# Tweaking Metasploit to Evade Encrypted C2 Traffic Detection

Gonçalo Xavier, Carlos Novo, Ricardo Morla FEUP and INESC TEC, University of Porto Porto, Portugal { up201604506, carlos.novo, ricardo.morla } @ fe.up.pt

Abstract—Command and Control (C2) communication is a key component of any structured cyber-attack. As such, security operations actively try to detect this type of communication in their networks. This poses a problem for legitimate pentesters that try to remain undetected, since commonly used pentesting tools, such as Metasploit, generate constant traffic patterns that are easily distinguishable from regular web traffic. In this paper we start with these identifiable patterns in Metasploit's C2 traffic and show that a machine learning-based detector is able to detect the presence of such traffic with high accuracy, even when encrypted. We then outline and implement a set of modifications to the Metasploit framework in order to decrease the detection rates of such classifier. To evaluate the performance of these modifications, we use two threat models with increasing awareness of these modifications. We look at the detection evasion performance and at the byte count and runtime overhead of the modifications. Our results show that for the second, increasedawareness threat model the framework-side traffic modifications yield a better detection avoidance rate (90%) than payloadside only modifications (50%). We also show that although the modifications use up to 3 times more TLS payload bytes than the original, the runtime does not significantly change and the total number of bytes (including TLS payload) reduces.

Index Terms—Command and Control, Adversarial learning, Penetration Testing

### I. INTRODUCTION

Pentesters put themselves in the shoes of attackers [1] to test the security of their client's systems. Remote control of compromised assets plays a major role in pentesting; this is achieved through the deployment of payloads and the communication between the payloads and the pentester's command and control software [2]. Detecting this communication is important for security operations, not as much to detect legitimate pentesters as to detect actual attackers that use pentesting tools [3]. This poses a challenge to the legitimate pentester - if C2 communications are blocked how can the pentester uncover vulnerabilities and other security issues elsewhere in the target system? IP-specific rules for the pentester could be enacted by the client's IT staff to allow the pentester's traffic. but this goes against the premise that a pentest is as good as the assumptions for the attacks are weak. Clients may not take seriously the security issues that the pentester finds if they think the attack is only possible because they allowed it. One way for the pentester to overcome these difficulties is to modify its C2 traffic in order to avoid detection.

In this paper we explore different ways in which the pentester can modify its traffic and what is the impact of those modifications in evading a detector. Pentesting C2 traffic can be distinguished from other, non-C2 traffic with machine learning techniques. Machine learning is especially relevant if the C2 traffic is encrypted and blacklisting server IPs or certificates is not viable because of inexpensive certificate replacement or cloud-based server-side IP sharing [4]. Different pentesting frameworks are likely to have different C2 traffic profiles that a machine learning-based detector may need to learn to distinguish from normal traffic. An approach for C2 traffic communication is to have the payload periodically open a connection to the server, inquiring for orders or reporting results. This ends up creating a regular traffic pattern that can be much different from web and other, non-C2 traffic - and makes it relatively easy to detect. This is the case of Metasploit<sup>1</sup>, one of the best known pentesting tools. Armed with traffic captures of C2 traffic from a specific tool like Metasploit and of non-C2 traffic from their local networks, security operations are able to train a machine-learning based detector of C2 traffic for their networks. In section IV we show results confirming that for a specific dataset with Metasploit traffic it is possible to train a detector with extremely high performance.

Knowing that its C2 traffic is regular and may be easily detected, the pentester may feel the need to change the traffic pattern to evade a detector. In section V we describe a set of traffic pattern changes to C2 traffic and show how the Metasploit Framework can be modified in order to generate such traffic and evade a machine learning-based detector. We then consider a second threat model in section VI, where the target's network detection system is aware of the modifications made to the C2 traffic and the pentester employs adversarial learning techniques [5] in order to evade the detector. We present detector evasion results and discuss the overhead and performance impacts associated with the framework modifications.

# II. RELATED WORK

The communication patterns of open-source C2 frameworks have been studied before. For example, [6] highlights common practices and communication behaviours present in some of the most popular pentesting frameworks, and shows how a machine learning model could be trained based on these

https://github.com/rapid7/metasploit-framework/

behaviours to distinguish each framework's traffic. Additionally, and specifically focusing on Metasploit, [7] studied the entire behaviour of a Metasploit payload inside a victim's host, focusing on C2 communication patterns and on the presence of the payload in the victim's host memory. Although these studies are important for improving the performance of intrusion detection systems (IDS's), they fail to inform a legitimate pentester on how to overcome such identifiable characteristics and evade detection systems. Additionally, due to the active and open-source development of such frameworks, these studies have quickly become outdated and no longer accurately describe the current implementation of these tools.

Evading detection systems has been a concern for the Metasploit's development team, as defenders more quickly react to new threats by automatically analyzing and updating detection rules [8]. In 2018, the Metasploit team incorporated evasion techniques like code obfuscation, code stub injection and encryption in the Metasploit Framework modules [9]. Casey et al. [10] studied the performance of these modules in evading commonly used anti-virus software and showed that most anti-viruses are successful at detecting them, likely due to the public availability of the documentation of these modules. The techniques used in these evasion modules are targeted at evading static signature-based antivirus checks like binary file analysis and they don't change the C2 patterns that can give away the presence of pentesting tool traffic on the network.

