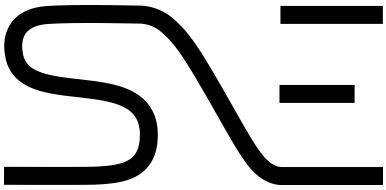
University of Amsterdam



MSc System and Network Engineering

Repurposing defensive technologies for offensive Red Team operations

Kristiyan Mladenov | Arne Zismer
{kmladenov,azismer}@os3.nl
February 12, 2017

Abstract

Performing a successful penetration test highly depends on being stealthy and remaining undetected. While being on assignment, the Red Team performing the test has to avoid triggering traps and getting caught by the Blue Team. In this paper we take a close look at a popular class of systems used for detecting or preventing intrusions - the so called IDS or IPS. A promising idea is to place such a device under the control of the Red Team so that it can prevent them from performing tests noticeable by their opponents. In the paper we show how this system can also be utilised to further help the attacker blend in with the evaluated environment. First we look at the capabilities of the engines to manipulate the traffic through their normalisers. Then, for the scenarios where they come short, we also propose tools from the toolbox of the defensive team and show a way to integrate them with the IPS device. Finally we evaluate their effectiveness against the de-facto standard for network scanner - `Nmap`.

CONTENTS

1 INTRODUCTION 2

2 THEORETICAL BACKGROUND 3
- 2.1 IDS/IPS solutions . . . 3
- 2.2 Kernel network traffic processing . . . 4
- 2.3 OS detection . . . 4
  - 2.3.1 TCP/IP fingerprinting . . . 5
  - 2.3.2 Passive OS fingerprinting . . . 6
  - 2.3.3 Service and version detection . . . 6
- 2.4 Honeypots . . . 6
- 2.5 Port knocking . . . 7

3 RELATED WORK 8
- 3.1 IDS/IPS engine . . . 8
- 3.2 Kernel network traffic processing . . . 8
- 3.3 Other traffic processing tools . . . 8
- 3.4 Virtual honeypots . . . 8

4 METHODOLOGY 10
- 4.1 General environment setup . . . 10
- 4.2 Testing IPS/IDS engines . . . 11
  - 4.2.1 Snort 2.9.9.0 . . . 12
  - 4.2.2 Snort 3 alpha . . . 13
  - 4.2.3 Suricata 3.2 . . . 13
- 4.3 Defeating OS detection . . . 14
  - 4.3.1 Evading TCP/IP fingerprinting . . . 14
  - 4.3.2 Hiding open ports . . . 14
- 4.4 Integration and deployment . . . 16

5 RESULTS 18
- 5.1 IDS/IPS engines tests . . . 18
- 5.2 Defeating OS detection . . . 18
  - 5.2.1 TCP/IP fingerprinting . . . 18
  - 5.2.2 Hiding open ports . . . 19

6 DISCUSSION 21

7 CONCLUSION 22

1 INTRODUCTION

Cybersecurity professionals have borrowed a term originally used by the military during training. The teams get divided into a Blue Team having defensive functions and a Red Team having offensive tasks. So in the terms of the cybersecurity, the Red Team is performing a penetration test while the Blue Team is monitoring the systems trying to detect them. A typical engagement of the Red Team has several stages and usually begins with reconnaissance and gradually flows into exploitation of identified vulnerabilities. However, at each stage of the engagement, the penetration testers might get detected, usually by unintentionally rising an alert into the Intrusion Detection System or Intrusion Prevention System (IDS or IPS) under the Blue Team's control. This happens mainly when they make an easily identifiable mistake. So what would be the outcome if the Red Team has the same powerful solutions as the ones used by the other side to catch them? That raised our main question:

> **To which extend could the covertness of Red Team engagements be improved by utilising an IDS/IPS device under their control?**

Some of the most popular current open source IDS/IPS engines were investigated. The main area in which a focus was placed is:

(1) **How applicable would a current IDS/IPS system be in filtering out suspicious behaviour of the Red Team during their operations?**

Answering this question aims at preventing the Red Team from executing suspicious attacks inadvertently. However, soon we found out that the engine itself would not be able to handle all the situations a penetration tester might find himself into. It appeared to be very straightforward to prevent him from executing well-known attacks, so we moved our attention into preventing him from being passively or actively fingerprinted. Our second question got the following formulation:

(2) **Could an IDS solution be customised in a way that it raises alerts when the offencive team is getting scanned and disguise their identity.**

As we checked the possibilities provided by the IDS/IPS engines we also looked at some other ways in which tools typically used by the defensive teams might be utilised to prevent the attacker's presence from being detected.

The remainder of the this paper is structured as follows: In chapter 2 on the following page we provide a brief overview of the theoretical background that we based our research on, including IPS solutions, kernel routing, OS-detection, honeypots and port knocking. Chapter 3 on page 8 summarises relevant research that has already been conducted in this field. The setup of our isolated test environment and the design of the conducted experiments are presented in chapter 4 on page 10. In chapter 5 on page 18 we present our results which we discuss in chapter 6 on page 21. We conclude our research and present ideas for future research in chapter 7 on page 22.

2 THEORETICAL BACKGROUND

In this chapter we present important concepts that we based our research on. We also use it to make the reader aware of some of the technologies we have investigated and their working mechanisms.

2.1 IDS/IPS SOLUTIONS

In the title of this subsection we introduce the terms IPS and IDS interchangeably. It means either an Intrusion Prevention System or Intrusion Detection System. The reason we take this liberty is because a certain system, no matter if it is IDS or IPS, uses the same engine for finding malicious activities in the network traffic. The difference between the two comes when they actually identify a threat. The IDS can only raise an alert that something is happening, while the IPS can actively make a decision to drop the traffic that seems bad-natured in addition to raising the alert.

There are different ways of identifying a threat. The most popular ones are by using signatures or by making a comparison between the current metrics of the traffic[1] and a predefined baseline of the same metrics. Those two methods are also known as signature-based detection or anomaly-based detection. The reader should be aware that signature-based detection can also be used for certain anomalies in the traffic, but those anomalies are rather violations of the protocols and the standards than variations from a baseline as the one specified above. So when one talks about anomaly-based intrusion detection he usually means the deviation from the norms rather than protocol misbehaviour. Two good examples for signature-based detection are the open-source tools `Snort` and `Suricata`. An example for anomaly-based detection is the `Bro` software.

There is also a third type of system capable of finding and alerting for intrusions in the network. It is called Security Information and Event Management (SIEM) system. It does not look at the traffic flowing in the network but rather collects and aggregates events logged at different places in the network - switches, routers, firewalls, servers, etc. and tries to correlate them. It is not an IDS/IPS system in the classical sense, but can still be a powerful tool for identifying threats in the network.

Given all the above an anomaly-based IDS/IPS system would not be applicable for short-term engagement, as it needs to know in advance what is the baseline for the network traffic. Of course such data may be leaked or obtained by other means, but it would require specific fine-tuning for each separate Red Team engagement. The situation with the SIEM solutions is similar, as it is highly unlikely for the attacker to have the real-time logging data from the devices in the network. Moreover, those two systems would only be able to notify the attacker that he might have gave away his presence in the network and not prevent him from revealing himself in the first place. This is why we placed our focus on researching signature-based IDS/IPS systems. They are capable of standing inline for the traffic and actively making a decision whether to forward a certain packet or not, thus having a better chance to improve the covertness of the attacker.

[1] like for example bandwidth, latency, response time, packets per second, communicating computers, etc.

2.2 KERNEL NETWORK TRAFFIC PROCESSING

The Linux kernel is capable of handling network traffic not only as an end-host in the network, but also as an intermediary device. Depending on the layer of the OSI model at which the decision for whether and where to forward the traffic is made we can distinguish a few separate terms:

- with `bridging` the decision is based on `Ethernet` MAC addresses found at OSI Layer 2.
- with `routing` the forwarding decisions are based on fields like destination IP address found at OSI Layer 3.

A Linux machine can also act as a firewall and take forwarding decisions based on the aforementioned fields in the data and the TCP/UDP port numbers available at the Transport layer (Layer 4). `netfilter` is the packet filtering framework inside the Linux kernel which also provides multiple hooks on which external processes might attach and listen to traffic. A general overview of the hooks is shown in Figure 1[13]. We have made further use of a program called `ebtables` which provides a means for defining the filtering rules in a bridged environment, as the one our transparent solution needs.

