS solution known as HeaderHunter. HeaderHunter efficiently analyzes encrypted network traffic by focusing solely on analyzing the packet header and metadata. This disregard for network traffic content significantly limits potential signature development; signatures that look for information within the packet header or metadata will have value when applied to encrypted traffic. Additionally, HeaderHunter functions in a way that made signature generation easy for administrators (Papadogiannaki & Ioannidis, 2021).

## Value of Rulesets in Network Security Monitoring Solutions

Signature-based IDS engines such as Suricata function by comparing packets of network traffic flowing across a network to rules contained within rule files. Andress (2019) explained that signature-based IDS solutions have similar functionality to signature-based antivirus (AV) solutions. Both signature-based IDS and AV store signatures in a file or database, allowing the solutions to detect the desired anomalies. Additionally, both signature-based NSM and antivirus have the same shortfall of restrictively detecting anomalies with existing signatures. Novel threats may go unidentified, severely limiting the effectiveness of signature-based solutions (Andress, 2019). To combat this shortfall, this research project's author suggests that rule-writing has become a necessary skill of any IDS administrator.

Ganesan et al. (2019) described two types of threats, *located* threats, and *novel* threats. These novel threats have not yet been located currently in existence. Additionally, Ganesan et al. explained that if a threat changes in the slightest manner, it may require the signature to be adjusted to catch this change. The constant need for signature updating and management is one of the critical factors that lead to this project's development. Failure to quickly adjust rulesets can result in false negatives (where an alert should have been generated but was not).

## Scripting and Automation Potential

The scripts contained within this research aim to automate the generation of signatures that matched data in the packet header or packet metadata. In a training video produced by the OISF and Applied Network Defense (AND) (2017), the author explained that signatures that match DNS calls and other regularly occurring events are straightforward to develop and have the potential to be scripted. A focus on IP packet headers and metadata was applied when developing the code for this research project.

# Literature Review

## Network Security Monitoring Background and Definition

In Bejtlich's (2013) book, *Practical Network Security Monitoring,* the author explained that the principles of the NSM methodology are collection, analysis, and incident escalation upon detection. Bejtlich also stated that NSM is a methodology that allows entities to detect intrusions and minimize their opportunity to wreak havoc (2013). The book, *Foundations of Information Security,* delineated that an entity could detect malicious activity by auditing the information in an environment. IDS solutions are monitoring and alerting tools that can generate alerts (Andress, 2019).

In their book, *Applied Network Security Monitoring,* Sanders and Smith (2014)illustrated that the practice of network security monitoring has three main functions. Those functions were collection, detection, and analysis. When practicing NSM, the collection process could involve various data types depending on the organization's resources. Some organizations will capture full packet capture data, while others will have enough storage space to store alert data. The detection function of NSM is the function related to the signatures within the rulesets that the IDS is utilizing. Sanders and Smith described the analysis stage of NSM as having an analyst investigate and interpret the data located during the detection stage. Various types of analysis can occur at the analysis stage of NSM. Some types of analysis include packet analysis, network forensics analysis, host forensic analysis, and malware analysis. The analysis takes the most time out of the three stages because it involves engaging incident response personal (Sanders & Smith, 2014).

Einay et al. (2021) published a research article that explained that one of the primary weaknesses of many security solutions is their inability to detect intrusions at a network level. IDS can detect intrusions at a network level, subsequently accounting for this weakness. IDS utilize inline network traffic to detect many types of anomalies (Einay et al., 2021).

The Center for Internet Security (2018) posted on their website blog that they offer an NSM solution known as *Albert*. Albert is a passive IDS that detects malicious activity by generating alerts from threat signatures. The Albert IDS utilizes the Suricata engine to facilitate this detection capability (CIS, 2018). In a blog on their website, CIS explained that State, Local, Tribal, and Territorial government entities utilize Albert to detect malicious activity promptly. CIS additionally stated that NSM plays an essential role in an organization's defense-in-depth strategy (CIS, 2021b). The CIS Controls V8 13th control, Network Monitoring and Defense, highlights the need to monitor for threats at a network level (CIS, 2021a).

### The History of Network Security Monitoring

In Richard Bejtlich's (2013) publication, *The Practice of Network Security Monitoring,* Bejtlich provided the history of Network Security Monitoring. Bejtlich stated that NSM informally began when Todd Heberlein developed an application named the Network Security Monitor in 1988. Heberlein's Network Security Monitor was the first IDS to utilize data traffic going across the network as the data used to generate alerts. According to Bejtlich, in 1993, Heberlein worked with the Air Force Computer Emergency Response Team (AFCERT) to deploy another version of the Network Security Monitoring named the Automated Security Incident Measurement (AISM) system. Bejtlich codified NSM's definition during a webcast for SearchSecurity in 2002 (2013).

Sanders and Smith (2014) explained that before the conceptualization of NSM, the methodology of locating malicious threats at a network level was known as intrusion detection and focused on detecting known malicious signatures. In contrast, detection is one of the three processes within the NSM methodology. Intrusion detection is a vulnerability-centric approach, while NSM is a threat-centric approach. The goal of NSM is to detect in hopes of responding more time efficiently (Sanders & Smith, 2014).

### Purpose and Value of Network Security Monitoring

Sikorski and Honig (2012) published a book, *Practical Malware Analysis,* that described how signature-based IDS and IPS solutions allow network administrators to employ content-based countermeasures. These solutions often allow analysts to write signatures for IP addresses or URLs, making them ideal for network-wide detection. By writing signatures to match upon desired traffic anomalies, security and incident response personnel are now able to protect against previously identified threats (Sikorski & Honig, 2012).

Alhomoud et al. (2011) published a research article in *Procedia Journal of Computer Science* which explained that IDS solutions are an essential component of any entity's environment. IDS solutions have the functionality of detecting malicious traffic going across the network. Alhomoud et al. added that organizations are currently in need of valuable detection systems (2011).

## Intrusion Detection and Intrusion Prevention Solutions

Sanders (2017) posited that NSM does not involve prevention as preventative measures eventually fail. Instead, the NSM methodology focuses on the detection of anomalous activity. While teams implementing the NSM methodology can use IPS, NSM deals exclusively with IDS (Sanders, 2017). Andress (2019) explained that IDS could solely monitor and detect. IDS cannot take any direct action preventing or altering the flow of network traffic.

On the other hand, IPS systems sit in line with the network traffic (Firewalls also must sit inline), allowing them to take actions depending on what the engine is utilizing for a ruleset (Andress, 2019). In intrusion detection, there are four possible outcomes for each alert. These possible alert outcomes are false positive, false negative, true positive, and true negative. True positive alerts are alerts that generate when intended. True negatives are events that should and do not generate an alert. False positives are alerts that have been generated but should not have been. False negatives are alerts that did not generate when they were supposed to (Vacca, 2014).

### Intrusion Detection Systems

Andress (2019) described IDS solutions as hardware or software tools that monitor hosts, networks, and applications for desired anomalies. There are two types of IDS solutions, signature-based and anomaly-based. Sanders and Smith (2014) indicated that signature-based IDS are the most common and function by analyzing packets of network traffic for signatures derived from IOCs. In signature-based IDS, when a packet of network traffic matches a signature, an alert is generated. Modern IDS solutions can detect malicious activity that goes beyond the scope of traditional malware infections (Sanders & Smith, 2014).