Other studies have focused on the use of adversarial learning techniques against machine learning-based detection systems, specially targeting malware's traffic characteristics. For example, [11] showed how a generative adversarial network (GAN) could be used to adapt the inter-packet time and packet length variance of botnet attacks, in order to bypass an IDS model. Based on the same type of traffic features, Yang et al. [12] showed the performance of different adversarial algorithms against an IDS, in a black-box scenario and the authors of [13] have proposed multiple defense methods against adversarial attacks targeting intrusion detection systems. In a more C2specific work, Rigaki et al. [14] showed how by adapting a Remote Access Trojan (RAT) C2 communication scheme, according to a GAN, it was possible to mimic the traffic of a legitimate application, thus evading a detector. Although these studies contribute to highlighting the weakness of IDSs against adversarial examples, they all focus on incorporating adversarial learning in malware samples, not on pentesting tools. Specifically, these studies don't focus on adapting source code such that an IDS that focuses on encrypted traffic can be evaded.

Another conceptually similar topic is the one of censorship circumvention, where an actor masks traffic features by mimicking allowed traffic in an attempt to bypass the censor's detector. The authors of [15] have used a GAN for automatic traffic generation and censor circumvention, and advancements made in this research field may generally be used by pentesters and vice versa.

#### III. METASPLOIT C2 TRAFFIC PATTERNS

During a normal Metasploit exploit process the deployed payload and the Metasploit framework have to communicate. When using a stageless payload this communication is bootstrapped as follows:

- The exploit occurs and the payload starts on the victim's host:
- The payload's dispatch loop starts and the payload reaches out to the framework on a specific host and HTTP URL;
- 3) The Metasploit framework on the pentester host recognizes this request as a new session and negotiates TLV C2 encryption keys. Afterwards, any communication for this session between the payload and the Metasploit Framework consists of GET or POST requests with encrypted data.

In our setup we configure the Mettle payload (Mettle is the C-based Metasploit payload<sup>2</sup>) to use HTTPS (HTTP over TLS), which additionally secures the bootstrapping and prevents direct eavesdropping of e.g. the type and headers of the HTTP request that the payload issues. When using this type of payload, the Metasploit framework communicates using packets that follow the structure presented in Figure 1, with the encrypted data transmitted between the Metasploit Framework and the payload encapsulated in the TLV layer of each packet. As the Metasploit framework does not internally differentiate between HTTP and HTTPS traffic, its communication workflow is the same whether using HTTP or HTTPS.

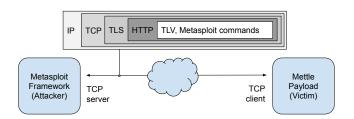


Fig. 1: Metasploit's packet structure when using a HTTPS payload.

Figure 2 shows the payload's communication workflow after a secure communication session has been established. The payload's dispatch loop repeatedly contacts the framework in a pooling manner, using empty HTTP GET requests on a new URL generated by the framework. The framework responds to each of these GET requests by sending any queued commands back to the payload. These commands are then individually processed and their results returned using POST requests back to the framework. If no commands are returned to the payload, the interval between the pooling requests doubles up to a maximum of 10 seconds.

This pooling pattern generates very consistent and periodic traffic not commonly present in regular web traffic. Mettle

<sup>&</sup>lt;sup>2</sup>https://github.com/rapid7/mettle

uses the *libcurl* library to execute its HTTP requests. *curl* configures requests including custom options, headers, and payload through the *handle* programming construct. Every time a requests is made and its response is received, Mettle terminates the corresponding *handle*. This automatically causes a new TCP connection to be initiated and terminated whenever the payload needs to communicate with the framework. This can be seen in Figures 3 and 4.

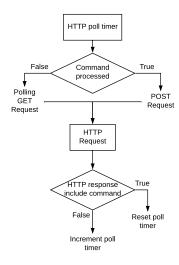


Fig. 2: Mettle HTTP communication scheme

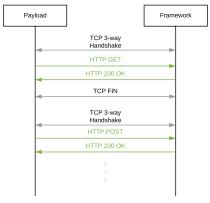


Fig. 3: Diagram for a traffic flow sample between HTTP payload and the Metasploit framework