It is worth mentioning that the Linux kernel developers have come up with multiple ways for processing traffic in both incoming and outgoing direction. Some of them are more suitable for multithreaded environments, while others are not. There are also multiple ways in which traffic can be acquired. Two of them were analysed further: `AF_PACKET` and `NFQUEUE`. The first one supports raw packet acquisition while the other one supports some preprocessing and filtering by the `netfilter` framework and then passing the filtered data in a queue [15].

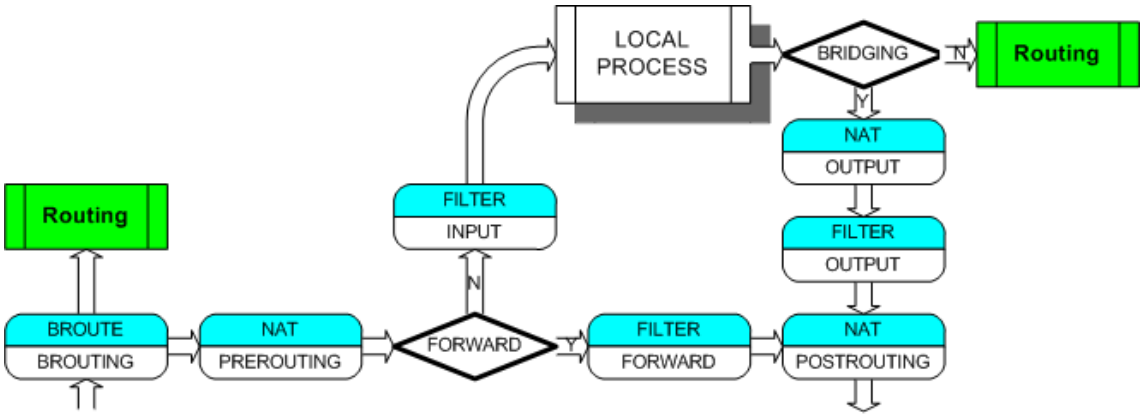


Figure 1: Traversal process and kernel hooks of `ebtables` [2]

2.3 OS DETECTION

This section discusses the background of operating system (OS) detection which is predominantly used for determining vulnerabilities of target hosts, tailoring custom exploits and social engineering. Another area of application which is particularly relevant for our research is the detection of unauthorised and dangerous devices on a network. [10] When connecting to the targeted network, chances are high that an attentive system administrator detects the new device and starts investigating it. In a corporate network predominantly populated with Windows desktops a Linux machine will definitely raise suspicion. Therefore, it is important to understand

[2] www.ebtables.netfilter.org/br_fw_ia/br_fw_ia.html#section2

how OS detection works and how it might be defeated. The most common and capable tool for this purpose is the network exploration tool Nmap. We will therefore focus on the OS detection mechanisms of Nmap.

2.3.1 TCP/IP fingerprinting

The primary idea behind Nmap's OS detection is to investigate the details of the TCP/IP stack implementation of a machine. This investigation mainly focuses on features that are not specified in the respective RFCs. It sends up to 16 TCP, UDP and ICMP probes to open and closed ports of the examined machine. The OS-specific implementations of the TCP/IP stack are different enough to use the behaviour, observed through the responses, as fingerprints.[10]

One implementation specific property is the way sequences are generated. So the first property examined by Nmap is sequence generation. The tool sends six TCP SYN packets with different TCP options and TCP window field values. The sequence and acknowledgement numbers are randomly generated and and saved, so Nmap can differentiate responses. The probing packets are sent exactly 100 milliseconds apart because some sequence algorithms are time dependent. The remaining tests include ping replies, the support for TCP explicit congestion notifications, the greatest common divisor of the initial sequence number (ISN), the ISN counter rate and the TCP initial window size. Furthermore, since RFC 793[3] did not specify the ordering of the TCP header options, these orderings provide further information about a particular TCP/IP implementation. Nmap collects the results of all tests in its own fingerprint format. As seen in listing 2.1 this format represents the result of the various tests (SEQ, OPS, WIN, ECN, T1 - T7, U1 and IE) and maps them to a human readable OS description. The known fingerprints are stored in the nmap-os-db file which is part of Nmap. Once, an OS scan is performed, the resulting fingerprint is compared to the existing fingerprints in order to find a match. [10]

```
1  Fingerprint Apple Mac OS X 10.4.11 (Tiger) (Darwin 8.11.0, PowerPC)
2  Class Apple | Mac OS X | 10.4.X | general purpose
3  CPE cpe:/o:apple:mac_os_x:10.4.11
4  SEQ(SP=C-1DA%GCD=1-6|>1000000%ISR=D0-122%TI=I%CI=I%II=I%SS=S%TS=1)
5  OPS(O1=M5B4NW5NNT11SLL%O2=M5B4NW5NNT11SLL%O3=M5B4NW5NNT11%O4=M5B4NW5NNT11SLL%O5=M5B4NW5NNT11SLL%O6=
       M5B4NNT11SLL)
6  WIN(W1=FFFF%W2=FFFF%W3=FFFF%W4=FFFF%W5=FFFF%W6=FFFF)
7  ECN(R=Y%DF=Y%T=3B-45%TG=40%W=FFFF%O=M5B4NW5SLL%CC=N%Q=)
8  T1(R=Y%DF=Y%T=3B-45%TG=40%S=O%A=S+%F=AS%RD=0%Q=)
9  T2(R=N)
10 T3(R=N)
11 T4(R=Y%DF=Y%T=3B-45%TG=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
12 T5(R=Y%DF=N%T=3B-45%TG=40%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)
13 T6(R=Y%DF=Y%T=3B-45%TG=40%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)
14 T7(R=Y%DF=N%T=3B-45%TG=40%W=0%S=Z%A=S%F=AR%O=%RD=0%Q=)
15 U1(DF=N%T=3B-45%TG=40%IPL=38%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=0%RUD=G)
16 IE(DFI=S%T=3B-45%TG=40%CD=S)
```

Listing 2.1: Example of an Nmap TCP/IP fingerprint.

Nmap's OS detection mechanism can be defeated by manipulating the way packets are crafted to match that of another OS. Tools such as IP Personality[4] and Fingerprint Fucker[5] are kernel modules which provide this functionality. They manipulate the TCP/IP stack in order to make it behave like a specific TCP/IP implementation. This solution is not very flexible, since it requires the kernel to be recompiled. Furthermore, these modules are not maintained anymore and their last versions are only suitable for Linux kernels 2.4 and 2.2 respectively, which renders them useless for recent Linux versions. [10]

[3] www.rfc-editor.org/rfc/rfc793.txt
[4] ippersonality.sourceforge.net/
[5] nmap.org/misc/defeat-nmap-osdetect.html#FF

2.3.2 Passive OS fingerprinting

Methods exist that make it possible to identify the Operating System of a certain network-connected computer by only studying the traffic that it generates. They do not require the host to respond to any specifically generated packet but still need to look at existing traffic flows between the machine under investigation and some other computer. The fields in the traffic that go under investigation are the ones that are not specified strictly in the standards describing the networking protocol and that each OS vendor is free to interpret and implement in whatever way he finds appropriate. [10]

2.3.3 Service and version detection

Another way to determine a network device's OS is to examine its services. `Nmap` is capable of detecting distinct versions of services which help to identify the OS on which the service is running on. As seen in line 8 and 10 of listing 2.2, a service scan against a Ubuntu machine running an `ssh` service reliably detects the OS.

```
1  $ sudo nmap -sV 10.0.0.220
2
3  Starting Nmap 7.01 ( https://nmap.org ) at 2017-02-03 16:31 CET
4  Nmap scan report for 10.0.0.220
5  Host is up (0.000065s latency).
6  Not shown: 999 closed ports
7  PORT   STATE SERVICE VERSION
8  22/tcp open  ssh     OpenSSH 7.2p2 Ubuntu 4ubuntu2.1 (Ubuntu Linux; protocol 2.0)
9  MAC Address: 00:0C:29:40:E7:6A (VMware)
10 Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
11
12 Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
13 Nmap done: 1 IP address (1 host up) scanned in 1.94 seconds
```

Listing 2.2: `Nmap` service scan against a Ubuntu machine running an `ssh` daemon.