### Intrusion Prevention Systems

IPS solutions are security solutions that often take data sent by the IDS to take action on desired anomalies. An example of an IPS taking action would be rejecting traffic from a specified IP address. The IDS solution, on the other hand, could locate the traffic and generate an alert. IDS cannot take action as they sit out of line with the network egress point (Andress, 2019).

## Network-Based vs. Host-Based Solutions

Broucek and Turner (2004) stated that there are four types of intrusion detection systems, host-based IDS (HIDS), application-based IDS (AIDS), stack-based IDS, and network-based IDS (NIDS). Vacca (2014) indicated that network-based IDSs monitor the packets of network traffic traveling across a network for signs of malicious or desired activity. Vacca also stated that the IDS could identify attacks before reaching their targets (2014).

Sikorski and Honig (2012) described how signatures developed from IOCs such as IP addresses and URLs effectively defend against malware and malicious activity. However, these signatures may be of value for a short time. The limited lifespan of these signatures is due to the malicious actor's ability to quickly change infrastructure to evade security solutions (Sikorski & Honig, 2012).

## Intrusion Detection Systems

In an article published in *Network Security,* Shipulin (2018) stipulated that many organizations utilize IDS solutions to control and inspect the traffic inside of the network. These solutions allow organizations to investigate what type of traffic flows across their network. Broucek and Turner (2004) explained that signature and anomaly-based IDS detect anomalies and malicious activity. Signature-based IDS constantly compare the collected data is against a database of known attacks and vulnerabilities. Regarding anomaly-based IDS, the system attempts to identify abnormalities or changes in behavior instead of the system's everyday activities. Anomaly-based IDS are subject to a more significant number of false-positive alerts than signature-based IDS solutions.

Sanders and Smith (2014) suggested that an anomaly is an event in a system or network that is observable and considered unusual. Both signature-based and anomaly-based IDS locate anomalies through detection methods. Upon locating anomalies, detection tools such as IDS generate alerts.

### Signature-Based Intrusion Detection Systems

Andress (2019) explained that signature-based IDS function similarly to signature-based antivirus (AV) solutions. Both signature-based IDS and AV utilize a database of signatures and compare data to those databases. This data might be the file on a hard drive or an outbound web connection regarding AV solutions. In network-based IDS, this data is network traffic flowing to and from the network egress point.

A signature is a portion of data used to distinguish that specific data. Signatures contain a combination of a string of characters, hex values, or various other static values. IDS utilize signatures to locate security concerns in network traffic (Joshi et al., 2012). Shipulin (2018) stated that signature sets are pivotal in detecting known attack vectors.

### Anomaly-based Intrusion Detection Systems

Andress (2019) delineated that anomaly-based IDS work by developing a standard network profile based on the normal activities on the network and alerting when the network traffic deviates from the profile. Some of the data collected by the anomaly-based IDS include the amounts and type of traffic sent over the network. Andress (2019) additionally acknowledged that anomaly-based IDS tend to produce more false-positive alerts than signature-based IDS. Broucek and Turner (2004) stated that anomaly-based intrusion detection systems generally have a higher rate of false-positive alerts generated when compared to signature-based IDS.

## Suricata

Suricata is a signature-based IDS/IPS solution that utilizes rulesets to alert or act upon rule matches in sniffed traffic. Suricata is a networked-based solution; other types of IDS solutions include host-based or application-based IDS solutions. The OISF developed Suricata with funding from the Department of Homeland Security's Directorate for Science and the Homeland Open Security Technology (HOST) program. The OISF released Suricata in July of 2010 (Alhomoud et al., 2011).

In the official Suricata documentation provided by the OISF, they defined Suricata as a well-functioning network IDS, IPS, and NSM engine. Suricata utilizes threads, thread-modules, and queues to process the network packets provided to the engine. Suricata additionally has three types of thread-modules, which are decoding, detecting, and output modules. Each thread can utilize one or more thread-modules. Runmodes refer to the Suricata engine's different configurations for threads, thread modules, and queues (OISF, 2021).

### Suricata Rules

In the official Suricata documentation from OISF (2021), rulesets and the signatures they contain are critical to the functionality of the Suricata engine. The rules utilized by the Suricata engine have three main sections. The first section of each Suricata rule is the rule action that determines the Suricata engine's action when a detection is made by triggering a match. The second section is the header that defines the rule's protocol, IP addresses, ports, and directionality. For more complex and rules, Suricata can utilize Lua scripts for more granular detection capabilities.

Additionally, the official Suricata documentation stated that rulesets affect both the detection capabilities of the Suricata engine and the performance of the system hosting the Suricata engine. Suricata uses rules to facilitate detection capabilities in detection thread modules. Suricata utilizes system resources to facilitate detection functionality.

The OISF (2021) discussed the different capture configurations which Suricata can use to ingest network traffic. The official Suricata documentation stated that there are a variety of modes to configure Suricata to capture packets. These modes include pcap, pfring, and afpacket. Each of these modes captures network traffic slightly differently (OISF, 2021).

### Open Information Security Foundation

Suricata is developed and maintained by the OISF, which is a non-profit organization. The OISF licensed the Suricata source code under version 2 of the GNU General Public License (GPL). Suricata is open source, and the source code is available on GitHub. Suricata is compatible with various operating systems, including Linux-based OS such as Ubuntu or Red Hat Enterprise Linux, Mac OS X, and Windows (OISF, 2021).

## Signatures and Signature-Based Security Solutions

Andress (2019) identified that signature-based IDS function similarly to signature-based antivirus programs. Both types of solutions utilize a database that contains signatures that match desired anomalies. Shipulin (2018) explained that IDS solutions, AV solutions, and Web Application Firewall (WAF) solutions utilize signature-based detection methods.

In Special Publication 800-94 published by the National Institute for Standards and Technology (NIST), Scarfone and Mell (2007), explained that signature-based detection methods are the simplest detection methods to implement as they compare data from a packet, log entry, or another source to predefined values in order to locate matches. False-positive events are more common in anomaly-based detect solutions as profile training is inherently tricky (Scarfone & Mell, 2007).

## Network Traffic

Devices can communicate once networked together as long as they utilize a common networking protocol (Sanders & Smith, 2014). The TCP/IP protocol stack is what facilitates Internet-connected devices. The TCP/IP data is transmitted within an Ethernet packet using a process called data encapsulation. The Ethernet data is transmitted with a physical layer protocol and contains data for higher layer protocols such as TCP/IP, DNS, HTTPS, and more (Sanders & Smith, 2014). Figure 2 shows the structure of the data within an IPv4 packet.

**Figure 2**

*IPv4 Packet Structure*

Graphical user interface

Description automatically generated with medium confidence

*Note.* Image obtained from *Applied Network Security Monitoring* (Sanders & Smith, 2014).

Sanders and Smith (2014) explained that the Suricata engine takes network packets from a monitoring port, decodes them, analyzes them utilizing the ruleset, and generates any alerts located in the packets under analysis. The Suricata IDS engine goes through multiple steps before finally generating an alert. First, The packet capture thread obtains the packet off of the monitoring port. Next, Suricata attempts to decode information from the network packets. Lastly, Suricata utilizes the detection engine to detect signature matches before finally providing output (Sanders & Smith, 2014).