Time	10.1.	1.131	1.101	Comment
4.982268419	39928	39928 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=14	8080	TCP: 39928 → 8080 [SYN] Seq=0 Win=64240 Len=
4.982290525	39928	8080 → 39928 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len	8080	TCP: 8080 → 39928 [SYN, ACK] Seq=0 Ack=1 Win
4.982552148	39928	39928 → 8080 [ACK] Seq=1 Ack=1 Win=64256 Len=0 T.	8080	TCP: 39928 → 8080 [ACK] Seq=1 Ack=1 Win=6425
4.982552195	39928	GET /alQivKdRFqFsa2ppDMi9vAh4ukD_de8v4Ux/ HTTP/1.1	8080	HTTP: GET /alQivKdRFqFsa2ppDMi9vAh4ukD_de8v
4.982578132	39928	8080 → 39928 [ACK] Seq=1 Ack=276 Win=64896 Len=0	8080	TCP: 8080 → 39928 [ACK] Seq=1 Ack=276 Win=64
4.983062021	39928	HTTP/1.1 200 OK	8080	HTTP: HTTP/1.1 200 OK
4.983294642	39928	39928 → 8080 [ACK] Seq=276 Ack=549 Win=63744 Len.	8080	TCP: 39928 → 8080 [ACK] Seq=276 Ack=549 Win=
4.983819416	39928	39928 → 8080 [FIN, ACK] Seq=276 Ack=549 Win=64128	0000	TCP: 39928 → 8080 [FIN, ACK] Seq=276 Ack=549
4.983883683	39928	8080 → 39928 [FIN, ACK] Seq=549 Ack=277 Win=64896	8080	TCP: 8080 → 39928 [FIN, ACK] Seq=549 Ack=277
4.984129395	39928	39928 → 8080 [ACK] Seq=277 Ack=550 Win=64128 Len.	8080	TCP: 39928 → 8080 [ACK] Seq=277 Ack=550 Win=
5.482585007	39930	39930 → 8080 [SYN] Seq=0 Win=64240 Len=0 MSS=14	8080	TCP: 39930 → 8080 [SYN] Seq=0 Win=64240 Len=
5.482604329	39930	8080 → 39930 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len	8080	TCP: 8080 → 39930 [SYN, ACK] Seq=0 Ack=1 Win
5.482821974	39930	39930 → 8080 [ACK] Seq=1 Ack=1 Win=64256 Len=0 T.	8080	TCP: 39930 → 8080 [ACK] Seq=1 Ack=1 Win=6425
5.482822011	39930	POST /alQivKdRFqFsa2ppDMi9vAh4ukD_de8v4Ux/ HTTP/1.	8080	HTTP: POST /alQivKdRFqFsa2ppDMi9vAh4ukD_de8
5.482838895	39930	8080 → 39930 [ACK] Seq=1 Ack=649 Win=64512 Len=0	8080	TCP: 8080 → 39930 [ACK] Seq=1 Ack=649 Win=64
5.483494383	39930	HTTP/1.1 200 OK	8080	HTTP: HTTP/1.1 200 OK

Fig. 4: Wireshark capture for a traffic flow sample between HTTP payload and the Metasploit framework

As a consequence of this behaviour, when using an HTTPS payload, the traffic flow between Metasploit and the payload rarely exceeds more than two TLS Application Data records (type 23, AppData). This stands out when compared to non-C2, web traffic. Table I highlights this characteristic by grouping all TLS records belonging to the same TCP connection and then filtering them by their TLS record type. C2 traffic only shows 1 HTTP request per connection, which produces 2 TLS AppData records – one in each direction. Web traffic shows several TLS AppData records exchanged during the same TCP connection.

1	2	3	4	5	6	7	8	9
396	5698	454	3552	452	16408	16408	16408	16408
32	560	1280	32	528	512	32	560	1280
448	1494	453	2644	452	16408	16408	16408	16408
32	560	1280	32	528	512	32	528	512
288	288							
704	176							
288	288							
624	176							

TABLE I: First 9 TLS Application Data record sizes for C2 traffic and non-C2 (web) traffic for sample TCP connections. Top: non-C2 traffic, bottom: C2 traffic.

The communication workflow we describe in this section is for the standard Metasploit and Mettle versions available online. In sections V and VI we show how to modify Mettle and Metasploit implementations to change traffic patterns while keeping the same workflow and evading a detector. Before that, in section IV, we show how the standard communication pattern is easily detectable from non-C2 traffic without these modifications.

## IV. METASPLOIT C2 TRAFFIC DETECTION

The model we use to detect C2 traffic takes as input a sequence with the sizes of the first 20 TLS AppData records in both directions of a TCP connection. These sequences of sizes are extracted from traffic captures using Cisco's Joy<sup>3</sup>

<sup>&</sup>lt;sup>3</sup>https://github.com/cisco/joy

tool. These features are then filtered by their TLS content type and the length of the AppData records is fed to the model. We pad flows with fewer than 20 TLS Appdata records using the value -1 and truncate flows with more than 20 TLS Appdata records.

Internally, the model consists of a multi-layer neural network with an input layer of 20 nodes (for the first 20 TLS AppData records), 3 "hidden layers" of 2048, 1024 and 512 nodes respectively, all with a *relu* activation function, and a final output layer with 2 nodes for categorical classification with a *softmax* activation function. In each layer, we add a 20% dropout layer to prevent overfitting. The *softmax* function used in the output layer returns a 2-dimensional probability distribution vector on whether the input sequence is considered a C2 traffic flow or not. The Adam optimizer was used for updating weights during training and we use categorical crossentropy as the loss function.