The services are simply detected by reading the `nmap-services` file which statically maps port numbers to services. The more valuable information about the exact version of the service is derived from the `nmap-service-probes` file. When detecting an open TCP or UDP port `Nmap` sends probes from this file to the services and compares the received responses with the expected responses from the file. Furthermore, it also tries to interact with the service in order to obtain additional service details such as the application name, version number, device type and OS family. [10]

2.4 HONEYPOTS

Honeypots are isolated network devices that are intended to attract the attacker's attention by exposing known vulnerabilities which let them appear as easy targets. They are deployed to facilitate the detection of an attack and to learn more about the attacker's approach. There is a large variety of honeypots which can roughly be divided into *high-interaction* honeypots and *low-interaction* honeypots. [11]

High-interaction honeypots are complete OSes with fully functional services that let an attacker interact with the honeypot the same way as with a real OS. Is usually logs the all interactions in order to analyse the attacker's behaviour and is isolated from the network so that it can be fully compromised without posing a thread. [11]

Low-interaction honeypots merely simulate a subset of the OS, typically the networks stack and a few services. In contrast to high-interaction honeypots they gather less information about an attack and are easier to detect but the fact that they are more flexible and light-weight makes them very suitable for our research because it allows us to simulate OS-specific services. [11] The `Honeyd` service, which is discussed in more detail in section 3.4 on the next page is even capable of simulating OS-specific TCP/IP fingerprints.

2.5 PORT KNOCKING

Originally, port knocking is a method that is intended to provide authentication for protected services via communication with closed ports. Even though it is not possible to communicate with closed ports directly, since there are no services to interact with, attempted connections to a closed port are still registered and can be monitored. Amongst others, the registered information includes the IP address of the initiating host, the contacted port and a timestamp. This information allows to detect sequences of attempted connections to closed ports by a certain IP address. A host that wants to connect to a protected service can contact closed ports in a given sequence in order to authenticate itself, similarly to entering a PIN number. Once the contacted host detects that given port sequence, it offers the specified service to the IP address the port sequence came from. Usually, a temporary exception for that IP address is added to the IP filter which opens the port behind which the protected service is running. Thus, by "knocking" closed ports in a certain order, another port opens which reveals the protected service. [6] [8]

In section 4.3.2 on page 14 we present how we repurposed the idea of port knocking to hide open ports that could raise a system administrator's suspicion.

3 RELATED WORK

Since there was no directly related research to be found, this section presents conducted research that is relevant for certain aspects of our project.

3.1 IDS/IPS ENGINE

The Evaluation studies of three intrusion detection systems under various attacks and rule sets [16] paper points at similar performance metrics of Snort, Suricata and Bro.

Another feature of the IDS/IPS engines that we will look into further is their ability to do traffic normalisation. Adversarial attacks against this feature are described in the Evasion, traffic normalization, and end-to-end protocol semantics paper[7].

3.2 KERNEL NETWORK TRAFFIC PROCESSING

In their work Eric Leblond and Giusepe Longo study the different packet acquisition methods available for Suricata [9]. Also the `ebtables` documentation is quite extensive on traffic processing routines[12].

3.3 OTHER TRAFFIC PROCESSING TOOLS

Kees van Reeuwijk and Herbert Bos do some nice work with their `Ruler`[17] framework. However we were not able to find its application in the traffic processing that we need, as it does not deal with header recalculation and manipulation.

3.4 VIRTUAL HONEYPOTS

Provos [14] conducted extensive research about virtual honeypots. He developed the `Honeyd` framework, that allows to simulate entire network topologies of honeypots including network latency and packet loss. Each node in the virtual network is a highly configurable low-interaction honeypot. The framework allows to open individual TCP or UDP ports open, disable ICMP, proxy connections to other hosts on the network and is able to call user-defined scripts that can, for example, simulate a web service like `telnet`. Another outstanding feature of `Honeyd` is its personality engine which can simulate the TCP/IP stack of various operating systems. The TCP/IP fingerprints are stored in the same format as `Nmap`'s fingerprints. Therefore, the `Honeyd` can easily be expanded with fingerprints of operating systems that are not included in its original fingerprint collection.

Since we are merely looking for a way to evade OS detection, we are not interested in simulating an entire network topology. A single virtual honeypot is sufficient. A configuration file that defines a single honeypot with a `Windows XP` personality is shown in listing 3.1 on the next page. In line `6`, all connections to port `22` are reflected back to the source IP address, line `7` forwards connections to RDP port `3389` on another machine in the network and line `8` calls a `BASH` script

that simulates a low-interaction `Telnet` service behind port `23`, mimicking always failing login attempts.

```
create winxp
set winxp personality "Microsoft Windows XP Professional SP1"
set winxp default tcp action reset
set winxp default udp action reset
set winxp default icmp action open
add winxp tcp port 22 proxy $ipsrc:22                          # ssh
add winxp tcp port 3389 proxy 10.0.0.60:3389                   # rdp
add winxp tcp port 23 "/etc/honeypot/scripts/fake_telnet.sh"   # telnet

bind 10.0.0.200 winxp
```

Listing 3.1: Example of a `Honeyd` configuration file that defines a single honeypot.

4 METHODOLOGY

4.1 GENERAL ENVIRONMENT SETUP

The test environment we deployed during our research was entirely virtualised. As a hypervisor we used VMware ESXi 5.5.0 build-3116895 deployed on Dell PowerEdge R230 server with Intel Xeon CPU E3-1240L v5 and 16GiB of DDR4 RAM. We choose the supervisor due to its good support of the Microsoft Windows operating system, as we intended to use it as a Guest OS to better simulate a typical Windows-based corporate environment. The hypervisor was ran with unlimited feature set for the duration of the 60-day trial license, which was enough to conduct the experiments we had planned. We divided the 8 vCPUs and the amount of RAM in a proportional manner, so each virtual machine received 1 vCPU and 2 GiB of RAM.

The test environment included a Windows 7 machine as an example of old but still supported OS[1] and a Windows 10 machine representing a new and current OS. We also used the current Long Term Support release of Ubuntu Server 16.04.1 LTS (GNU/Linux 4.4.0-59-generic x86_64). It was chosen due to our good familiarity with it and also the good community support available. However, we managed to use no functionality specific to the OS. The issues we faced and described further were related to the kernel itself, which is shared by all the other flavours of open-source operating systems. We have also armed the simulated attacker with Kali Linux version 2016.2.

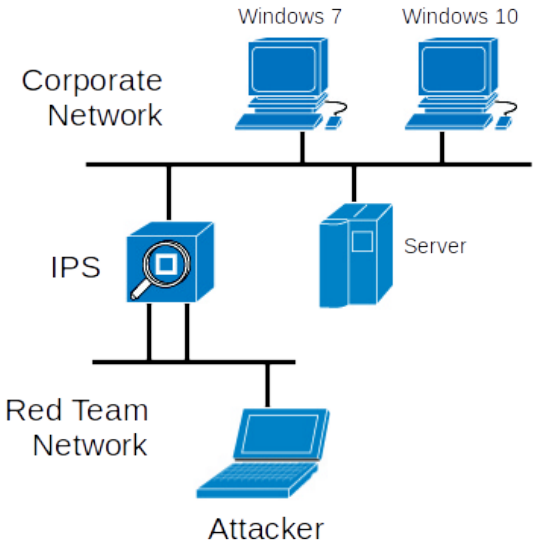


Figure 2: Virtual environment overview

Both networks marked on figure 2 as Corporate Network and Red Team Network are created with the standard virtual switch available within the ESXi hypervisor. By doing this we are trying to mimic a typical Red Team engagement where they simulate what a disgruntled employee or a malicious guest might achieve. The Red Team Network is usually not separated from the Corporate Network under investigation, but in order to introduce the IPS inline we had to make them disjointed. Moreover, the device should be introduced as a bump-in-the-wire, making its two interfaces part of the same network. So we had to arrange an important trade-off which would make a difference compared to a corporate environment and this is the mode in which the virtual switches are running. Their default configuration does not permit MAC address changes and forged transmits[2]. So the only way we could make traffic run through the IPS engine and pass it between the two networks was when we enabled promiscuous mode and allowed forged transmits in the network. It was required because when traffic is bridged through the IPS engine it does not rewrite the MAC address placed in the Ethernet frame by the attacker's computer. More on bridging will follow in Section 4.4 on page 16. The third interface of the IPS

[1] Under extended support until January 14, 2020
[2] transmitting data with a MAC address different than the address assigned to the machine by the hypervisor

device in the figure is used for management purposes. It is also the only one with assigned IP address. We were not able to verify a working solution where the management is done through the other interfaces.