The official Suricata documentation elaborated on the various packet processing procedures for each of the available Suricata runmodes. Most runmodes utilize both packet capture threads and packet processing threads. Certain runmodes do not utilize packet capture threads but instead pass the packet directly to the packet processing threads (OISF, 2021). Figure 3 illustrates the Suricata packet processing method when configured to use the autofp runmode with multiple capture threads.

**Figure 3**

*Suricata Packet Processing using autofp Runmode and Multiple Capture Threads*

Diagram

Description automatically generated

*Note.* Image obtained from official Suricata documentation (OISF, 2021)

Network data exists at multiple layers of the Open Systems Interconnection (OSI) model (Sanders, 2017). The OSI model is a seven-layer model that explains the data's state during different transmission phases. The OSI model goes from the first layer, which is physical, to the application layer, which utilizes application-specific protocols (Sanders, 2017). Figure 4shows a graphic showing how data traverses the OSI model before being transmitted.

**Figure 4**

*OSI Model Showing Data Transmission*

Diagram

Description automatically generated

*Note.* Image obtained from *Practical Packet Analysis* (Sanders, 2017).

### Data Encapsulation

Data encapsulation allows for data to go through the OSI model (Sanders, 2017). Each layer of the OSI model adds a header or footer, allowing that layer to communicate. Sanders stated that the entire packet with all protocol headers passes across the network medium at layer one of the OSI model. Due to data encapsulation, each OSI model layer can only communicate with the layers above and below them. The layer one traffic, which contains all the header and footer values for each additional layer, is formally known as a protocol data unit (PDU). Figure 5shows a graphic displaying what data encapsulation looks like at each layer of the OSI model.

**Figure 5**

*Data Encapsulation Visualization*

Diagram

Description automatically generated

*Note.* Image obtained from *Practical Packet Analysis* (Sanders, 2017).

### Developing Signatures for Encrypted Traffic

Papadogiannaki and Ioannidis (2021) described the IDS they had developed to account for traffic encryption. Papadogiannaki and Ioannidis stated that over 75% of current internet traffic is encrypted. Papadogiannaki and Ioannidis created an IDS solution named HeaderHunter. HeaderHunter accounted for encrypted traffic by solely analyzing packet headers and metadata. Papadogiannaki and Ioannidis concluded that most IDS utilizes rules that depend on deep packet inspection techniques. Encrypted traffic is impossible to analyze by deep packet inspection in the same fashion as unencrypted traffic (2021).

In a blog post by Salesfore Engineering, John Althouse (2019) described JA3 and JA3S. JA3 is an open-source method used to identify TLS clients within encrypted traffic, JA3S functions the same way but for the server-side of the TLS communications. At the time of writing, JA3 and JA3S were accessible through <https://github.com/salesforce/ja3>. The OISF stated that Suricata comes with JA3 integration in keywords such as ja3.hash and ja3.string. Both of these JA3 keywords have corresponding JA3S keywords (OISF, 2021).

## The Value of Rulesets in Signature-Based Network Security Monitoring Solutions

Broucek and Turner (2004) stated that signature-based IDS are as valuable as the signature database. The OISF (2021) explained that Suricata uses rules that contain signatures to generate alerts. The rulesets are crucial to the functionality of the Suricata engine. Additionally, the OISF acknowledged that most Suricata administrators exclusively utilize rulesets developed by rule writing entities like Emerging Threats (OISF, 2021).

Proofpoint (2020) explained that their ET PRO ruleset contains IDS rules for the latest vulnerabilities and malware campaigns. The industry standard ruleset is the Emerging Threats Pro ruleset developed and maintained by a division of Proofpoint known as the Emerging Threats. Proofpoint reported analyzing three million malware samples per day in their proprietary sandbox (Proofpoint, 2020).

### Default Rulesets

In the article "We Need to Talk About IDS Signatures," Shipulin (2018) explained that these currently available rulesets result from numerous hours of research and development by individuals and entire organizations. Shipulin added that two of the most famous vendors who produce IDS rulesets are Emerging Threats and Cisco Talos (2018). The organization, Emerging Threats, is part of a larger enterprise security company known as Proofpoint. Proofpoint is responsible for developing a freely available set of IDS rules known as the ET Open ruleset and a paid alternative known as ET Pro. The ET Pro ruleset contains recently developed rules to detect active threats in a timely fashion (ProofPoint, 2021). The OISF stated that in most Suricata deployments, administrators utilize existing rulesets (2021).

## The Shortcomings of Signature-Based Network Security Monitoring

One of the most significant shortcomings of signature-based intrusion detection is that if a signature does not currently exist for a specific attack, it is likely that this will not generate an alert (Andress, 2019). Malicious actors may manipulate the traffic they generate specifically to avoid generating any alerts. These malicious actors may even utilize IDS tools in the process of testing their infrastructure (Andress, 2019). Vacca (2014) explained that signature-based IDS solutions depend on the signatures located within their rulesets to function. If an attack or an anomaly occurs, these signatures raise awareness by generating an alert. Signatures that are too specific may lead to false negatives. Missed alerts are false negatives in the context of an IDS (Vacca, 2014).

Sanders and Smith (2014) explained that NSM is challenging to practice, and NSM practitioners require a specific skill set. To practice NSM effectively, an organization must employ personal with a specialized skill set to conduct the collection, detection, and analysis stages (Sanders & Smith, 2014). Bejtlich (2013) stated that network-based IDS could not effectively monitor node to node traffic within the monitored network.

### False Negative Events: Missed Alerts

False negative alerts are events where an alert generation should have occurred but did not. In signature-based IDS, false negative events can occur when an anomaly happens, but there is no signature to detect it. Additionally, employing signatures that match a wide variety of attack vectors can lead to fewer false negatives than signatures that match specific content (Vacca, 2014).

## The Importance of Rule-Writing

In the CIS Controls V8, the 13th control, Network Monitoring and Defense, stated that organizations cannot rely on our network defenses never to fail. Security tools such as IDS are most effective when supported by a continuous monitoring process (CIS, 2021a). Signature-based IDS solutions could not detect attacks or anomalies that do not yet have a signature developed for detection (Broucek & Turner, 2004). A critical shortfall in many signatures that even the slightest variation may lead to a signature failing to report on the desired traffic anomaly (Scarfone & Mell, 2007)

Emerging Threats is an organization within Proofpoint, Inc. that develops rulesets for IDS such as Snort and Suricata. On Emerging Threats' (2019) website, they provided recommendations for IDS administrators on utilize rulesets. Emerging Threats recommended determining which rulesets are irrelevant to one's environment and disabling them. Additionally, Emerging Threats recommended writing rulesets that correspond to the Suricata administrator's network configuration. For example, analysts can write IDS rules can for the use of any ports which should be unused. Emerging Threats developed the Emerging Threats Pro and Emerging Threats Open rulesets for use in most environments (Emerging Threats, 2013).

In Brandon Rice's thesis published by James Madison University, Rice (2014) described a project automating the generation of signatures for the Snort IDS. Rice's goal was to develop a tool that generated Snort rules from any file type. Rice designed an algorithm that takes an input file path and outputs a syntactically acceptable rule for the Snort IDS. Rice stated that his project could simplify the rule development process while also allowing a fully automated process by utilizing a tool to periodically execute the rule generation script. Rice additionally acknowledged that his tool could assist the rapidly deploy rules upon identifying a novel threat (2014).