The model was trained and evaluated using a dataset of locally generated Metasploit C2 traffic together with a dataset of non-C2 traffic. Our locally generated Metasploit C2 traffic consists of a series of Metasploit commands typically used during a pentest, such as <code>sysinfo</code>, <code>ps</code>, <code>ipconfig</code>, <code>route</code>, <code>download /etc/shadow</code>, <code>ls</code>, that are invoked automatically once the payload contacts Metasploit. We repeatedly spawn payloads on a virtual machine and capture the traffic the payloads generate towards the Metasploit host. Our non-C2 traffic was generated using BrowserTime<sup>4</sup> and consists of 10 visits to each of the top 1k Alexa web sites<sup>5</sup>.

We achieved a detection accuracy of 99% on 111k balanced C2 and non-C2 samples. This high accuracy confirms our intuition that Metasploit's C2 traffic is highly identifiable and that a pentester could be easily detected when using this framework.

# V. EVADING A REGULAR METASPLOIT DETECTOR

The detector model developed in section IV is tested with Metasploit C2 samples generated from the standard Metasploit framework. Our first new threat model considers that a pentester is smart enough to attempt to evade such a detector by modifying its C2 traffic.

Threat model #1: A C2 traffic detector, aware of the traffic profiles generated by regular web traffic and the standard Metasploit framework, is able to consistently detect the presence C2 traffic. Knowing about this, and to remain undetected, a pentester is able to modify the communication scheme of the Metasploit framework in two ways: 1) increase the size of TLS records (section V-A), and 2) change the number of HTTP requests used within the same TCP connection (section V-B). We describe both approaches and their evasion results next.

### A. Packet Stuffing

A noticeable aspect of C2 traffic is the small size of TLS records when compared to web traffic. To try and change this

characteristic, we develop a generic packet stuffing method. This method operates on the "HTTP Request" block of the Mettle communication scheme (as described in Figure 2) and adds a stuffing HTTP header to each HTTP request. This has the indirect effect of increasing the size of the TLS records analysed by the detector's model. It also does not change the functionality of the framework. Figure 5 shows how this stuffing occurs on actual packets. By applying a similar method on the framework-side it is possible to achieve packet stuffing in both sides of the communication.

To evaluate the evasion rate of this technique using the same methodology as in section IV, we generated two datasets of C2 traffic, one with constant packet stuffing of 50 bytes and one with random size packet stuffing. In both datasets packet stuffing occurs in the packets from the framework and from the payload. We then assess the datasets' evasion rate. We define evasion rate as the percentage of C2 traffic samples that were misclassified by the detector. Table II shows the evasion rate for regular Metasploit traffic and for 50 byte and random stuffing traffic.

Fig. 5: HTTP header used in packet stuffing

Dataset type	Total number of samples	Evasion rate
Regular Metasploit	111853	9 (0.01%)
50 byte Stuffing	67513	3259 (4.83%)
Random byte size Stuffing	66768	66680 (99.87%)

TABLE II: Metasploit C2 evasion rate results (default vs. packet stuffing)

After implementing this technique, when using 50 byte length headers as stuffing, the pentester would only be able to achieve a slightly better evasion rate as when using the unchanged Metasploit tool. However, when using random length headers, the pentester can almost completely evade the detector. Although this technique alone might be enough to trick a detector aware of the traffic generated by the standard Metasploit Framework, it does not alter the other identifiable characteristic of the Metasploit framework: the number of HTTP requests per TCP connection; we consider this next.

## B. HTTP Request per TCP Connection

As shown in Figures 3 and 4, in Table I, and as discussed in section III, the immediate termination of the *curl handle* causes very short TLS Application Data flows across all Metasploit traffic. This could be seen as the most distinguishable characteristic in Metasploit's C2 traffic. Several TLS

<sup>&</sup>lt;sup>4</sup>https://github.com/sitespeedio/browsertime

<sup>&</sup>lt;sup>5</sup>http://s3.amazonaws.com/alexa-static/top-1m.csv.zip

AppData packets can be included in the same TCP connection in order to make C2 traffic less distinguishable from web traffic, which typically has TCP connections with more TLS records. To achieve this, the *curl handles* are now re-used until a number of Metasploit HTTP requests have been exchanged in the same TCP connection.

Figure 6 shows the difference in Metasploit's TLS traffic flow with adaptation V-B when compared to Figure 3. We can now observe multiple HTTP requests in a single TCP connection. Although we changed the way HTTP requests are organized into TCP connections, the overall workflow remains the same and the GET and POST requests that can be observed correspond to polling and command response as in the default Metasploit payload.

Table III shows the impact adaptation V-B has on the sequence of TLS AppData record sizes per TCP connection. We observe this modification causes the Metasploit to generate a traffic profile that to a certain extent is closer to that of non-C2, web traffic shown in Table I.

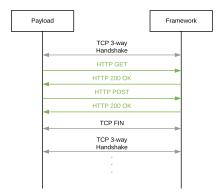


Fig. 6: Diagram for a traffic flow sample between HTTP payload and the Metasploit framework with adaptation V-B.

1	2	3	4	5	6	7	8	9
336 320	272 112	320						
320 320 320	560 304	768 512	176 176	784 528	288 304	560 720	176 176	
320	304	1056	176	1072	288	720	170	
528 576	176 176	544 592	288 288					
624	176							

TABLE III: First 9 TLS Application Data record sizes for sample TCP connections and C2 traffic using adaptation.