4.2 TESTING IPS/IDS ENGINES

One of the main goals we have set for the research was to find out how an IPS engine can help the attackers to remain more stealthy and raise less suspicion in the surrounding environment. We took the ability of the Intrusion Prevention System to actually prevent the attacker from executing a well-known attack with abundance of signatures that can identify it and took a decision to not question it and take for granted. The more interesting aspect was how to use the IPS engine to further minimise the possibility of active and passive fingerprinting of the attackers OS and the probable services running on it. It was also not obvious how to do all the above without making an additional OSI Layer 3 hop and keep the device operating transparently at OSI Layer 2.

OS	TTL	TCP window (B)
Windows 7	128	8192
Windows 10	128	8192
Kali Linux	64	29200

Figure 3: Default TTL and initial TCP window sizes

In terms of passive OS fingerprinting we checked whether the engines were capable of rewriting the Time-To-Live (TTL) value in the IP header and also the advertised initial window sizes found in the TCP header during the three-way handshake. Some common values can be found in an article by Netresec covering the topic of passive OS fingerprinting [5]. Table 3 points out the values relevant to the systems in the test environment.

The goal in the further tests would be to find a way to adapt the values of the Kali machine to the ones common for the Windows operating systems. Upon successful completion the modification would also have consequences that should be pointed out. The TTL manipulation will interfere with the ability of the attacker to use tools such as `traceroute`. On the other hand, if successfully performed, the initial window size value downscale from 29200 Bytes to 8192 Bytes would not interfere with the ability of the attacker to establish TCP sessions. Nevertheless, it might affect the data transfer speed, which is a problem common for every OS and is also known as TCP slow start. It might also reduce the ability of the attacker to make use of vulnerabilities in the TCP/IP stack of the victim through the initial packet of the TCP three-way handshake. However, those attacks are easily detectable, so they either should not be used, or should be filtered by the Red Team IPS before they reach the victim's network.

In order for the traffic to pass between the interfaces of the machine hosting the IPS engine, first the network adapters needed to be set in promiscuous mode. This mode allows the NIC to receive data that was not originally destined to the machine. The relevant settings are shown in Listing 4.1. Two additional settings are given in lines 3 and 4 in the same listing. The abbreviations `gro` and `lro` stand for Generic Receive Offload and Large Receive Offload. They allow the kernel to receive and prepare for transmit packets with greater size than the Maximum Transmission Unit (MTU) while the NIC handles the segmentation and reassembly. This technique should be disabled to let the IPS engine gather maximum visibility and control over the packets flowing through it. During the tests of the three IPS solutions, all the packets were fed to the respective engine using the `AF_PACKET` acquisition method. It was chosen due to its ability to get the packets directly from the interface's buffers.

```
up ifconfig $IFACE 0.0.0.0 up
up ip link set $IFACE promisc on
post-up ethtool -K $IFACE gro off
post-up ethtool -K $IFACE lro off
```

Listing 4.1: Adjustments for the promiscious interfaces

We have also considered using an engine capable of performing inline substitutions of certain fields in the packets based on regular expressions, like for example `Ruler`[17]. That idea was found unfeasible, as both the TTL value and TCP window size value are included in the calculation of the checksum in the respective header. This means, that upon replacement of the value, the checksum also needs to be recalculated, which we didn't find possible with a regular expression.

4.2.1 Snort 2.9.9.0

For the initial test we decided to use Snort version 2.9.9.0 GRE (Build 56). As of the time of executing the experiments this is the current stable version of Snort, also available in the repositories for the OS we chose. Its working state can be reproduced by following the guides by Noah Dietrich[1]. In order to enable the normalizers, the configuration option `-enable-normalizer` should be added while building from source.

The normalisers (or preprocessors) we considered to achieve the goals we already highlighted were for IP and TCP. Their original purpose is to deserialise the data captured from the wire in a way similar to the operating systems which might be under attack. The goal is to minimise the chance an attack packet evades the inspection due to differences in the way the reassembly is done between Windows and Linux for example.

We were successful with changing the TTL value of the packets by using the IP normaliser. It was set up with the additional options shown in Listing 4.2. They are an exempt from the Snort configuration. In the example the TTL value is changed to 128 for all packets that have a value lower than 65.

```
config min_ttl: 65
config new_ttl: 128
```

Listing 4.2: IP TTL value rewriting

Unfortunately, we did not have the same success with the modification of the size of the initial TCP window. We tried adjusting it as an option in the `stream5` preprocessor, but it did not support it. We also tested the option to reassemble the data as a Windows target, but that did not work either. A summary of the tested options can be found in Listing 4.3. It represents an exempt from the `snort.conf` configuration file. Some of the options tested were quite aggressive. Limiting the window size to 8192 Bytes is a quite desperate measure which might also fire an alert in an anomaly-based IDS engine. Still we were not able to see the result of the preprocessor's actions in the traffic processed by the engine.

```
preprocessor normalize_tcp: ips trim_win ecn stream
preprocessor stream5_tcp: policy windows, max_window 8192
```

Listing 4.3: TCP initial window size change

The ability of the IPS engine to react to active reconnaissance against the attacker was also investigated. Except for triggering an alert in the event of a portscan or service fingerprinting

activity we also looked for a way to make the engine interactively respond and mask the identity of the unusual for the corporate environment computer of the attacker. However, the possibilities given by `flexresp` - the engine that takes care for this, are limited to various forms of TCP resets and ICMP unreachable messages. We could not find a way to customise them. This is why we decided to look for tools which will be able to help the attacker to remain undetected, or at least make the detection harder. They are explained further in section 4.3 on the following page.

4.2.2 Snort 3 alpha

The second test we did was with the new version of Snort which is still an alpha version - 3.0.0-a4 (Build 223). Again the working state of this machine can be reproduced by following the guide Noah Dietrich has put together[2]. The working configuration of the Snort 2.9.9.0 was copied and parsed with the `snort2lua` conversion tool provided with the newer engine. The tests were rerun with the same level of success.

Upon careful reading of the improvements made between the Snort versions [4] one might notice that main changes were made in favour of performance gain. What grabbed our focuse was the addition of LuaJIT [3] which might have added more possibilities for customisation. However, the only one that we found was the addition of the `find` option which can be added during the signature rule definition. We were not able to make it fire an external script upon matching a rule.

4.2.3 Suricata 3.2

The last engine we put to the test was the Suricata 3.2 engine. It was built with the instructions provided by the official documentation [3] with the additional options to enable LuaJIT support - `-with-libluajit-includes` and `-with-libluajit-libraries`. We wanted to test the Lua scripting capabilities as the documentation did not suggest TTL rewrite features as the ones we tested with Snort. We were also not able to modify the behaviour of the TCP reassembly engine. It could have been helpful in reducing the initial TCP window size.

Although it lacks the normalisers that Snort has, the Suricata engine allows for further traffic inspection by passing a given packet that matches a rule to the LuaJIT engine. Then a Lua script is triggered to check for additional anomalies or signs of compromise. The script should return either a `0`, telling Suricata that a match has **not** been found, or a `1`, indicating a match. A script call can be placed inside a rule configured to drop traffic when its conditions are met, thus preventing some forms of attack. An example is given in Listing 4.4

```
drop tcp 10.0.0.200 any -> any any (msg:"Inspected TCP SYN by LUA"; flags:S; sid 1000002; rev:001;
    luajit:tcpinspect.lua;)
```

Listing 4.4: Call Lua script inside Suricata rule

We were able to verify that the Lua script can even call system executables from the underlying OS. It does it with the privileges of the user that started Suricata at the first place. Such type of behaviour makes it possible to execute commands modifying the traffic processing capabilities of the host in a way that can be modelled for different purposes. An example might be a `ebtables` rule as the ones mentioned in Section 4.4 on page 16. It can also be used for dropping the original packet while copying its specific values into a new packet. The later can be designed to fit the OS specifics described in Section 4.2 on page 11.

