

POLTECNICO DI TORINO & EURECOM

Master's Degree in Computer Engineering



Master's Degree Thesis

Developing a Proof-of-Concept Malware Detection Engine for Cisco Secure Endpoint

Supervisors

Prof. Davide BALZAROTTI

Prof. Cataldo BASILE

Dr. Jonas ZADDACH

Candidate

Alessandro PISANI

January 2022

Summary

In recent years, exploits for SMB vulnerabilities such as Eternal Blue and Eternal Romance have been released and integrated into malware and attack frameworks. Exploits for NTLM vulnerabilities such as Rotten Potato have been integrated into tools like Juicy Potato, Mimikatz and Metasploit. While Cisco has been asking to their customers to apply vendor patches to protect themselves from these vulnerabilities, it was not providing any visibility into, or detection or prevention from these. Even if an enterprise is patched against these attacks, customers expect Cisco to detect an attempt. Therefore, the main challenge is to research how Cisco Secure Endpoint may detect network based attack against the endpoint or originating from it, while taking into account context, such as what local application is the source or destination of the network traffic. In a second instance, investigate whether the solutions could be used to prevent the attacks in addition to detecting them.

The need to have a way to define some rules which can provide visibility into specific protocol types or detect specific attack patterns can be perfectly handled by Snort IPS/IDS. Currently this solution is not portable across all the Secure Endpoint supported platforms, in particular on Windows. For this reason, the goal is to build snort 3 on Windows to intergate it in the Cisco Secure Endpoint in order to detect network-based attacks such as Eternal Blue, Eternal Romance, Zerologon, DCShadow and DCSync.

Acknowledgements

ACKNOWLEDGMENTS

Table of Contents

List of Tables	VIII
List of Figures	IX
Acronyms	XI
1 Introduction	2
1.1 Outline	3
1.2 Cisco Systems	3
2 Snort 3 and Related Work	5
2.1 Snort: Network Intrusion Detection & Prevention System	5
2.1.1 Snort Layout on the FileSystem	6
2.1.2 Snort Rules	8
2.1.3 Snort Modes of Operation	9
2.1.4 Snort Dependencies	11
2.2 Related Work	12
2.2.1 Suricata	13
2.2.2 Microsoft Defender for Identity	13
2.2.3 ClamAV	14
3 Packet Capture	17
3.1 Windows Network Driver	17
3.1.1 Windows Network Driver Architecture	18
3.1.2 NPF and NDIS	19
3.1.3 Relationship of WPCAP.DLL and PACKET.DLL	20
3.2 LibPCAP	21
3.2.1 MSYS2 and MinGW LibPcap	21
3.3 WinPCAP: Windows Packet Capture	22
3.4 Win10PCAP: WinPcap for Windows 10	25
3.5 Npcap: Windows Packet Capture Library & Driver	25

3.5.1	NPcap License	26
3.6	WinDivert: Windows Packet Divert	26
3.6.1	WinDivert Architecture	27
3.7	Which PCAP Module for Snort 3?	28
4	LibDAQ: The Data Acquisition Library	31
4.1	LibDAQ Introduction	31
4.2	LibDAQ Configuration	32
4.3	DAQ Modules Features	32
4.4	DAQ Modules	33
4.4.1	BPF Module	33
4.4.2	Dump Module	33
4.4.3	Netmap Module	34
4.4.4	PCAP Module	34
4.4.5	Savefile Module	35
4.4.6	Trace Module	35
5	Porting Snort3 to Windows	37
5.1	Running Linux programs on Windows	37
5.1.1	MSYS2	38
5.1.2	MinGW-w64	38
5.1.3	Microsoft Visual C++ (MSVC)	38
5.2	Installing Snort Dependencies for MSVC	39
5.2.1	Npcap Install	39
5.2.2	Libdnet Install	40
5.2.3	Vcpkg Install	40
5.3	Porting LibDAQ to Windows	41
5.4	Re-writing Linux specific code into Windows specific code	42
5.4.1	Porting BSD sockets with Winsock API	42
5.4.2	Porting UNIX Signals in Windows	46
5.4.3	Data structure alignment	48
5.4.4	Thread Local Storage (TLS)	49
5.4.5	SO_PUBLIC and SO_PRIVATE Snort Types	50
5.4.6	Miscellaneous MSVC Fixes	51
6	Automating Snort 3 Installation	55
6.1	Windows Container Base Images	55
6.2	Installing Snort Dependencies in a Windows Container	56
6.3	Installing Visual Studio Build Tools in a Windows Container	57
6.4	Installing Npcap in a Windows Container	57