# Discussion of Findings

## The Value of Network Security Monitoring

In the simplest of terms, the practice of NSM can detect suspicious events on a network. NSM involves collecting network traffic and detecting and analyzing anomalies within that traffic (Sanders & Smith, 2014). Network-based IDS can detect anomalies within network traffic. Signature-based network IDS detect these anomalies by comparing packets of network traffic to signatures contained within the applied ruleset (Bejtlich, 2013). Signature-based IDS function similarly to signature-based AV solutions (Andress, 2019). The Suricata engine utilizes rulesets to identify the signatures which generate alerts. Additionally, practicing NSM provides entities with a valuable layer of defense.

The Center for Internet Security validated the need for NSM within their CIS Controls V8. The 13th control from the Center for Internet Securities CIS Controls V8 is the ability to operate processes and tooling to practice network monitoring for security threats across an organization's network (CIS, 2021). A primary weakness of many security solutions is their inability to detect intrusions at a network level. Practicing NSM with NSM solutions such as Suricata can account for this weakness.

## The Value Rulesets of Signature-Based Intrusion Detection Systems

In the official Suricata 7.0 documentation, the OISF stated that it is necessary to install signatures for Suricata as the engine utilizes those signatures to trigger alerts. Suricata's detection thread-modules utilize signatures to detect anomalies. Most Suricata runmodes utilize detection thread-modules to perform detection (2021). IDS such as Suricata function by comparing the packets of network traffic to signatures contained within the IDS ruleset (Bejlitch, 2013). If no rulesets are applied, the IDS will not generate an alert due to the inability of the detection engine to facilitate matches.

The Suricata documentation further stated that rulesets are crucial to both the detection capabilities of the IDS and the performance capabilities of the host system. The Suricata engine utilizes system resources for each signature within each detection thread-module. Enabling inoperative signatures on an entity's IDS can take valuable resources away and potentially overwhelm the system (OISF, 2021). Shipulin (2018) explained that for every one thousand signatures or 500 Mbps of network traffic, the system hosting the IDS application requires a CPU core.

### Signatures for Encrypted Traffic

Papadogiannaki and Ioannidis designed their HeaderHunter IDS to alert on encrypted traffic by focusing the signatures utilized by the IDS to the network packet headers and metadata. HeaderHunter does not analyze the data contained within each packet, which allows it to utilize fewer system resources (Papadogiannaki and Ioannidis, 2021). HeaderHunter does not analyze network packet content which prevents the solutions from matching on application-level data. On the other hand, Suricata is able to perform detection on application layer data.

Various projects focus on performing detection on encrypted network traffic. Engineers at Salesforce created a solution known as JA3 and JA3S to develop signatures for TLS encrypted traffic (Althouse, 2019). Suricata comes with JA3 and JA3S integrations. Suricata utilizes unique rule option keywords to match JA3 and JA3S strings and hash values (OISF, 2021).

Suricata allows rule writers to develop signatures that detect anomalies within encrypted traffic by utilizing keyword integrations for signatures within application layer encryption methods such as JA3, SSL/TLS, and SSH (OISF, 2021). Because of the compatibility for various application-level encryption protocols, Suricata can adequately detect anomalies in both encrypted and unencrypted traffic. Due to their value in encrypted traffic, the author implemented many command-line switches for application-level encryption protocols in the suri-rule-gen.py Python script. Specifically, suri-rule-gen.py has CLI switches for JA3, SSH, and TLS keywords.

## Signature-Based Intrusion Detection System Limitations

Signature-based IDS are limited to detecting exclusively the anomalies for which they have signatures in their rulesets. If a specific threat does not yet have a signature within the applied ruleset, there will be no detection, and subsequently, no generation of an alert. When an anomaly exists without the generation of an alert, that is known as a false negative event. In the context of a network-based IDS, a false negative is when an anomaly occurs (Vacca, 2014). Signature-based-IDS are as valuable as the rulesets which they utilize (Broucek & Turner, 2004)

## Simplifying the Rule Writing Process

Tools such as the one developed within this research project provide Suricata analysts with more accessible routes to obtaining the rules they need. The development of future tools that automate steps of the rule-writing process will undoubtedly help NSM administrators develop and implement novel rules. Custom scripts can generate simple signatures such as those intending to match DNS calls (OSIF & AND, 2017). Suri-rule-gen.py implemented as many valuable keywords as possible in the form of CLI switches. Additionally, consideration for traffic encryption is applied when selecting which keywords to implement.

## Generating Suricata Signatures with Suri-rule-gen.py

The Python script developed during this research project outputs a Suricata rule which information from a command-line syntax. The resulting Python script, suri-rule-gen.py, fulfills this purpose by accepting variables through CLI switches constructing those variables into a Suricata rule. Suri-rule-gen.py implements various input validation checks which

catch common errors made when writing Suricata rules. The suri-rule-gen.py GitHub repository is available at https://github.com/CarrCyberSec/suri-rule-gen. Suri-rule-gen .py was written the Python 3 interpreter and the GitHub repository comes with a Python virtual environment. This virtual environment contains a Python interpreter that is compatible with the suri-rule-gen.py source code.

### Argparse Module Application

The author designed suri-rule-gen.py for the Linux CLI. The author used the Argparse Python module to create CLI switches used to pass variables to the script. Figure 6 illustrates the use of the Argparse Python module within suri-rule-gen.py. The Argparse Python module facilitates the core functionality of implementing command-line interface switches within suri-rule-gen.py. The user would provide the script with variables through CLI switches; two dashes precede these switches. The --help or -h switch are used to obtain the suri-rule-gen.py CLI options.

**Figure 6**

*Use of Argparse Python Module in suri-rule-gen.py*

A screen shot of a computer screen

Description automatically generated with low confidence

*Note.* Image of Argparse module usage within suri-rule-gen.py.

### Input Validation

The author implemented Input validation in several ways. The author used lists to validate variables with a finite number of options. For variables with a finite number of valid values, the author used lists to define the valid values, and then the variables were tested for being present in the list. For more complex values such as source IP and destination IP, the author implemented regular expressions. These regular expressions limit the invalid values entered while still maintaining enough flexibility to accept the various formats specified in the Suricata documentation. Figure 7 shows the Validation lists and regular expressions used for input validation. Figure 8 illustrates the use of regular expressions in suri-rule-gen.py. Figure 8 shows the regular expression for IP patterns named ip\_pattern validating the destination IP address.

**Figure 7**

*Lists and Regular Expressions Used for Input Validation*

A screenshot of a computer

Description automatically generated with medium confidence

*Note.* Image of lists and portion of the regular expressions used for input validation.

**Figure 8**

*Input Validation Performed on User-Provided Destination IP*

A screen shot of a computer

Description automatically generated with low confidence

*Note.* Image of input validation list and regular expression applied to the destination IP variable.