[3] Just-in-Time compiler for the Lua scripting language

4.3 DEFEATING OS DETECTION

As explained in section 2.3 on page 4, there are two main methods to detect an operating system. In this section we present our approaches of evading the active fingerprinting method.

4.3.1 Evading TCP/IP fingerprinting

One way to evade TCP/IP fingerprinting is to modify the TCP/IP implementation of the machine. `IP Personality`[4] is an open-source patch that can be compiled into Linux kernels. It is based on the `netfilter`[5] framework and can be configured to resemble different operating systems. The fact that the module needs to be recompiled into kernel, each time a different OS should be simulated greatly reduces the flexibility of the tool. Furthermore, `IP Personality` is only compatible with kernel versions 2.4 which would require us to either downgrade the kernel 4.4 of the IPS machine or rewrite the tool for kernel version 4.4. Due to performance and security reasons we decided against downgrading the kernel. A rewrite to kernel version 4.4 requires deep insights into kernel programming which we do not exhibit. Therefore, we ruled out the possibility of making use of this tool.

Instead, we made use of the `Honeyd` framework presented in section 3.4 on page 8 which is capable of creating virtual honeypots with OS-specific TCP/IP fingerprints. In order to let a potential `Nmap` scan inspect the honeypot, instead of the Red Team's machine, we bound the honeypot to the Red Team's IP address. Since we want to keep the IPS as transparent as possible, we want to direct as much traffic as possible to the Red Team's machine and merely direct certain ports to the honeypot. A closer look at `Nmap`'s source code [6] revealed that its OS scan is looking for three specific ports: A closed UDP port, a closed TCP port and an open TCP port. `Nmap` will choose the first suitable ports it finds. Therefore, we need to direct at least three ports to the honeypot.

The `Honeyd` daemon needs to listen to an interface of the IPS machine, while still maintaining the ability to get its traffic filtered. The attachment point under number 2 from Figure 4 was chosen. This gave us the ability to configure rules on which traffic should be forwarded to to the daemon and which - to the attacker's machine. More on this will follow in section 4.4. The relevant settings can be found in listing 3 on page 27 in the appendices.

It is worth mentioning that `Honeyd` did not make use of either the MAC address assigned to interface `br192` on which it was listening to, nor the MAC address in its configuration. This was troublesome as it also responded with a wrong address to the ARP requests for the IP it shares with the attacker's machine. The issue is also addressed in the following section.

4.3.2 Hiding open ports

During the course of the work we were also considering a scenario in which the attacker hosts a malware Command and Control (CnC) server. The server should be reachable for the beacons of the malware that have been deployed during or prior to the assignment. At some point the attention of the systems administrator at the evaluated company may actually get focused on the attacker's computer. A logical next step for him is to scan the suspicious system. The least thing that he expects is to find an open port which should not be there. So how could we prevent the port from appearing to be open?

Different techniques exist in the IPS engine itself to drop data based on the content being exchanged inside a TCP or UDP session. However, none of them can actually prevent a session from being established in the first place. And it is the initial session establishment, which is being evaluated by network scanners like `Nmap` to evaluate which ports are open and which are not. So a reaction was needed as early as the first step of the TCP three-way handshake.

[4] ippersonality.sourceforge.net
[5] www.netfilter.org
[6] www.github.com/nmap/nmap/blob/master/osscan2.cc#L1139-L1225

A closer look was needed at how Nmap works and what is the sequence in which the ports are being evaluated. We found out that the general scan, which takes place when no ports are specified at the command line, probes 1000 ports. According to Nmap's documentation, the ports are scanned in random order but well-known ports were scanned first. In order to get a better understanding about which ports are scanned first, we ran 1000 Nmap scans to find out which ports were scanned before the port we want to hide is reached. The script we used to run the experiment and the obtained pcap files can be found in our GitHub repository [7]. Based on the results presented in section 5.2.2 on page 19 we decided to consider every connection to any of the well-known ports as a potention port scan. We configured knockd to react when it detects a port scan and close (rather than open as it is usually used to) the port used by the malware for communication, so that the scanning engine does not detect it. knockd closes a port by adding the according rule to iptables and optionally removing it after a certain timeout.

Unfortunately, knockd can only detect *sequences* of knocked ports and can therefore not react on single ports. A sequence turned out to consist of at least two ports. Since Nmap scans ports in random order, we generated rules for each possible sequence in which the well-known ports could be scanned. An exempt of the resulting configuration file is shown in listing 4.5. However, running knockd with this configuration file revealed another restriction of the tool: knockd only listens to the first listed sequence of rules with the same first port. Thus, when using the configuration shown in listing 4.5, after knocking port 199, knockd would only detect a sequence when port 3306 is the second port. Knocking port 554 will not be detected by the tool. Our final solution to this problem was to split the configuration file and run multiple instances of knockd which all listened to different sequences in parallel.

```
[close80_199_3306]
    sequence      = 199,3306
    seq_timeout   = 15
    tcpflags      = syn,ack
    start_command = iptables -A INPUT -s %IP% -p tcp -m multiport --dports 80 -j REJECT
    cmd_timeout   = 10
    stop_command  = iptables -D INPUT -s %IP% -p tcp -m multiport --dports 80 -j REJECT
[close80_199_554]
    sequence      = 199,554
    seq_timeout   = 15
    tcpflags      = syn,ack
    start_command = iptables -A INPUT -s %IP% -p tcp -m multiport --dports 80 -j REJECT
    cmd_timeout   = 10
    stop_command  = iptables -D INPUT -s %IP% -p tcp -m multiport --dports 80 -j REJECT
...
```

Listing 4.5: Exempt from knockd.conf

It should be noted that there is a difference between a probe being dropped and being rejected and Nmap can notice it. So it is important to configure the rules properly. The default system behaviour is to reject the connections to closed ports with a TCP Reset, instead of not replying to the probe. A port scanner usually displays the difference between the two as it reports a given port respectively as closed or filtered.

It is also worth mentioning that the port knocking daemon supports executing a second command after a given time period from the triggering knocking activity. In the current scenario this translates to being able to automatically open the closed port again and reenable the malware communication.

Since our solution for hiding a port depends on detecting a sequence of two scanned ports, there will inherently be situations in which the port cannot be closed on time. This is especially the case when Nmap starts its port scan with the port that is to be hidden. In such a case Nmap will always find the port to be opened. As a consequence, our solution merely reduces the

[7] github.com/hopfenzapfen/rp1/tree/master/Experiments/ScannedPorts

probability of a port to be found open. In order to investigate the reliability of the solution, we ran another 1000 port scans and checked how often our hidden port was found to be open. We also investigated the reliability of hiding two ports with the same method. The script we used to run the experiment and the obtained `pcap` files can also be found in our GitHub repository [8]

4.4 INTEGRATION AND DEPLOYMENT

The packets flowing between the Red Team Network and the Corporate Network should be able to pass through the device hosting the IPS engine only upon successfully being inspected. A challenge was to make this happen while the attacker's machine was having the same IP address as the one used by the daemon supposed to be spoofing the attacker computer's identity. Solutions like NAT were not applicable, as they would have interfered with the desired transparency. To prevent addressing conflicts `Honeyd` had to handle only those parts of the traffic destined to the attacker that are needed to properly deceive the scanning activities by the defensive personnel. Neither `iptables` nor `nftables` were effective in that filtering, as with all the test configurations either the traffic was received by both the attacker and `Honeyd` or by none of them. That made us look a layer below and find `ebtabes` - a solution that was able to make the distinction between the two parts of the traffic - for `Honeyd` and for the attacker.

In order for an `ebtables` rule to be processed it needs to be applied to an interface, which is part of a bridge. Initially there were no bridge interfaces in the setup, so the workaround was to build bridge interfaces which had only one physical interface attached to them, instead of two or more as it is by definition. The layout is shown in figure 4. This allowed us to have four different attachment points for the different processes running on the system (e.g. IDS/IPS engine, honeypot, etc). It also gave us greater options to adjust the filtering rules. We did not study whether the bridges have a negative impact on the traffic processing performance. However, if they do, we do not expect it to be high, as all the processing is done in the kernel space without the need to switch to user space.