We evaluate the evasion rate of this technique using a C2 Metasploit dataset with 3 HTTP requests per TCP connection and with a random number of HTTP requests per connection. Table IV shows the evasion rate results for these techniques. By changing the amount of HTTP requests per TCP connection, a pentester can completely evade what was previously considered a reliable detector.

Dataset type	Total number of samples	Evasion rate		
Default Metasploit	111853	9 (0.01%)		
3 HTTP requests per TCP connection	41094	41094 (100%)		
Random # HTTP requests per TCP connection	12415	12415 (100%)		

TABLE IV: Evasion rate of multiple HTTP requests sent in the same TCP connection

After noticing this clear improvement in his evasion rate, the pentester can modify Metasploit to start using a random number of TLS AppData packets per TCP connection, circumventing the periodic pattern that lead to the detection of C2 traffic in the first place.

#### VI. EVADING AN INCREASED AWARENESS DETECTOR

In order to further analyse the threat model presented in section V, a more robust model, aware of the previous changes made to the Metasploit's communication scheme, was trained and evaluated. This model's layers and input features are the same as the one presented in section IV, however the training data now includes samples from the dataset "Random # HTTP requests per TCP Connection" discussed in section V-B. As a result of this addition, the detector is now able to detect Metasploit's C2 traffic with multiple HTTP requests per TCP connection with an accuracy of 98.4%. With this model in place, we consider a second threat model, as follows.

Threat model #2: A new C2 detector is trained with Metasploit traffic samples containing a random number of HTTP requests per TCP connection, as described in the beginning of this section. Knowing about this, and to remain undetected, the pentester is able to additionally modify the communication pattern of the Metasploit framework from section V.B by adding stuffing bytes to the C2 traffic that are specifically chosen to mislead the detector through adversarial machine learning techniques.

Adversarial machine learning is a technique that tries to trick machine learning models into wrongly classifying specially generated inputs. By taking advantage of the model's properties, these inputs only need to slightly differ from normal ones in a specific way in order to be incorrectly classified. In this paper, we use the Fast Gradient Sign Method [16], an adversarial type of attack that changes the value of each input feature by a fixed  $\epsilon$  in the direction of the detector's loss gradient,  $x^* = x + \epsilon sign(\nabla_x J(\theta, x, y))$ , thus attempting to achieve a wrong classification. This attack is implemented using the CleverHans [17] library.

# A. Adversarial Packet Stuffing

By using the "Random # HTTP request per TCP Connection" dataset as the original input for the CleverHans FGSM

attack, we are able to generate a sequence of TLS AppData packet lengths that is likely to trigger a non-C2 classification from the C2 traffic detector, while still following the profile of C2 traffic. By then combining the *Packet Stuffing* and the *Multiple HTTP Request per TCP connection* modifications, it is possible to implement this sequence in real C2 traffic. We add a number of stuffing bytes equal to the difference between the recommended adversarial size and the size of the HTTP request content; if the latter is larger than the former we do not add any stuffing to the HTTP request.

Adversarial packet stuffing can happen in two different scenarios:

- One-side Adversarial Stuffing: To keep the payload's size
  to a minimum and avoid placing the entire adversarial
  sequence in a binary file inside the victim's host that
  may be subject to reverse engineering, the pentester
  only implements the adversarial sequence produced by
  CleverHans in the framework-side packets. In this case,
  the payload communicates normally, without any type of
  packet stuffing. This approach has the downside of only
  generating one half of the adversarial sequence (from
  framework to payload).
- Two-side Adversarial Stuffing: While still trying to minimize the payload's size and not expose the full sequence of adversarial samples to eventual reverse engineering, the pentester modifies the framework-payload communication method to send the adversarial stuffing sizes to the payload. We describe this communication method in section VI-B. This way, the entire adversarial sequence can be executed on both sides of the communication without encoding the sequence in the payload.

In both scenarios the number of TLS Application Data packets exchanged during the same TCP connection is defined by the length of the adversarial sequence used.

## B. Framework-to-payload Stuffing Protocol

Using a similar method as the one described in section V-A, we modified the Metasploit framework to include an HTTP header that informs the payload of what the next adversarial stuffing size should be. Using the received adversarial size with the *Packet Stuffing* method, the payload executes its part of the adversarial stuffing sequence. Figure 7 shows a Wireshark capture with an example of the stuffing protocol header highlighted.

In the last framework-side request of an adversarial sequence, the framework signals the payload to terminate its curl handle by changing the Connection HTTP header, from "Keep-alive" to "close". In that last request, the framework informs the payload of which value it should use for the stuffing size of the first TLS AppData record in the next TCP connection. This way, the first value of new framework-payload TCP connections can also be part of an adversarial sequence.