7 Conclusion and Future Work	59
7.1 Future Work	60
Bibliography	62

List of Tables

3.1	Summary of the PCAP Libraries and their key features.	28
4.1	Summary of the DAQ modules and associated DAQ types.	36
5.1	Snort dependencies and relative installation method on Windows. . .	39
5.2	Signals supported in Windows and their meanings.	47
5.3	Signals used by Snort and their meanings.	47
5.4	MSVC unsupported keywords and their logical operators counterparts.	52

List of Figures

2.1	Snort Layout on the FileSystem.	7
2.2	PulledPork v3.0.0-BETA Display Banner.	9
2.3	Suricata logo	13
2.4	ClamAV logo	15
3.1	Main components of the Windows Network Driver Architecture. . .	18
3.2	daq_pcap_start function hierarchy.	22
3.3	MSYS2 disassembled pcap_create_interface function.	23
3.4	MinGW disassembled pcap_create_interface function.	23
3.5	Win10Pcap device driver "Windows 10 Compatible" logo from Microsoft.	25
3.6	Npcap Silent Installation Error Banner.	27
3.7	The basic architecture of WinDivert.	28
3.8	Snort 3 banner on Windows.	29
5.1	Snort Analyzer's Finite State Machine.	48
6.1	Windows container base images.	56

Acronyms

API

Application Programming Interface

ATP

Advanced Threat Protection

AV

Anti Virus

BPF

Berkley Packet Filter

DAQ

Data Acquisition

DNET

Dumb Network Library

DoS

Denial of Service

FLEX

Fast Lexical analyzer

FST

Flow State Table

GCC

GNU C compiler

GPL

General Public License

HWLOC

Hardware Locality

IDS

Intrusion Detection System

IEEE

Institute of Electrical and Electronics Engineers

IOCTL

Input Output Control

IPC

Inter Process Communication

IPS

Intrusion Prevention System

LINUX

Linus Torvald's UNIX

LUAJIT

LUA Just-In-Time Compiler

LWF

Light-Weight Filter

MD5

Message Digest 5

MINGW

Minimalist GNU for Windows

MSDN

Microsoft Documentation

MSVC

Microsoft Visual C++

NDIS

Network Driver Interface Specification

NFQ

NetFilter Queue

NIC

Network Interface Card

NIDS

Network Intrusion Detection System

NPF

Netgroup Packet Filter

NSM

Network Security Monitoring

OS

Operating System

OSI

Open System Interconnection

PCAP

Packet Capture

PCRE

Perl Compatible Regular Expressions

PDF

Portable Document Format

POSIX

Portable Operating System Interface for Unix

PP

PulledPork

PP3

PulledPork3

SHA

Secure Hash Algorithm

SMB

Server Message Block

SSL

Secure Sockets Layer

SWF

Shockwave Flash

TCP

Transport Control Protocol

TLS

Thread Local Storage

TTY

TeleTYpewriter

VLAN

Virtual Local Area Network

VM

Virtual Machine

WSL

Windows Subsystem for Linux

ZLIB

Z Library

Chapter 1

Introduction

Nowadays threat actors have rapidly increased in sophistication, using techniques that make them harder to detect and that threaten even the savviest targets.

Back then, malware authors had already found ways to evade traditional antivirus solutions, which rely on static analysis, by using techniques such as polymorphism, metamorphism, encryption, obfuscation and anti-reversing protection.

Unfortunately, this increase in sophistication makes reliable malware detection based on Signature-Based Detection (SBD), on Behavior-Based Detection, on Heuristics-Based Detection, or network communication alone inadequate to reliably detect malware completely and secure the target systems.

Cisco Secure Endpoint (formerly AMP - Advanced Malware Protection for Endpoints) integrates prevention, detection, threat hunting and response capabilities in a single solution, leveraging the power of cloud-based analytics to understand which files are effectively malicious.