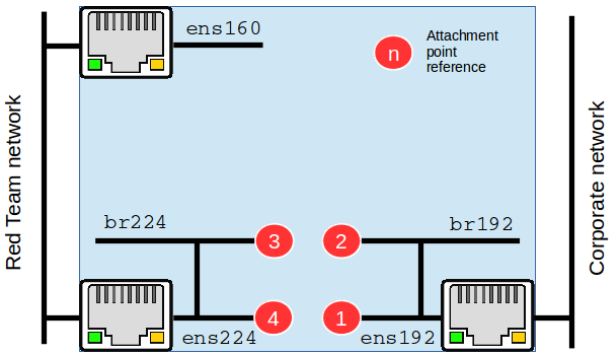


Figure 4: IPS interface layout

In order to enable the bridging in the kernel, some adjustments were needed. First of all the `br_netfileter` module had to be loaded. A way of doing this during system boot is by adding the module's name in the `/etc/modprobe` file. To finally enable the filtering capabilities, the `sysctl` changes listed in listing 4.6 were made.

```
net.bridge.bridge-nf-call-arptables = 1
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
```

Listing 4.6: `sysctl` adjustments for `ebtables`

Given this configuration, the IPS engine was always attached between points 1 and 3 in Figure 4. This provided the ability to filter traffic to and from the attacker's machine on interface

[8] github.com/hopfenzapfen/rp1/tree/master/Experiments/HidePorts

ens224. In the same time the spoofing daemon was listening on interface br192, having the label 2 in Figure 4 on the previous page. This enabled the ebtables filtering rules on interface ens192 to effectively handle the connection filtering for traffic incoming or outgoing to the IPS machine.

We also had to find a solution to the problem of directing only certain ports to the honeypot software, as already mentioned in Section 4.3.1 on page 14. Without manipulation all the traffic entering the IPS machine was able to reach both the attacker and the Honeyd. So in order to allow a certain packet to reach only one of them, the packet should be prevented from reaching the other. An example configuration for blocking a port to the attacker's machine, thus permitting it only to the Honeyd is shown in Listing 4.7. A similar rule should be in place for interface ens192 in order to drop all ports except for those used by the honeypot for fingerprint evasion.

```
1 ebtables -A OUTPUT -o ens224 --ip-dst 10.0.0.200 -p ipv4 --ip-protocol 6 --ip-destination-port 3389 -j
      DROP
```

Listing 4.7: ebtables rules to permit only specific ports to Honeyd

A problem that we faced during the integration of the IPS engine with Honeyd was that when Honeyd handled the traffic, it produced a response with a MAC address different than the one of the attacker's computer. It was also responding to ARP requests with the wrong hardware identifier. This issue was resolved by using the settings shown in Listing 4.8. The command in line 2 was handling the Ethernet source MAC address, while the one in line 4 was handling the MAC address specified in the ARP packets. Such a setting also does not interfere with the ability of the attacker to perform ARP spoofing attacks, as all the ARP packets that he uses will also have the same source MAC as the one specified in the rewriting commands.

```
1 # rewrite source MAC address as well as the one in the ARP replies
2 $ ebtables -t nat -A POSTROUTING -o ens192 -j snat --to-src 00:0C:29:71:FD:4D --snat-arp
```

Listing 4.8: ebtables rules to rewrite the source MAC address of Honeyd

It is also worth mentioning that a transparent configuration like this can also be used to protect the attacker's presence from being detected with periodic ping sweeps. During a ping sweep a large group of hosts are being contacted by something as simple as an ICMP echo request. Even though some of them may have filtered the ICMP protocol in their host firewall, they are still responding to the ARP packets trying to bind their IP addresses with their MAC address. A fine-tuned IPS or the respective rate limitng rules in ebtables shown in Listing 4.9

```
1 ebtables -A INPUT -i ens192 -p ARP --limit 20/second -j ACCEPT
2 ebtables -A INPUT -p ARP -i ens192 -j DROP
```

Listing 4.9: ebtables rules to rate limit ARP

5 RESULTS

In this section we present the results of the experiments that we explained in the previous section.

5.1 IDS/IPS ENGINES TESTS

The tests we ran against the three different open source IDS/IPS engines are summarised in Table 1. Only two of the three tested engines were able to change the TTL value of the packets flowing through them. Moreover, none of them was able to change the value of the TCP window size, negotiated during the three-way handshake.

IPS Engine	TTL handling	TCP window handling
Snort 2.9.9.0	yes	no
Snort 3 alpha	yes	no
Suricata 3.2	no	no

Table 1: Results from the IPS comparison

However, with reasonable additional development Suricata might be able to regenerate the TCP segments that are in question. It should also be able for the engine to handle the TTL rewrite in a similar manner, however it might lead to a lot of additional overhead as the operation should be done for every packet, not only the initial packet of the TCP session establishment.

5.2 DEFEATING OS DETECTION

This section presents in how far we were able to defeat OS detection by evading TCP/IP fingerprinting and hiding ports that we do not want to be found open by an `Nmap` scan.

5.2.1 TCP/IP fingerprinting

We managed to mislead `Nmap`'s OS detection by deflecting a certain portion of ports to `Honeyd` configured to mimic the behaviour of a Windows host. We were able to stir the port scanner from concluding that the attacker is running Linux, which Kali actually is, into giving indeterminate result. Furthermore, when using `Nmap` with the `-osscan-guess` option, which also lists the probable but not conclusive guesses, `Nmap` showed that the system might be running a Windows-based OS. We consider this as a success, as the Windows 10 host in our environment which we used as a reference in the scans also caused the OS fingerprinting to give inconclusive results. An example is given in listing 1 on page 26. A port scan was also ran for comparison against a Windows 10 machine. The inconclusive result is shown in Listing 2 on page 27

However, during the year prior to the experiments `Nmap` also made a change in the way the OS fingerprints were defined. This made them incompatible with `Honeyd`, so an up to date version of the signatures will not be available for the attacker if he wants to mimic another OS relevant to the environment which is being evaluated.

5.2.2 Hiding open ports

In Figure 5 we list the results of the 1000 port scans we ran in order to find determine the well-known ports that `Nmap` scans first. We checked which port is always scanned before ports `80` and `443`. Note that these ports are just examples of ports that can be used as Cnc ports. An arbitrary port can be picked instead of those two, but the randomisation of the sequence in which those ports are checked will always bring similar results. The results show, that there are only 29 ports that `Nmap` considers "well-known" ports. Furthermore, we can see that each port is equally likely to be scanned before the port we want to hide. The third chart of Figure 5 indicates that less ports are scanned before either port `80` or `443` is scanned. This result indicates that it is harder to react on a port scan on time, when trying to hide two ports. Based on these results we decided to listen to all these ports to detect an `Nmap` scan.

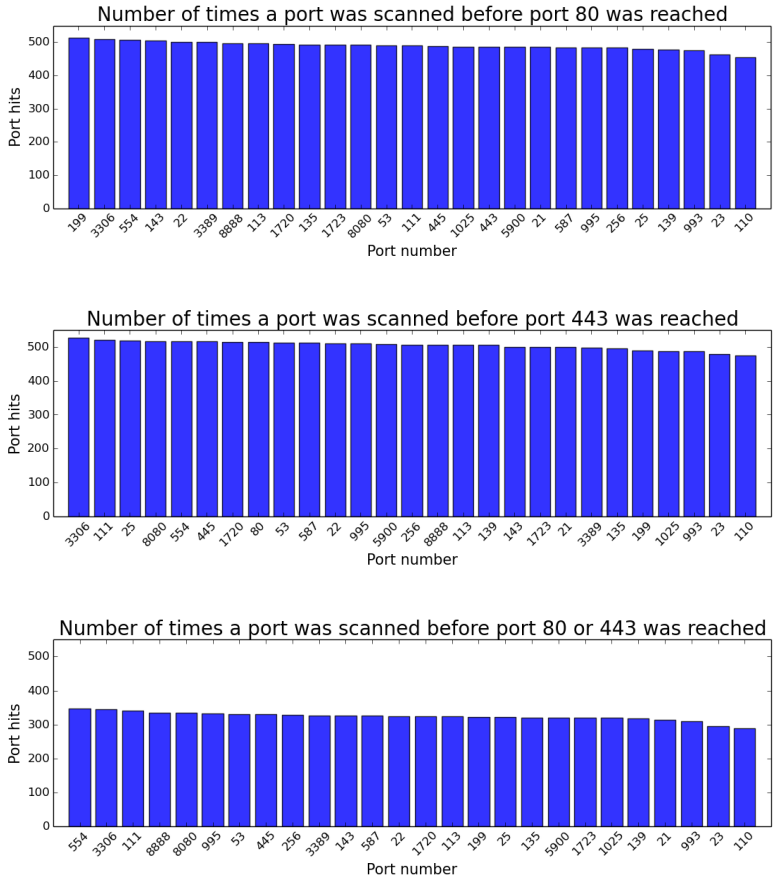


Figure 5: Distribution of ports that were scanned before port 80 and 443 were reached out of 1000 port scans.