### File Handling

Suri-rule-gen.py produces three types of output, stdout, log data, and Suricata rules. The stdout output is displayed directly to the user in the console while utilizing the suri-rule-gen.py. Suri-rule-gen.py writes log data to suri-rule-gen.log. The rules are either output to suri-rule-gen.rules or the output file specified with the --output switch. Before writing a newly created rule to the rule file, suri-rule-gen.py first checks to see if the file exists. If the output file exists, then the rule is written as a new line in the output file. Figure 9 illustrates this output file test.

**Figure 9**

*File Handling from suri-rule-gen.py*

Text

Description automatically generated

*Note. Image of output file handling from suri-rule-gen.py.*

### Suri-rule-gen.py Usage

Suri-rule-gen.py is compatible with the Linux CLI. Suri-rule-gen.py has various CLI switches that allow the user to customize the content going into the rule. Figure 10 shows a Suricata rule generated with suri-rule-gen.py and the contents of the rule file. The available CLI switches for suri-rule-gen.py are obtainable with the --help or -h switch. Figure 11 shows A portion of the output obtained from running suri-rule-gen.py with the --help switch. Suri-rule-gen.py also provides information in a log file named suri-rule-gen.log. Suri-rule-gen.log is in the same directory as the suri-rule-gen.py Python script. Suri-rule-gen.py writes information about the values provided by the user and any errors to the log file.

**Figure 10**

*Writing a Suricata rule with suri-rule-gen.py*

Text

Description automatically generated

*Note. Image of suri-rule-gen.py used to generate a Suricata rule.*

**Figure 11**

*Command-line switches in suri-rule-gen.py*

Graphical user interface, text, application

Description automatically generated*Note. Image of suri-rule-gen.py used obtained with --help switch.*

## Future Research and Recommendations

Signature-based-IDS are as valuable as the rulesets they utilize (Broucek & Turner, 2004). The effectiveness of signature-based NSM solutions has direct ties to the ruleset utilized. The development of additional rule-writing tools may assist in the development and deployment of rules. In this research project, suri-rule-gen.py assists in the rule-writing process by helping to format rules from a command-line syntax. Other rule-writing tools have previously been developed, including Brandon Rice's tool autoSnortSig.py, which was published in his paper *Automated Snort Signature Generation* (Rice, 2014). Tools such as the one developed in this research project provide Suricata administrators with a lower entry bar for the rule-writing process. Still, more tools are needed to assist Suricata administrators, such as rule generators with a graphical user interface (GUI) or tools that can parse multiple rules from a single file. Further improvements in rule writing for IDS and specifically Suricata is possible by creating solutions that automate these rules' development, testing, and implementation. Many organizations cannot deploy a novel signature rapidly.

Additionally, there are still few scholarly works related to the subject of NSM. There are numerous materials related to IDS and network traffic analysis, but few for NSM. While books and blog posts from subject matter experts and NSM practitioners provide valuable insights, many individuals learning about NSM are negatively affected by the lack of quality reference materials (Sanders & Smith, 2014).

# Conclusion

This research project consisted of a deep dive into signature-based NSM and resulted in the development of a Python script that can assist users in writing rules for Suricata. The research contained within this project attempted to answer the questions: 1) Why is NSM valuable? 2) How necessary are signatures in signature-based NSM solutions? Furthermore, 3) How can rule-writing for Suricata be simplified for Suricata administrators?

In summary, NSM is a methodology concerned with collecting, detecting, and analyzing anomalies within network traffic. The value of NSM comes from the entity's ability to detect and minimize the consequences of intrusions at a network level (Andress, 2014). NSM employs detection methods as prevention eventually fails (Bejlitch, 2013; Sanders & Smith, 2014). Solutions such as network-based IDS can facilitate these detection capabilities. Signature-based security solutions are easier to use and tend to have a lower rate of false positives when compared to anomaly-based solutions (Broucek & Turner, 2004).

Suri-rule-gen.py generates rules that the Suricata engine can utilize. Suri-rule-gen.py assists with syntax issues while writing rules by accepting variables via CLI switches and automatically outputting a syntactically valid Suricata rule. Additionally, suri-rule-gen.py logs many of its actions and any errors related to the variable values. Various input methodologies generate these errors.

While this script can assist in the rule-writing process, it is not foolproof. Poorly written rules can still have detrimental performance consequences to the Suricata engine. Therefore, all rules generated with suri-rule-gen require verification for quality and functionality before being implemented in a production environment. Additionally, the rules which Suricata can use go beyond the scope of this project. Experienced NSM analysts with a significant amount of rule-writing experience may find suri-rule-gen.py limiting. The author's intention behind suri-rule-gen.py was to ease the rule-writing process for inexperienced Suricata administrators. Appendix A contains the source code for suri-rule-gen.py. <https://github.com/CarrCyberSec/suri-rule-gen> is the URL for the GitHub repository hosting the suri-rule-gen.py project. "As it has been since the early 1990s, NSM will continue to be a powerful, cost-effective way to counter intruders" (Bejlitch, 2013, p. 309).

# References

Alhomoud, A., Munir, R., Disso, J. P., Awan, I., & Al-Dhelaan, A. (2011). Performance Evaluation Study of Intrusion Detection Systems. *Procedia Computer Science*, 173-180.

Althouse, J. (2019). TLS Fingerprinting with JA3 and JA3S

https://engineering.salesforce.com/tls-fingerprinting-with-ja3-and-ja3s-247362855967

Andress, J. (2019). Foundations of Information Security*.* *No Starch Press.*

Bejtlich, R. (2013). The Practice of Network Security Monitoring*.* No Starch Press.

Broucek, V., & Turner, P. (2004). Intrusion Detection: Issues and Challenges in Evidence Acquisition. *International Review of Law Computers*, 149-164.

Center for Internet Security. (2021a). *CIS Controls V8.*

https://www.cisecurity.org/controls/v8/

Center for Internet Security. (2021b). 4 Reasons SLTTs use Network Monitoring Systems.

*https://www.cisecurity.org/blog/4-reasons-sltts-use-network-monitoring-systems/*

Center for Internet Security. (2018). Albert: A Smart Solution for Network Monitoring.

https://www.cisecurity.org/blog/albert-a-smart-solution-for-network-monitoring/

Einay, S., Oz, C., & Navaei, Y. D. (2021). The Anomaly- and Signature-Based IDS for Network

Security Using Hybrid Inference Systems. *Mathematical Problems in Engineering*.