Cisco Secure Endpoint represents a holistic detection approach to tackle the issues analyzed and to consolidate the detection of events through different collaborating engines.

Achieving a binary distribution of *Snort 3 NIDS for Windows* will lay the foundations for the integration of Snort 3 into the behavioral protection of *Cisco's Secure Endpoint*. This will provide huge benefits to the Security-in-depth principle already adopted by the Secure Endpoint in order to deploy multiple layers of security and effectively detect malware.

1.1 Outline

The following work is divided in these chapters:

- [Chapter 2](#) illustrates the related work and Snort 3 features.
- [Chapter 3](#) performs an in-depth analysis of the available PCAP libraries (and their features) for Windows.
- [Chapter 4](#) analyses the Data AcQuisition library (LibDAQ), which allows Snort to be agnostic of how it gets its data.
- [Chapter 5](#) is the core of Snort 3 porting to Windows, combines the various implementation alternatives and the development achievements.
- [Chapter 6](#) gathers all the final thoughts about this work and the future improvements.

1.2 Cisco Systems

Cisco Systems, Inc. is an American multinational technology corporation headquartered in San Jose, California. Cisco develops, manufactures and sells networking hardware, software, telecommunications equipment and other high-technology services and products.

Cisco Systems was founded in December 1984 by Sandy Lerner along with her husband Leonard Bosack from Stanford University seeking an easier way to connect different types of computer systems.

Cisco Systems shipped its first product in 1986 and is now a multi-national corporation, with over 35,000 employees in more than 115 countries. Today, Cisco solutions are the networking foundations for service providers, small to medium business and enterprise customers which includes corporations, government agencies, utilities and educational institutions.

As part of a rebranding campaign in 2006, Cisco Systems adopted the shortened name "Cisco". Cisco acquired the cyber-security firm Sourcefire, in October 2013. On June 16, 2014, Cisco announced the acquisition of ThreatGRID, a company that provided dynamic malware analysis and threat intelligence technology.

Cisco provides IT products and services across five major technology areas: Networking (including Ethernet, optical, wireless and mobility), Security, Collaboration (including voice, video, and data), Data Center, and the Internet of Things.

Cisco Systems France is a daughter company counting on many company offices throughout France: Mougins, Toulouse, Lyon, Rennes, Lille, Strasbourg and ILM (Issy-les-Moulineaux).

Chapter 2

Snort 3 and Related Work

Snort is the foremost Open Source Intrusion Prevention System (IPS) in the world. Snort IPS uses a series of rules that help define malicious network activity and uses those rules to find packets that match against them and generate alerts for users.

Snort can be deployed inline to stop these packets, as well. Snort has three primary uses: As a packet sniffer like tcpdump, as a packet logger — which is useful for network traffic debugging, or it can be used as a full-blown network intrusion prevention system. [1]

2.1 Snort: Network Intrusion Detection & Prevention System

SNORT® has released its new major version, Snort 3, in January 2021. This version represents a sweeping upgrade to feature improvements and new features which lead to enhanced performance, faster processing, improved scalability for users' network and a range of more than 200 plugins so users can create a custom set-up for their network.

- More Adaptable: Snort has changed the entire code base from C to C++¹ making the code base more modular and easier to maintain on users' network.
- More Efficient: Threading and shared memory allow to scale Snort 3 to user's network and create a much faster start-up. This allows multiple packet

¹Moved from 470,000 lines of C (with an average of 400 lines per file) to 389,000 lines of C++ (with an average of 200 lines per file)

processing to free up more memory for more packet processing power.

- **More Customizable:** Plugins with LuaJIT allows users to write their own plugins much easier than before to do things like add your own Snort Rule options, in-depth file processing, and more.
- **Better Performance:** Snort Rule Syntax has been updated to make it easier to write and to understand. The rule syntax is more concise with fewer rule parts which will allow rules to run quicker.

As of today many versions of Snort 2.9 are reaching their End Of Life (EOL). For example, Snort version 2.9.18.0, which had been initially released on the 15th of June 2021, has reached its EOL on the 30th of November 2021 and upgrading to Snort 3 is strongly recommended.