With the proposed solution for running multiple `knockd` instances, it is possible to reduce the rate at which the CnC port appears as being open. Figure 6 on the following page shows our success rates when one or two ports need to be hidden. As expected, there are cases in which our solution fails to close the ports on time. This is caused by `Nmap` scans that start scanning with the hidden port and by the reaction time of `Honeyd`.

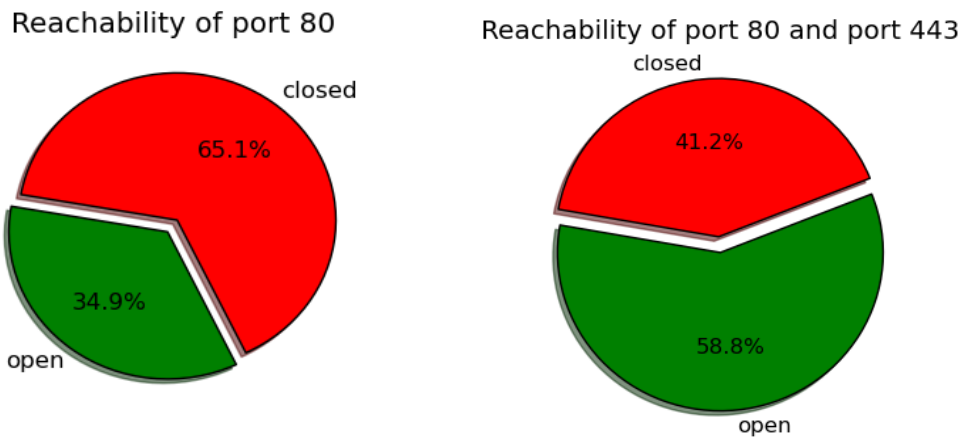


Figure 6: Success rate in hiding CnC ports

6 DISCUSSION

We found that there was an IPS engine capable of changing one of the two investigated values - the TTL of the IP packet. This might reduce the chances of being detected by slow scans utilising ICMP echo requests and reading the TTL values of the responses. However, such a modification should be studied further against anomaly detection IDS engines, as there might be no OS with the combination of the TTL value specific for Windows and the initial TCP window size characteristic for Linux.

An observation that we made while testing the honeypot software was that it added a few milliseconds of delay in the probe responses compared to the attacker's computer on whose behalf it was responding. However, scanning software like `Nmap`, which was used as a de facto standard for network scanners, does not report on delays, but rather on the total time the scan required. Moreover, a custom implementation of a port scanner might be needed in order for the latency anomaly between the responses to be detected. Whether an anomaly detection IDS is capable of pinpointing it is a matter of future work.

A similar issue is present when working with the port knocking daemon. It does not execute the rules for closing the supposed malware CnC port quick enough to actually make the service undetectable in a timely manner. If the port chosen for malware communication is between the first ports scanned by `Nmap`, one might expect the port will be scanned as first or second in less than 10% of the scans. However, based on the results, such expectation does not appear to be feasible. The delay in the reaction might be contributed to the bursty nature of the scan as some of the probe packets might already be in the input buffer of the interface before the rules that should filter them are processed and triggered. However, this requires further investigation before a proper conclusion is reached.

7 CONCLUSION

In this report we have shown that it is possible for the studied IPS engines to be used for improving the covertness of the Red Team. This is shown to be possible not only through their ability to drop suspicious traffic but also through the way they can modify non-malicious flows. This was shown to be feasible either through the traffic normalisation preprocessors or suggested to be done by external utilities triggered by predefined rules.

Where we could not find a solution built in the tested products against the active fingerprinting methods, we looked further into other tools primarily used by the Blue Team. By the use of a honeypot software we successfully prevented a network scanner such as `Nmap` from finding out that the attacker's machine is running a Linux distribution. We also show that it is possible to evade a port scan by utilising a port knocking daemon. Although the success rate was about 2/3 it is still statistically better to run such a service than to stay unprotected.

A solution for filtering traffic in an environment with bridged interfaces is also successfully deployed. It allows further integration of other daemons within the same host as the IPS/IDS engine. Overall the solution is flexible, but would still require some automation to become easy to use.

FUTURE WORK

We suggest the following topics as possible future work.

- Some of the features that we suggest in multiple pieces of software might be tested for interchangeability. For example could the port scanning protection provided by `knockd` be exchanged with a similar or better success rate if triggered by a Suricata rule with LuaJIT script attached to it? Also does any IDS/IPS engine provide customisation of the responses triggered by the rules configured to drop traffic? Is it possible to simulate some of the low-interaction capabilities of the honeypots by those customised responses?
- Although we deem a solution which is improving the covertness by some measurable metrics against popular intrusion detection and reconnaissance software, they might make the behaviour of the attacker more suspicious for the engines focused on detecting anomalies. It is worth investigating further whether a solution like the one we propose is not having an adverse effect on the detection rate of the attacker when anomalies in the network are being observed.
- As already discussed, the `Honeyd` software is not up to date with the latest operating systems and requires an update of the way it handles port scans. It will also be beneficial if templates for newer operating systems like Windows 10 get included by default.
- Currently the solutions we suggest require additional configuration in order to make them fit each separate Red Team engagement. A future work can be done in the direction of automation and making the solution Plug-and-Play.

ACKNOWLEDGEMENT

We would like to thank the Cyber Risk Services department at Deloitte Netherlands and more specifically the people performing the Red Teaming Operations, whose idea for utilising an IPS during assignments we developed into our research. They provided us with valuable input about how a typical engagement is performed. Their ideas served both as a guideline and inspiration to find additional methods of improving the covertness during their operations.

BIBLIOGRAPHY

[1] Noah Dietrich. Snort 2.9. 9. x on ubuntu. http://sublimerobots.com/2017/01/snort-2-9-9-x-ubuntu-installing-snort/, 2017.

[2] Noah Dietrich. Snort 3 on ubuntu. http://sublimerobots.com/2017/01/installing-snort3-in-ubuntu/, 2017.

[3] Suricata Documentation. Deploying suricata on ubuntu. https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Ubuntu_Installation, 2016.

[4] Joel Esler. Introducing snort 3.0. http://blog.snort.org/2014/12/introducing-snort-30.html, 2014.

[5] Erik Hjelmvik. Passive os fingerprinting. *Netresec blog*, 2011.

[6] Sebastien Jeanquier. An analysis of port knocking and single packet authorization. *Information Security Group: Royal Holloway College, University of London*, 2006.

[7] Christian Kreibich, Mark Handley, and V Paxson. Network intrusion detection: Evasion, traffic normalization, and end-to-end protocol semantics. In *Proc. USENIX Security Symposium*, volume 2001, 2001.

[8] Martin Krzywinski. Port knocking from the inside out. *SysAdmin Magazine*, 12(6):12–17, 2003.

[9] Eric Leblond and Giuseppe Longo. Suricata idps and its interaction with linux kernel. In *Proceedings of netdev 1.1, Feb 10-12, 2016, Seville, Spain*, 2016.

[10] Gordon Fyodor Lyon. *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[11] Eric Peter and Todd Schiller. A practical guide to honeypots. *Washington Univerity*, 2011.

[12] Netfilter project. ebtables documentation. http://ebtables.netfilter.org/documentation/docs.html.

[13] Netfilter Project. Netfilter framework. https://www.netfilter.org/.

[14] Niels Provos et al. A virtual honeypot framework. In *USENIX Security Symposium*, volume 173, pages 1–14, 2004.

[15] Dan Siemon. Queueing in the linux network stack. https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/, 2013.

[16] Kittikhun Thongkanchorn, Sudsanguan Ngamsuriyaroj, and Vasaka Visoottiviseth. Evaluation studies of three intrusion detection systems under various attacks and rule sets. In *TENCON 2013-2013 IEEE Region 10 Conference (31194)*, pages 1–4. IEEE, 2013.