Emerging Threats. (2013) What Every IDS User Should Do.

https://doc.emergingthreats.net/bin/view/Main/WhatEveryIDSUserShouldDo

Emerging Threats. (2012) About Emerging Threats.

https://doc.emergingthreats.net/bin/view/Main/AboutEmergingThreats

Ganesan, A., Parameshwarappa, P., Peshave, A., Chen, Z., & Oates, T. (2019) Extending

Signature based Intrusion Detection Systems with Bayesian Abductive Reasoning https://arxiv.org/abs/1903.12101

IC3. (2021). *Internet Crime Report 2020:*

https://www.ic3.gov/Media/PDF/AnnualReport/2020\_IC3Report.pd*f*

https://doi.org/10.1016/S1361-3723(21)00038-5

OISF. (2021). Suricata User Guide

https://buildmedia.readthedocs.org/media/pdf/suricata/latest/suricata.pdf

OISF & AND. (2017) Intrusion Detection with Suricata [video] networkdefense.io.

https://www.networkdefense.io/library/intrusion-detection-with-suricata/

Papadogiannaki, E., & Ioannidis, S. (2021, February 6). Acceleration of Intrusion Detection in Encrypted Network Traffic Using Heterogeneous Hardware. *Sensors*. https://doi.org/10.3390/s21041140

Proofpoint. (2020) Proofpoint ET Pro Ruleset.

https://www.proofpoint.com/sites/default/files/pfpt-us-ds-et-pro-ruleset.pdf

Rice, B. (2014). Automated Snort Signature Generation [Master's Theses, James Madison

University]. JMU Scholarly Commons. https://commons.lib.jmu.edu/cgi/viewcontent.cgi?article=1314&context=master201019

Sanders, C. (2017). Practical Packet Analysis (3rd ed.). No Starch Press

https://doi.org/10.1016/S1353-4858(17)30046-6

Sanders, C., & Smith, J. (2014) Applied Network Security Monitoring. Syngress

https://doi.org/10.1016/b978-0-12-417208-1.00001-5

Scarfone, K., & Mell, P. (2007, February). *NIST SP 800-94.* Fhttps://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-94.pdf

Shipulin, K. (2018). We need to talk about IDS signatures. *Network Security*(3), 8-13. https://doi.org/10.1016/S1353-4858(18)30024-2

Sikorski, M., & Honig, A. (2012). Practical Malware Analysis. No Starch Press.

Vacca, J. R. (2014). Network and Systems Security (2nd ed). Syngress.

https://doi.org/10.1016/C2011-0-08779-3

# Appendix A

suri-rule-gen.py Source Code

#!/usr/bin/env python3

# Automating Steps of Suricata Rule-Writing

# by

# Brian T. Carr

# A Capstone Submitted to the Faculty of

# Utica College

# August 2021

# in Partial Fulfillment of the Requirements for the Degree of

# Master of Science in Cybersecurity

#

# email: brian@briantcarr.com

# website: briancarr.org

import os

import argparse

import re

import logging

from types import FrameType

#output file

outfile = 'suri-rule-gen.rules'

# Header Variables

rule\_action = "alert"

protocol = "ip"

source\_ip = "any"

source\_port = "any"

direction = "->"

dest\_ip = "any"

dest\_port = "any"

#option\_variables

message= "!!!Describe The Rule Here!!!"

rev='rev:001'

sid='sid:000001'

list\_of\_vars\_in\_options = [sid, rev]

#logging configuration

logger = logging.getLogger('suri-rule-gen.py')

logger.setLevel(logging.DEBUG)

# create file handler which logs even debug messages

fh = logging.FileHandler('suri-rule-gen.log')

fh.setLevel(logging.DEBUG)

# create console handler with a higher log level

ch = logging.StreamHandler()

ch.setLevel(logging.ERROR)

# create formatter and add it to the handlers

formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s', datefmt='%Y-%m-%d %H:%M:%S')

ch.setFormatter(formatter)

fh.setFormatter(formatter)

# add the handlers to logger

logger.addHandler(ch)

logger.addHandler(fh)

# 'application' code

#Regular Expressions for input validation

ip\_pattern = re.compile("^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$|^[\!]\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$|^\$HOME\_NET|^\$EXTERNAL\_NET|^\[|any|^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}\/\d{1,3}$|^\$EXT\_NET")

port\_pattern = re.compile("^\d{1,6}|^\[\d{1,6}|^any|^!d{1,6}|^!\d{1,6}")

# Lists for Input Validation

action\_options = ['alert', 'pass', 'drop', 'reject', 'rejectsrc', 'rejectdst', 'rejectboth']

proto\_options = ['tcp', 'udp', 'icmp', 'ip', 'http', 'ftp', 'tls', 'smb', 'dns', 'dcerpc',

'ssh', 'smtp', 'imap', 'modbus', 'dnp3', 'enip', 'nfs', 'ike', 'krb5', 'ntp'

'dhcp', 'rfb', 'rdp', 'snmp', 'tftp', 'sip', 'http2']

direction\_options = ['->','<>']

target\_options = ['src\_ip', 'dest\_ip']

ipopts\_options = ['rr', 'eol', 'nop', 'ts', 'sec', 'esec', 'lsrr', 'ssrr', 'satid', 'any']

# CLI arguments

parser=argparse.ArgumentParser()

parser.add\_argument('--action', action="store", type=str, help="Use to set rule action - Default is alert")

parser.add\_argument('--protocol', action="store", type=str, help="Use to set protocol.")

parser.add\_argument('--sip', action="store", type=str, nargs='+', help="Use to set source IP for rule. Format examples: | 10.0.0.0 | 10.0.0.0/8 | !10.0.0.0 | [10.0.0.0, 192.168.0.0/24, !172.16.0.0]")

parser.add\_argument('--srcport', action="store", type=str, nargs='+', help="Use to set the source port. Format exampels: | 80 | [80,81,82] | [8080:] | !80 | [1:80,![2,4]]")

parser.add\_argument('--direction', action='store', type=str, help="Use to se the direction of the rule valid options inclde: ")

parser.add\_argument('--dip', action="store", type=str, nargs='+', help="Use to set destination IP for rule. Format examples: | 10.0.0.0 | 10.0.0.0/8 | !10.0.0.0 |[10.0.0.0, 192.168.0.0/24, !172.16.0.0]")

parser.add\_argument('--destport', action="store", type=str, nargs='+', help="Use to set the destination port. Format exampels: 80 | [80,81,82] | [8080:] | !80 | [1:80,![2,4]]")

parser.add\_argument('--message', action="store", type=str, nargs='+', help="Use to set a descriptive message about the rule.")

parser.add\_argument('--meta', action="store", nargs='+', type=str, help="Used to set metadata variables. Be careful with formatting. sample format: --meta key value | --meta key value, key value")

parser.add\_argument('--ttl', action="store", type=str, help="Use to set TTL value. Format: number")

parser.add\_argument('--outfile', action="store", type=str, help="used to specify a file to use instead of suri-rule-gen.rules. MUST END IN .rules." )

parser.add\_argument('--rev', action="store", type=str, help="Use to specify Revision Number.")

parser.add\_argument('--sid', action="store", type=str, help="Use to specify Signature Identification.")

parser.add\_argument('--content', action="store", type=str, nargs='+', help="Used to specificy payload content.")

parser.add\_argument('--classtype', action="store", type=str, help="Use to set classtype.")

parser.add\_argument('--gid', action="store", type=str, help="Use to set group ID.")

parser.add\_argument('--urlref', action="store", type=str, help="Used to set URL reference. Format: format.com")

parser.add\_argument('--cveref', action="store", type=str, help="Use to set CVE reference. Format: CVE-2021-1234")

parser.add\_argument('--priority', action="store", type=str, help="Use to set the rule priorty. Format: 1")

parser.add\_argument('--ja3', action="store", type=str, help="Use to set JA3 hash value, JA3 strings require md5 hashing.")

parser.add\_argument('--ja3s', action="store", type=str, help="Use to set JA3S hash value, JA3S strings require md5 hashing.")

parser.add\_argument('--target', action="store", type=str, help="Use to set rule target.")

parser.add\_argument('--ipopts', action="store", type=str, help="Use to set IP Option.")

parser.add\_argument('--geoip', action="store", type=str, help="used to set Geo IP value.")