2.1.1 Snort Layout on the FileSystem

This section details where the Snort 3 configuration files live, where the rules live, where the binaries are installed, and where databases, log files are.

The Snort 3 layout on the filesystem can be found in Figure 2.1 which shows the directory tree of the snort installation. This is will be helpful to understand where all these information need to be installed on a Windows system.

Snort Configuration

The *configuration* files are located in the `etc` sub-directory.

While Snort 2.X configuration files are written in Snort-specific syntax, Snort 3.0 configuration files are written in Lua. Hence, all Snort 3.0 configuration files are written in the Lua language.

The main configuration file is `snort.lua`, it contains Snort's main configuration, allowing the implementation and configuration of Snort inspectors, rules files inclusion, event filters, output, etc.

Instead, `snort_defaults.lua` contains default values for rules paths, networks, ports, wizards, and inspectors, etc.

Another interesting configuration file is `file_magic.lua` because it contains pre-defined file identities based on the hexadecimal representation of the files magic headers. Thanks to these Snort can identify the file types traversing the network (when applicable). Differently from the previous two files this one does not require any modification.

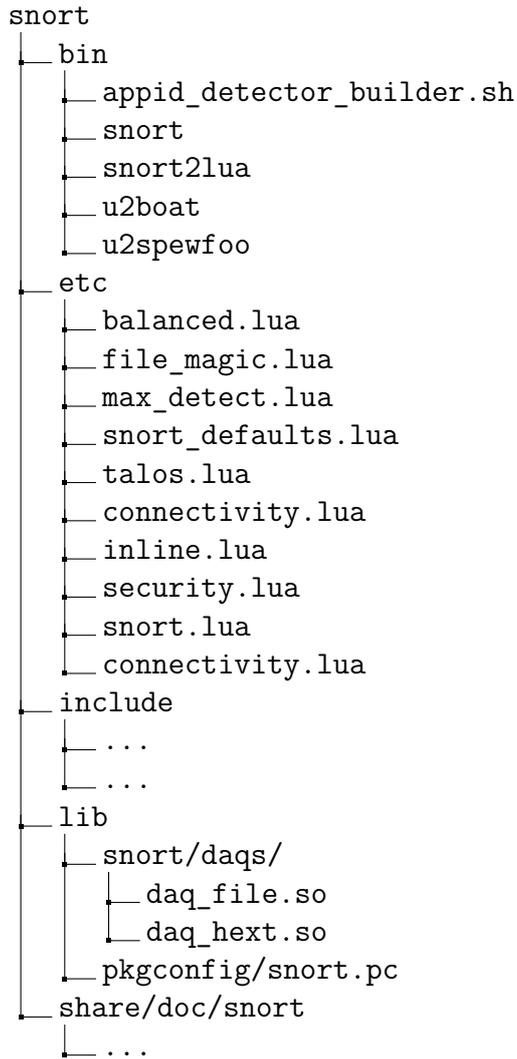


Figure 2.1: Snort Layout on the FileSystem.

Snort Executables

The essential command binaries are placed in the `bin` sub-directory. Snort comes with 4 executables, `snort` is the main executable, `snort2lua` is the tool used to convert a valid Snort 2.X configuration into Lua, finally `u2boat` and `u2spewfoo` are logging-specific and will be analyzed in section [2.1.3](#).

2.1.2 Snort Rules

Snort 3 uses a rule-based language combining signature, protocol, and anomaly inspection methods to detect malicious activity such as denial-of-service (DoS) attacks, Buffer overflows, stealth port scans, SMB probes, and OS fingerprinting attempts. It is capable of performing real-time traffic analysis and packet logging on IP networks.

Snort Rulesets

Rulesets are the backbone for Snorts intrusion detection engine. There are three types[2] of Snort Rules:

- Community Rules: Rules that are either written by people who didn't work for Sourcefire or Cisco or rules that Snort developers want to release to everyone immediately because they cover important vulnerabilities or malware that they want people to have coverage for. This ruleset is updated daily and is a *subset* of the subscriber ruleset. The Community Ruleset is a GPLv2 Talos certified ruleset that is distributed free of charge.
- Registered Rules: This ruleset is also free for use for individuals and businesses. This ruleset is 30 days behind the Snort Subscriber Rule Set and does not contain zero-day threats under the "limited" provision of the Snort Subscriber Rule Set License. This ruleset *does* contain the Community ruleset.
- Subscriber Rules: This is the full Snort Subscriber Ruleset, without delay. Rules that are developed, tested, and approved by Cisco Talos. Subscribers to the Snort Subscriber Ruleset will receive the ruleset in real-time as they are released to Cisco customers to stay current with emerging threats.