[17] Kees van Reeuwijk and Herbert Bos. Ruler: high-speed traffic classification and rewriting using regular expressions. Technical report, Technical report, Vrije Universiteit Amsterdam, 2006.

LIST OF FIGURES

Figure 1 Traversal process and kernel hooks of `ebtables` 4
Figure 2 Virtual environment overview 10
Figure 3 Default TTL and initial TCP window sizes 11
Figure 4 IPS interface layout 16
Figure 5 Distribution of ports that were scanned before port 80 and 443 were reached out of 1000 port scans. 19
Figure 6 Success rate in hiding CnC ports 20

APPENDICES

```
# OS Scan performed against unprotected Kali machine

clone@ubuntu-srv:~$ sudo nmap -O --osscan-guess 10.0.0.200

Starting Nmap 7.01 ( https://nmap.org ) at 2017-02-09 17:01 CET
Nmap scan report for 10.0.0.200
Host is up (0.00027s latency).
All 1000 scanned ports on 10.0.0.200 are closed
MAC Address: 00:0C:29:71:FD:4D (VMware)
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose
Running: Linux 2.4.X|2.6.X
OS CPE: cpe:/o:linux:linux_kernel:2.4.20 cpe:/o:linux:linux_kernel:2.6
OS details: Linux 2.4.20, Linux 2.6.14 - 2.6.34, Linux 2.6.17 (Mandriva), Linux 2.6.23, Linux 2.6.24
Network Distance: 1 hop

OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 3.94 seconds

# OS Scan performed against the Kali machine running behind the IPS engine and Honeyd used for spoofing.

sudo nmap -O --osscan-guess 10.0.0.200

Starting Nmap 7.01 ( https://nmap.org ) at 2017-02-09 17:09 CET
Nmap scan report for 10.0.0.200
Host is up (0.012s latency).
Not shown: 998 closed ports
PORT     STATE SERVICE
445/tcp  open  microsoft-ds
3389/tcp open  ms-wbt-server
MAC Address: 00:0C:29:71:FD:4D (VMware)
Aggressive OS guesses: Microsoft Windows 7 SP0 - SP1 (91%), Microsoft Windows Server 2008 SP2 or Windows
      10 Tech Preview or Xbox One (89%), Microsoft Windows Vista SP0 - SP2, Windows Server 2008, or
     Windows 7 Ultimate (89%), Microsoft Windows Vista, Windows 7 SP1, or Windows 8.1 Update 1 (89%),
     Microsoft Windows Vista (89%), Microsoft Windows 7 or Windows Server 2008 R2 (89%), Windows 7
     Professional SP1 (89%), Microsoft Windows 7 (89%), Microsoft Windows Vista SP1 (89%), Microsoft
     Windows Vista SP2 or Windows 7 Ultimate SP0 - SP1 (89%)
No exact OS matches for host (If you know what OS is running on it, see https://nmap.org/submit/ ).
TCP/IP fingerprint:
OS:SCAN(V=7.01%E=4%D=2/9%OT=445%CT=1%CU=32225%PV=Y%DS=1%DC=D%G=Y%M=000C29%T
OS:M=589C92AD%P=x86_64-pc-linux-gnu)SEQ(SP=FD%GCD=1%ISR=10E%TI=I%CI=I%II=I%
OS:SS=O%TS=7)SEQ(SP=105%GCD=2%ISR=10D%TI=I%TS=7)SEQ(SP=103%GCD=1%ISR=105%TI
OS:=I%CI=RD%TS=7)OPS(O1=M5B4NW0NNT11%O2=M5B4NW0NNT11%O3=M5B4NW0NNT11%O4=M5B
OS:4NW0NNT11%O5=M5B4NW0NNT11%O6=M5B4NW0NNT11)WIN(W1=2000%W2=2000%W3=2000%W4
OS:=2000%W5=2000%W6=2000)ECN(R=Y%DF=Y%T=FFFE%W=2000%O=M5B4NW0NNT10%CC=N%Q=)
OS:T1(R=Y%DF=Y%T=FFFE%S=O%A=S+%F=AS%RD=0%Q=)T2(R=Y%DF=Y%T=FFFE%W=0%S=A%A=S%
OS:F=AR%O=%RD=0%Q=)T3(R=Y%DF=Y%T=FFFE%W=0%S=A%A=Z%F=AR%O=%RD=0%Q=)T4(R=Y%DF
OS:=Y%T=FFFE%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)T5(R=Y%DF=Y%T=3E%W=0%S=Z%A=S+%F=AR%
OS:O=%RD=0%Q=)T6(R=Y%DF=Y%T=3E%W=0%S=A%A=Z%F=R%O=%RD=0%Q=)T7(R=Y%DF=Y%T=3E%
OS:W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)U1(R=Y%DF=N%T=3E%IPL=164%UN=0%RIPL=G%RID=G%
OS:RIPCK=G%RUCK=G%RUD=G)IE(R=Y%DFI=N%T=3E%CD=Z)

Network Distance: 1 hop

OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 68.55 seconds
```

Listing 1: Sample `Nmap` result from scans prior to and after the manipulations

```
sudo nmap -O --osscan-guess 10.0.0.60


Starting Nmap 7.01 ( https://nmap.org ) at 2017-02-09 17:11 CET
Nmap scan report for 10.0.0.60
Host is up (0.00016s latency).
Not shown: 998 filtered ports
PORT     STATE SERVICE
3389/tcp open  ms-wbt-server
5357/tcp open  wsdapi
MAC Address: 00:0C:29:1E:BE:B8 (VMware)
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Device type: general purpose
Running (JUST GUESSING): FreeBSD 6.X (94%)
OS CPE: cpe:/o:freebsd:freebsd:6.2
Aggressive OS guesses: FreeBSD 6.2-RELEASE (94%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 1 hop

OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 24.19 seconds
```

Listing 2: Sample `Nmap` result from scans of a Windows 10 machine

```
cat /etc/honeypot/honeyd.conf

create windows
set windows default icmp action open
set windows default tcp action reset
set windows default udp action reset
set windows personality "Windows-7"
add windows udp port 53 closed
add windows udp port 139 open
add windows tcp port 445 open
add windows tcp port 3389 open

set windows ethernet "00:0c:29:16:3c:80"
bind 10.0.0.200 windows
```

Listing 3: Sample `Honeyd` configuration for achieving the results above

```
#Translation table configuration listed by ebtables -t nat -L

Bridge table: nat

Bridge chain: PREROUTING, entries: 0, policy: ACCEPT

Bridge chain: OUTPUT, entries: 0, policy: ACCEPT

Bridge chain: POSTROUTING, entries: 1, policy: ACCEPT
-o ens192 -j snat --to-src 0:c:29:71:fd:4d --snat-arp --snat-target ACCEPT

#Filtering rules configuration listed by ebtables -L

Bridge table: filter

Bridge chain: INPUT, entries: 4, policy: ACCEPT
-p IPv4 -i ens192 --ip-dst 10.0.0.200 --ip-proto tcp --ip-dport 0:21 -j DROP
-p IPv4 -i ens192 --ip-dst 10.0.0.200 --ip-proto tcp --ip-dport 23:444 -j DROP
-p IPv4 -i ens192 --ip-dst 10.0.0.200 --ip-proto tcp --ip-dport 446:3388 -j DROP
-p IPv4 -i ens192 --ip-dst 10.0.0.200 --ip-proto tcp --ip-dport 3390:65535 -j DROP

Bridge chain: FORWARD, entries: 0, policy: ACCEPT

Bridge chain: OUTPUT, entries: 4, policy: ACCEPT
-p IPv4 -o ens224 --ip-dst 10.0.0.200 --ip-proto tcp --ip-dport 22 -j DROP
-p IPv4 -o ens224 --ip-dst 10.0.0.200 --ip-proto udp --ip-dport 53 -j DROP
-p IPv4 -o ens224 --ip-dst 10.0.0.200 --ip-proto tcp --ip-dport 445 -j DROP
-p IPv4 -o ens224 --ip-dst 10.0.0.200 --ip-proto tcp --ip-dport 3389 -j DROP
snorty@ubuntu-snort:~$
```

Listing 4: Resulting `ebtables` configuration