parser.add\_argument('--fragbits', action="store", type=str, help="Use to set fragbit values")

parser.add\_argument('--fragoffset', action="store", type=str, help="Use to set fragoffset values")

parser.add\_argument('--tos', action="store", type=str, help="use to set IP header TOS value.")

parser.add\_argument('--seq', action="store", type=str, help="Use to set TCP sequence number.")

parser.add\_argument('--tlssubject', action="store", type=str, help="use to tls cert subject")

parser.add\_argument('--tlsissuer', action="store", type=str, help="Use to set tls issuer value")

parser.add\_argument('--tlsserial', action="store", type=str, help='Use to set tls serial number')

parser.add\_argument('--tlsfingerprint', action="store", type=str, help="use to set tls cert SHA1 fingerprint")

parser.add\_argument('--pcre', action="store", nargs='+', type=str, help="use to set a Perl Compatible Regular Expresion value.")

parser.add\_argument('--sshproto', action="store", type=str, help="use to set ssh protocol version")

parser.add\_argument('--sshsoftware', action="store", type=str, help="use to set ssh software value")

parser.add\_argument('--hassh', action="store", type=str, help="use to set hassh value for client")

parser.add\_argument('--hasshstring', action="store", type=str, help="use to set hassh string value for client")

parser.add\_argument('--hasshserver', action="store", type=str, help="use to set hassh value for server")

parser.add\_argument('--hasshserverstring', action="store", type=str, help="use to set hassh string value for server")

logger.info('\*\*\*\*\*\*\*\*\*\*NEW RULE BEING GENERATED\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

#turn cli args into arg.<argument>

args = parser.parse\_args()

# Set rule sid and rev value, set first to populate log properly.

while True:

if args.sid is not None:

list\_of\_vars\_in\_options.pop(0)

sid = 'sid:'+args.sid

list\_of\_vars\_in\_options.insert(-2, sid)

break

else:

logger.warning('No sid value set!!! Please ensure that no rules have duplicate sid values.')

break

while True:

if args.rev is not None:

list\_of\_vars\_in\_options.pop(-1)

rev = 'rev:'+args.rev

list\_of\_vars\_in\_options.insert(-1, sid)

break

else:

logger.warning('No rev value set!!! Please ensure that there are no duplicate rev values with a common sid value.')

break

generated\_rule\_sid = 'Generated rule ' + sid + ' ' + rev + ' - '

#While loops to test the presence of CLI arguements

while True:

if args.action is not None:

if args.action.lower() in action\_options:

rule\_action = args.action.lower()

logger.info(generated\_rule\_sid +' Rule action set: ' + rule\_action)

break

else:

print("invalid selection")

logger.error(generated\_rule\_sid + ' Invalid action entered: ' + args.action)

break

else:

#print('no action selected')

logger.info(generated\_rule\_sid+' No action value entered.')

break

while True:

if args.protocol is not None:

if args.protocol.lower() in proto\_options:

protocol = args.protocol

logger.info(generated\_rule\_sid +'Rule protocol set: ' + protocol)

break

else:

print('invalid protocol selected valid protocol\n' + str(proto\_options))

logger.error(generated\_rule\_sid + ' invalid protocol value entered: ' + args.protcol )

break

else:

# print('no protocol selected')

logger.info(generated\_rule\_sid + ' no protocol value entered.')

break

while True:

if args.sip is not None:

test\_source\_ip = ' '.join(args.sip)

if ip\_pattern.match(test\_source\_ip) is not None:

source\_ip = test\_source\_ip

break

else:

logger.error('Invalid Source IP entered: ' + test\_source\_ip)

print('!!!!!Invalid Source IP entered!!!!!')

logger.error(generated\_rule\_sid+ ' invalid source IP entered: ' )

break

else:

#print('no Source IP specified with --sip')

break

while True:

if args.srcport is not None:

test\_source\_port = ' '.join(args.srcport)

if port\_pattern.match(test\_source\_port):

source\_port = test\_source\_port

logger.info(generated\_rule\_sid + ' source port value:' + source\_port)

break

else:

print("The port was not entered in the correct format.")

logger.error(generated\_rule\_sid+ ' invalid source port entered')

break

else:

# print('no Source port was specified with --srcport')

logger.info(generated\_rule\_sid + 'no value source port value entered.')

break

while True:

if args.direction is not None:

if args.direction in direction\_options:

direction = args.direction

logger.info(generated\_rule\_sid + 'direction set to:' + args.direction)

break

else:

print('invalid direction selected: please use \n \<\> or \-\> when using a bash terminal')

logger.error(generated\_rule\_sid + ' invalid rule direction ' + args.direction )

break

else:

logger.info(generated\_rule\_sid + 'no rule driection specified.')

break

while True:

if args.dip is not None:

test\_dest\_ip = ' '.join(args.dip)

if ip\_pattern.match(test\_dest\_ip):

dest\_ip = test\_dest\_ip

logger.info(generated\_rule\_sid + 'dest ip set to: ' + dest\_ip)

break

else:

logging.error(generated\_rule\_sid + 'destination IP set to invalid value: ' + test\_dest\_ip)

break

else:

#print('No Dest IP specified.')

logging.info(generated\_rule\_sid+ 'No dest ip specified, using any. ')

break

while True:

if args.destport is not None:

test\_dest\_port = ' '.join(args.destport)

if port\_pattern.match(test\_dest\_port):

dest\_port = test\_dest\_port

logger.info('Generated rule sid:' + sid + 'dest port set to:' + dest\_port)

break

else:

logger.error('Generated rule sid:' + sid + 'dest port entered incorrectly: ' + dest\_port)

break

else:

#print('no dest port specified')

logger.info(generated\_rule\_sid + 'no dest port specified specified.')

break

#Rule options start here

while True:

if args.message is not None:

message = " ".join(args.message)

logger.info(generated\_rule\_sid + 'Message value set: ' + message)

break

else:

logger.warning(generated\_rule\_sid + ' No message value set!')

break

while True:

if args.meta is not None:

meta\_var\_constructor = " ".join(args.meta)

meta\_var = 'metadata: ' + meta\_var\_constructor

logger.info(generated\_rule\_sid + ' meta var set to: ' + meta\_var )

list\_of\_vars\_in\_options.insert(0, meta\_var)

break

else:

break

while True:

if args.ttl is not None:

ttl = 'ttl:' + args.ttl

list\_of\_vars\_in\_options.insert(0, ttl)

logger.info(generated\_rule\_sid + ' ttl value set: ' + ttl)

break

else:

break

while True:

if args.content is not None:

content\_constructor = ' '.join(args.content)

content = 'content:' + content\_constructor

list\_of\_vars\_in\_options.insert(0, content)

logger.info(generated\_rule\_sid+ ' content set to: ' + content)

break

else:

break

while True:

if args.classtype is not None:

classtype = 'classtype:'+args.classtype

list\_of\_vars\_in\_options.insert(0, classtype)

logger.info(generated\_rule\_sid + ' classtype set to:' + classtype)

break

else:

break

while True:

if args.urlref is not None:

ref = 'reference: url, ' + args.urlref

list\_of\_vars\_in\_options.insert(0, ref)

logger.info(generated\_rule\_sid + ' refernce set to: ' + ref)