- Frame 3: 266 bytes on wire (2128 bits), 266 bytes captured (2128 bits) on interface eth0, id 0 Ethernet II, Src: 72:62:60:7e:8d:8b (72:62:60:7e:8d:8b), Dst: b6:32:36:9c:2b:90 (b6:32:36:9c:2b:90)
- Transmission Control Protocol, Src Port: 8080, Dst Port: 49506, Seq: 1, Ack: 1550, Len: 200 Hypertext Transfer Protocol
  - HTTP/1.1 200 OK\r\n Content-Type: application/octet-stream\r\n

  - Server: Apache\r\n
    > Content-Length: 0\r\n
    \r\n

[HTTP response 1/1] [Time since request: 0.000475570 seconds]

Fig. 7: HTTP header used in framework-side packets during adversarial packet stuffing

#### C. Evasion Rate Results

We evaluate the evasion rate of the adversarial packet stuffing technique against the C2 detector of section VI using a C2 Metasploit dataset with Adversarial Packet Stuffing. Table V shows our results. By incorporating adversarial techniques into the Metasploit Framework, a pentester can significantly avoid detection even if the detector is aware of the modification techniques made to the original Metasploit C2 traffic. We observe that when the technique includes adversarial stuffing from the framework side the detection avoidance rate is high at around 90%, whereas if the adversarial stuffing is only executed at the payload side the avoidance drops to 50%. This may be explained by the much more regular pattern of traffic going from the framework to the payload when compared with the traffic from the payload, as shown with an example in Figure 8. Executing adversarial stuffing on the payload side in addition to adversarial stuffing on the framework side does not seem to significantly increase avoidance.

Dataset type	Total number of samples	Evasion rate
Framework-side only	14194	12976 (91.42%)
Payload-side only	14986	7720 (51.51%)
Two-side	14185	12983 (91.53%)

TABLE V: Metasploit C2 evasion rate results using adversarial packet stuffing

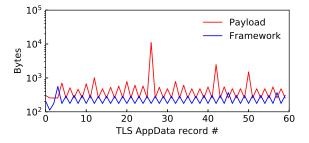


Fig. 8: TLS AppData record sizes for a sequence of 12 different Metasploit commands, split by direction: from the payload and from the framework.

Figure 9 helps us understand why our adversarial stuffing approach performs well against the increased awareness de-

tector. The detector learns to distinguish BrowserTime web traffic (on the left column in Figure 9) from a mix of regular traffic and traffic with a random number of HTTP requests per TCP connection (regular and RandTCP, on the two middle columns). We can observe that although the RandTCP traffic has more variability than the regular traffic, it is still far from looking like the web traffic. The adversarial stuffing approach takes advantage of this differences to evade the detector, which ends up deciding that most adversarial stuffing samples look more like web traffic than like the two middle columns, which is what it understands to be Metasploit traffic. We also observe that adversarial stuffing and web traffic have similar TLS record sizes (most black, some yellow) yet adversarial stuffing has more TLS records per TCP flow than web traffic – which, if used to retrain another detector, could be used to distinguish Metasploit from web traffic.

#### D. Overhead

In this section we look at the impact that our Metasploit modifications have on the number of bytes sent between framework and payload and on the time it takes to have the payload run a set of pentesting commands.

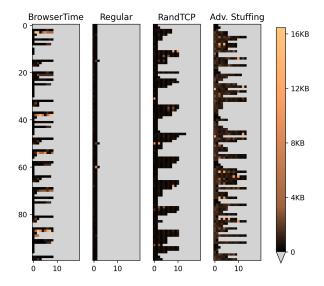


Fig. 9: TLS traffic sample visualization: each line of pixels corresponds to a sequence of TLS AppData records for a different TCP connection; pixel colors represent the size of TLS AppData records. Left: BrowserTime web traffic, middle-left: original Metasploit C2 traffic, middle-right: Metasploit traffic with random number of HTTP requests per TCP connection; right: adversarial stuffing traffic.

We observe no significant changes in the runtime of a sequence of 12 Metasploit commands<sup>6</sup> and a 3 fold increase in the number of TLS AppData record bytes. We did 20 runs of the 12 commands sequence and discarded 1 outlier, for both

adversarial stuffing modification and without modifications. The resulting average runtime for the 12 command sequence is 16.43 s (with less than 0.01 s difference between different runs) for both regular and adversarial stuffing. In addition to the 3 fold increase in byte count, Figure 10 shows much more variation in the byte count values with the adversarial stuffing modification. This is consistent with adversarial stuffing introducing not only more data but also more random data in the communication. Figure 11 illustrates this with an example of comparing TLS AppData record sizes for this sequence with and without our modifications.

We also found that, unlike the number of TLS AppData record bytes, the total number of bytes sent between the framework and the payload is 25% smaller for adversarial stuffing. Figure 12 shows the distribution for total number of bytes. Although this sounds counter intuitive as we know we have more TLS AppData record bytes, it can be explained by the overhead of the TLS handshake for each TLS/TCP connection. In fact, the *Multiple HTTP Request per TCP connection* modification reduces the number of TCP connections for the same number of Metasploit commands and, as such, reduces the TLS handshake overhead.

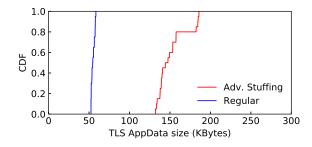


Fig. 10: CDF of the total TLS AppData record byte count per sequence of 12 Metasploit commands.