By default Snort3 is installed without any ruleset. This is the main reason why the users need to create a specific directory in which the downloaded ruleset will reside.

This particular directory should be named appropriately as it is the one described by the '`snort_defaults.lua`' configuration file. This configuration file highlights the default rules path (`RULE_PATH`) is '`../rules`'.

PulledPork and PulledPork3

Pulledpork is a collaboration effort in the open source community between Cisco and non-Cisco personnel that allows updating and managing Snort rules and Talos open-source IP address block list in a consistent and regular manner.

Differently from its predecessor (written in Perl) *Pulledpork3* is completely written

Snort Sniffing

Snort is either able to listen on an *interface* or read a *packet capture* file. In the latter case, Snort 3 will read and analyze the packets as if they came off the wire. This can be useful for testing and debugging Snort.

In the following we can see the two alternatives:

```
snort -i <iface>
```

```
snort -r <file>
```

Snort Logging

Snort can produce two kinds of log file format depending on snort output plugin option for that files: *pcap capture file* and snort's *unified2*.

Depending on the type of file the user is dealing with (tcpdump capture file or data) there are the following options:

- pcap capture file: Can be read with usual tools wireshark, `tshark -r, tcpdump -r`, or even re-injected in snort with `snort -r2`.
- Unified2: It is the "Native" snort format. Can be read with `u2spewfoo <file>` (included in snort), or converted to a pcap with `u2boat`.

If there is the need to transform it to another alert system (syslog, for example), barnyard2 can be used. Barnyard2^[3] is an open source interpreter for Snort unified2 binary output files.

The unified2 format is used because snort old unique thread design. The time snort spend waiting syslog, screen, etc. to ACK alert is time that snort is not using to analyze packets. So, the way was to dump them in a efficient binary format, and let another program (maybe with low CPU priority) to process them. This will likely be replaced with a FlatBuffer^[4] implementation.

Snort NIDS mode

In this mode Snort will notify the user with specific alerts if any threat was detected. The default logging and alerting mechanisms are to log in decoded ASCII format and use full alerts. The full alert mechanism prints out the alert message in addition to the full packet headers. In Snort 3 there is the possibility to have:

- *Event Logging*: the available options are `alert_csv`, `alert_fast`, `alert_full`, `alert_sfsocket`, `alert_syslog`, `unified2`.

²All these tools rely on libpcap and thus have the `-r` option mapped to the ReadBack capability.

- *Packet Logging*: the available options are `log_codecs`, `log_hext`, `log_pcap`.

Snort Active Response

Last but not least, Snort can have a more active role in securing network by sending active responses to shutdown offending sessions. When active response is enabled, Snort 3 will send TCP RST or ICMP unreachable when dropping a session. The active response plugin comes with three different flavours:

1. **Reject** performs active response to shutdown hostile network session by injecting TCP resets (TCP connections) or ICMP unreachable packets.
2. **React** sends an HTML page to the client, a RST to the server and blocks the flow.
3. **Rewrite** enables overwrite packet contents based on "replace" option in the rules. Using "rewrite" action without "replace" option will raise the appropriate rule alert but will not overwrite the packet payload. Rewrite/replace works for raw packets only.

2.1.4 Snort Dependencies

Required Dependencies

- DAQ for packet IO
- DNET for network utility functions.
- FLEX for JavaScript syntax parser
- C++14 compiler
- HwLoc for CPU affinity management.
- LuaJIT for configuration and scripting.
- OpenSSL for SHA and MD5 file signatures, the `protected_content` rule option, and SSL service detection.
- PCAP for tcpdump style logging.
- PCRE for regular expression pattern matching.
- pkgconfig to locate build dependencies.
- zlib for decompression.