break

else:

break

while True:

if args.cveref is not None:

ref = 'reference: cve, ' + args.cveref

list\_of\_vars\_in\_options.insert(0, ref)

logger.info(generated\_rule\_sid + 'CVE referenced: ' + ref)

break

else:

break

while True:

if args.priority is not None:

priority = 'priority:'+args.priority

list\_of\_vars\_in\_options.insert(0, priority)

logger.info(generated\_rule\_sid + 'priority set to: ' + priority)

break

else:

break

while True:

if args.ja3 is not None:

ja3 = 'ja3.hash; content:"' + args.ja3 + '"'

list\_of\_vars\_in\_options.insert(0, ja3)

logger.info(generated\_rule\_sid + 'ja3 value set to: ' + ja3)

break

else:

break

while True:

if args.ja3s is not None:

ja3s = 'ja3s.hash; content:"' + args.ja3s + '"'

list\_of\_vars\_in\_options.insert(0, ja3s)

logger.info(generated\_rule\_sid + 'ja3s value set to: ' + ja3s)

break

else:

break

while True:

if args.target is not None:

if args.target == 'src\_ip' or 'dest\_ip':

target = 'target:' + args.target.lower()

list\_of\_vars\_in\_options.insert(0, target)

logger.info(generated\_rule\_sid + ' Option value set: ' + target)

break

else:

logger.error(generated\_rule\_sid + 'Invalid target value entered:' + args.target)

break

else:

break

while True:

if args.ipopts is not None:

if args.ipopts.lower in ipopts\_options:

ipopts = 'ipopts: ' + args.ipopts.lower()

list\_of\_vars\_in\_options.insert(0, ipopts)

logger.info(generated\_rule\_sid + ' Option value set: ' + ipopts)

break

else:

logger.error(generated\_rule\_sid + 'Invalid ipopts value entered: ' + args.ipopts)

break

break

while True:

if args.geoip is not None:

geoip = 'geoip: ' + args.geoip

list\_of\_vars\_in\_options.insert(0, geoip)

logger.info(generated\_rule\_sid + ' Option value set: ' + geoip )

break

else:

break

while True:

if args.fragbits is not None:

fragbits = 'fragbits:' + args.fragbits

list\_of\_vars\_in\_options.insert(0, fragbits)

logger.info(generated\_rule\_sid + ' Option value set: ' + fragbits)

break

else:

break

while True:

if args.fragoffset is not None:

fragoffset = 'fragoffset:' + args.fragoffset

list\_of\_vars\_in\_options.insert(0, fragoffset)

logger.info(generated\_rule\_sid + ' Option value set: ' + fragoffset)

break

else:

break

while True:

if args.tos is not None:

tos = 'tos:' + args.tos

list\_of\_vars\_in\_options.insert(0, tos)

logger.info(generated\_rule\_sid + ' Option value set: ' + tos )

else:

break

while True:

if args.seq is not None:

seq = 'seq:' + args.seq

list\_of\_vars\_in\_options.insert(0, seq)

logger.info(generated\_rule\_sid + ' Option value set: ' + seq )

else:

break

while True:

if args.pcre is not None:

pcre = 'pcre:"' + " ".join(args.pcre) + '"'

list\_of\_vars\_in\_options.insert(0,pcre)

logger.info(generated\_rule\_sid + ' Option value set: ' + pcre)

break

else:

break

while True:

if args.tlssubject is not None:

tlssubject = 'tls.cert\_subject; content:"' + args.tlssubject + '"'

list\_of\_vars\_in\_options.insert(0, tlssubject)

logger.info(generated\_rule\_sid + ' Option value set: ' + tlssubject )

break

else:

break

while True:

if args.tlsissuer is not None:

tlsissuer = 'tls.cert\_isser; content:"' + args.tlsissuer + '"'

list\_of\_vars\_in\_options.insert(0, tlsissuer)

logger.info(generated\_rule\_sid + ' Option value set: ' + tlsissuer )

break

else:

break

while True:

if args.tlsserial is not None:

tlsserial = 'tls.cert\_serial; content:"' + args.tlsserial + '"'

list\_of\_vars\_in\_options.insert(0, tlsserial)

logger.info(generated\_rule\_sid + ' Option value set: ' + tlsserial )

break

else:

break

while True:

if args.tlsfingerprint is not None:

tlsfingerprint = 'tls.cert\_fingerprint; content:"' + args.tlsfingerprint + '"'

list\_of\_vars\_in\_options.insert(0, tlsfingerprint)

logger.info(generated\_rule\_sid + ' Option value set: ' + tlsfingerprint )

break

else:

break

while True:

if args.sshproto is not None:

sshproto = 'ssh.protoversion:' + args.sshproto

list\_of\_vars\_in\_options.insert(0, sshproto)

logger.info(generated\_rule\_sid + ' Option value set: ' + sshproto)

break

else:

break

while True:

if args.sshsoftware is not None:

sshsoftware = 'ssh.software; content"' + args.sshsoftware +'"'

list\_of\_vars\_in\_options.insert(0, sshsoftware)

logger.info(generated\_rule\_sid + ' Option value set: ' + sshsoftware)

break

else:

break

while True:

if args.hassh is not None:

hassh = 'ssh.hassh; content:"' + args.hassh + '"'

list\_of\_vars\_in\_options.insert(0, hassh)

logger.info(generated\_rule\_sid + ' Option value set: ' + hassh)

break

else:

break

while True:

if args.hasshstring is not None:

hasshstring = 'ssh.hassh.string; content:"' + args.hasshstring + '"'

list\_of\_vars\_in\_options.insert(0, hasshstring)

logger.info(generated\_rule\_sid + ' Option value set: ' + hasshstring)

break

else:

break

while True:

if args.hasshserver is not None:

hasshserver = 'ssh.hassh.server; content:"' + args.hasshserver +'"'

list\_of\_vars\_in\_options.insert(0, hasshserver)

logger.info(generated\_rule\_sid + ' Option value set: ' + hasshserver)

break

else:

break

while True:

if args.hasshserverstring is not None:

hasshserverstring = 'ssh.hassh.server.string; content:"' + args.hasshserverstring + '"'

list\_of\_vars\_in\_options.insert(0, hasshserverstring)

logger.info(generated\_rule\_sid+ ' Option set: ' + hasshserverstring)

break

else:

break

#Test if a different outfile should be used

while True:

if args.outfile is not None:

outfile = args.outfile

logger.info(generated\_rule\_sid + 'outfile set to: ' + outfile)

break

else:

break

# Constructing the final rule string

list\_of\_vars\_in\_header = [ rule\_action, protocol, source\_ip, source\_port, direction, dest\_ip, dest\_port]

message\_constructor = ' (msg:"' + message + '"'

list\_of\_vars\_in\_options.insert(0,message\_constructor)

new\_rule\_header = ' '.join(list\_of\_vars\_in\_header)

new\_rule\_options = '; '.join(list\_of\_vars\_in\_options) + ';)'

new\_rule = new\_rule\_header+new\_rule\_options

print(new\_rule)

# Check if outfile exists and create or write to it

if os.path.exists(outfile):

f = open(outfile, "a")

f.write(new\_rule + '\n')

else:

f = open(outfile, "x")

f.writelines("%s\n" % new\_rule)