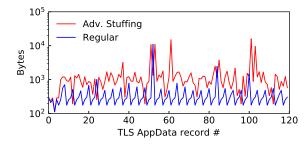


Fig. 11: Sequence of TLS record sizes for Regular Metasploit vs. Metasploit with Adversarial Stuffing

<sup>6</sup>sysinfo, ps, getuid, getpid, ipconfig, route,
pwd, ls, cat /etc/shadow, download /etc/shadow,
webcam\_list, exit

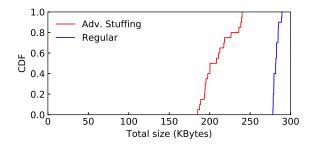
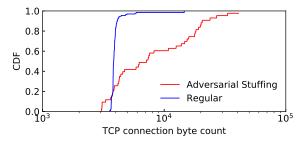


Fig. 12: CDF of the total byte counts per sequence of 12 Metasploit commands.

Figure 13 (top) shows that the minimum byte count value for a TCP connection is between 3 and 4 KBytes, which mostly accounts for the handshake of the TLS connection including certificates. Most regular TCP connections (without our modifications) have close to the minimum value, which indicates that most of these connections are likely polling requests without any Metasploit commands. Figure 13 (bottom) also shows that regular TCP connections are mostly short lived, which is consistent with the polling pattern. The byte count CDF (Figure 13, top) shows adversarial stuffing TCP connections have many different byte counts, as expected given they are trying to change the classifier decision by adding more random-sized stuffing and making the TCP connections last longer. This is confirmed by the TCP connection duration CDF. Although most adversarial stuffing TCP connections have more bytes than regular connections, the total number of adversarial stuffing connections is approximately 4 times smaller than in the regular case, which compensates the increase of byte count per connection.

We can also observe from Figure 14 that the time between the end of a TCP flow and the beginning of the next flow is extremely similar when comparing the regular and adversarial stuffing schemes. In the regular communication scheme this represents the off period between two polling requests, and is likely related to the backoff that the payload introduces in the polling period when there are no Metasploit commands from the framework. We do not modify the original polling backoff mechanisms, although it would make sense if we were to evade a detector that uses the time between consecutive TCP flows as a feature.



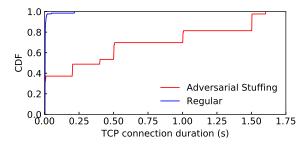


Fig. 13: CDF of the byte counts and duration per TCP connection for an example sequence of 12 Metasploit commands.

## VII. CONCLUSION

In this paper we validated what we suspect is the widely perceived notion that Metasploit C2 traffic can be easily detected even if it is wrapped under TLS encryption. To do so we trained a machine learning detector to distinguish between Metasploit TCP connections and web browsing TCP connections and validated this detector against Metasploit C2 traffic and web browsing traffic that we captured. This yielded 99% accuracy on a balanced dataset. We then modified the Metasploit and Mettle source code so that its C2 traffic can evade the detector and showed that with two different modifications to the traffic -1) adding bytes, 2) grouping two or more Metasploit HTTP requests into a single TCP connection - a pentester can evade the detector in almost 100% of the Metasploit's TCP connections. We then retrained the detector with samples from the modified Metasploit C2 traffic, which again yielded a high (98.4%) accuracy, and devised an approach to evade the new detector. This approach uses adversarial learning to choose how many bytes and TLS records a Metasploit C2 TLS connection should have in order to evade the improved detector. We conclude that the evasion rate drops slightly to 91% when compared to the evasion rate of the initial detector and that if the modifications to the traffic are done only on the payload side then the evasion rate drops significantly to 50%. We then look at the overhead of implementing the adversarial learning approach and find that although the size of the TLS payloads increases three fold due to our adding of bytes, the overall number of bytes sent between the Metasploit framework and the Mettle payload reduces by an average of 25% due to a smaller overhead in the total number of TLS handshakes. We also observe that the runtime of Metasploit commands in the payload with our

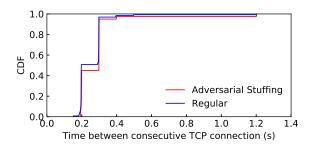


Fig. 14: CDF of the time between consecutive TCP connections, for an example sequence of 12 Metasploit commands.

modifications does not increase when compared to the original Metasploit.

In the future we intend to improve our Metasploit and Mettle modifications to split TLS records in two or more, thus considering the case where adversarial learning suggests a smaller TLS record size value than the one that needs to be sent. We also intend to choose adversarial samples according to the command that the Metasploit framework is about to issue to the payload rather than randomly choosing from a set of Metasploit samples. Regarding the adversarial learning approach, we expect to further study the cycle of retraining and adversarial learning; in this paper we did a single iteration of adversarial learning, but the detector could easily apply adversarial learning to its own model and retrain the model with the adversarial samples, which would lead to a cycle. We would like to understand the properties of such a cycle and whether after some point the detector or the pentester could prevail over the other. Finally, we expect to be able to study an alternative to adversarial learning that is closer to mimicry and attempts to squeeze Metasploit traffic into sequences of TLS record sizes obtained from non-Metasploit traffic, namely web.

We would not like to conclude this paper without providing some considerations about how the Metasploit modifications we study here could be used improperly – and by improperly we mean outside of the context of whitehat pentesting such as those targeting third-party systems and which can be detected by honeypot studies [3]. These modifications are relatively straightforward to implement given modest know-how in programming, networking, and the Metasploit framework, and we believe any modest threat actor could develop them. This is one reason for publishing this work – so network defenders can better understand what happens when a modified Metasploit payload is deployed in their networks. Less mature cybercriminals - whose actions likely generate the bulk of attacks these days – and that have only user-level experience of Metasploit, may find it harder to develop these modifications. These users would likely resort to downloading the code from a repository. Because of that, we decided not to publish our Metasploit and Mettle source code.

#### REFERENCES

- [1] K. F. Steinmetz, "Craft(y)ness: An Ethnographic Study of Hacking," The British Journal of Criminology, vol. 55, no. 1, pp. 125–145, 09 2014. [Online]. Available: https://doi.org/10.1093/bjc/azu061
- [2] F. Holik, J. Horalek, O. Marik, S. Neradova, and S. Zitta, "Effective penetration testing with Metasploit framework and methodologies," in 2014 IEEE 15th International Symposium on Computational Intelligence and Informatics (CINTI), 2014, pp. 237–242.
- [3] E. Ramirez-Silva and M. Dacier, "Empirical study of the impact of Metasploit-related attacks in 4 years of attack traces," in Advances in Computer Science – ASIAN 2007. Computer and Network Security, I. Cervesato, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 198–211.
- [4] R. Mateless, H. Zlatokrilov, L. Orevi, M. Segal, and R. Moskovitch, "IPvest: Clustering the ip traffic of network entities hidden behind a single ip address using machine learning," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.
- [5] N. Papernot, P. McDaniel, A. Sinha, and M. P. Wellman, "SoK: Security and privacy in machine learning," in 2018 IEEE European Symposium on Security and Privacy (EuroS P), 2018, pp. 399–414.
- [6] J. Piet, B. Anderson, and D. McGrew, "An in-depth study of open-source command and control frameworks," in 2018 13th International Conference on Malicious and Unwanted Software (MALWARE), 2018, pp. 1–8
- [7] K. Wadner, "An analysis of Meterpreter during post-exploitation," 2014.[Online]. Available: https://www.sans.org/reading-room/whitepapers/forensics/paper/35537
- [8] R. Wang, P. Ning, T. Xie, and Q. Chen, "MetaSymploit: Day-one defense against script-based attacks with security-enhanced symbolic analysis," in 22nd USENIX Security Symposium (USENIX Security 13), 2013, pp. 65–80.
- [9] W. Chen, "Encapsulating antivirus (AV) evasion techniques in Metasploit framework," 9 2018, White Paper. [Online]. Available: https://www.rapid7.com/globalassets/\_pdfs/whitepaperguide/ rapid7-whitepaper-metasploit-framework-encapsulating-av-techniques. pdf
- [10] P. Casey, M. Topor, E. Hennessy, S. Alrabaee, M. Aloqaily, and A. Boukerche, "Applied comparative evaluation of the metasploit evasion module," in 2019 IEEE Symposium on Computers and Communications (ISCC), 2019, pp. 1–6.
- [11] D. Shu, N. O. Leslie, C. A. Kamhoua, and C. S. Tucker, "Generative adversarial attacks against intrusion detection systems using active learning," in *Proceedings of the 2nd ACM Workshop on Wireless Security* and Machine Learning, ser. WiseML '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1–6.
- [12] K. Yang, J. Liu, C. Zhang, and Y. Fang, "Adversarial examples against the deep learning based network intrusion detection systems," in MILCOM 2018 - 2018 IEEE Military Communications Conference (MILCOM), 2018, pp. 559–564.
- [13] C. Zhang, X. Costa-Perez, and P. Patras, "Tiki-Taka: Attacking and defending deep learning-based intrusion detection systems," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 27–39.
- [14] M. Rigaki and S. Garcia, "Bringing a GAN to a knife-fight: Adapting malware communication to avoid detection," 2018 IEEE Security and Privacy Workshops (SPW), pp. 70–75, 2018.
- [15] J. Li, L. Zhou, H. Li, L. Yan, and H. Zhu, "Dynamic traffic feature camouflaging via generative adversarial networks," in 2019 IEEE Conference on Communications and Network Security (CNS), 2019, pp. 268–276.
- [16] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," arXiv preprint arXiv:1412.6572, 2015.
- [17] N. Papernot, F. Faghri, N. Carlini, I. Goodfellow, R. Feinman, A. Kurakin, C. Xie, Y. Sharma, T. Brown, A. Roy, A. Matyasko, V. Behzadan, K. Hambardzumyan, Z. Zhang, Y.-L. Juang, Z. Li, R. Sheatsley, A. Garg, J. Uesato, W. Gierke, Y. Dong, D. Berthelot, P. Hendricks, J. Rauber, and R. Long, "Technical report on the cleverhans v2.1.0 adversarial examples library," arXiv preprint arXiv:1610.00768, 2018.