Optional Dependencies

- [Asciidoc](#) to build the HTML manual.
- [CppUTest](#) to run additional unit tests with make check.
- [DBLATEX](#) to build the pdf manual (in addition to asciidoc).
- [Flatbuffers](#) for enabling the flatbuffers serialization format.
- [HS\(Hyperscan\)](#) for the regex and sd_pattern rule options and the hyperscan search engine.
- [ICONV](#) for converting UTF16-LE filenames to UTF8 (usually included in glibc).
- [Libunwind](#) for printing a backtrace when a fatal signal is received.
- [lzma](#) for decompression of SWF and PDF files.
- [SafeC](#) for additional runtime bounds checks on certain legacy C-library calls.
- [source-highlight](#) to generate the dev guide.
- [w3m](#) to build the plain text manual.
- [UUID](#) for unique identifiers.

2.2 Related Work

Among the open source technologies used in intrusion detection and prevention systems (IDS/IPS) we can identify Snort, Suricata and Zeek (formerly known as Bro)[5].

The main difference is that Snort and Suricata are *rule-based engines* and as such are designed to detect an exception. On the other hand Zeek is an intrusion detection system that works differently because of its focus on *network analysis*. Thus, while rules-based engines are designed to detect an exception, Zeek looks for specific threats and triggers alerts.

Other related works include Microsoft Defender for Identity and ClamAV, analyzed in Section 2.2.2 and 2.2.3 respectively.

2.2.1 Suricata

It is of paramount importance to highlight that a Windows port was barely supported in Snort 2 (mainly because the Open Source community did the initial port) and no support as of today was expected for Snort 3.

Zeek is still at an early stage of this porting procedure (mainly driven by Brim Security) as it emerges from an issue on their official Github repository[6].

Differently Suricata provides a Windows implementation and this is why has been analyzed (Section 2.2.1) being a good reference for a native Windows port of Snort 3.

From the Suricata homepage[7]:

Suricata is the leading independent open source threat detection engine. By combining intrusion detection (IDS), intrusion prevention (IPS), network security monitoring (NSM) and PCAP processing, Suricata can quickly identify, stop, and assess the most sophisticated attacks.

Despite the availability of a Windows port for Suricata the advantages of having an internal product like Snort 3 in the Cisco Secure endpoint are worth porting Snort 3 to Windows. These advantages include a direct communication channel with the developers which can be used for bug fixing and in order to receive rule updates before everyone else.



Figure 2.3: Suricata logo

2.2.2 Microsoft Defender for Identity

Microsoft Defender for Identity[8] cloud service helps protect the enterprise hybrid environments from multiple types of advanced targeted cyber attacks and insider

threats. This is strictly related to what the *Snort Integration into Cisco Secure Endpoint* will cover.

From the "What is Microsoft Defender for Identity?"[9] section in the MSDN:

Microsoft Defender for Identity (formerly *Azure Advanced Threat Protection*, also known as *Azure ATP*) is a cloud-based security solution that leverages your on-premises Active Directory signals to identify, detect, and investigate advanced threats, compromised identities, and malicious insider actions directed at your organization.

Defender for Identity enables SecOp analysts and security professionals struggling to detect advanced attacks in hybrid environments to:

- Monitor users, entity behavior, and activities with learning-based analytics
- Protect user identities and credentials stored in Active Directory
- Identify and investigate suspicious user activities and advanced attacks throughout the kill chain
- Provide clear incident information on a simple timeline for fast triage

It is important to highlight that in Microsoft Defender for Identity release 2.156, released on July 25th, 2021, the NPCAP driver executable is included in its sensor installation package. This will ensure that Npcap driver will be used instead of the WinPcap driver, as WinPcap is no longer supported[10]:

The Microsoft Defender for Identity team is currently recommending that all customers deploy the Npcap driver before deploying the sensor on a domain controller. This will ensure that Npcap driver will be used instead of the WinPcap driver. [...]

WinPcap is no longer supported and since it's no longer being developed, the driver cannot be optimized any longer for the Defender for Identity sensor. Additionally, if there is an issue in the future with the WinPcap driver, there are no options for a fix.

2.2.3 ClamAV

ClamAV is an open source antivirus engine for detecting trojans, viruses, malware other malicious threats.

From the ClamAV homepage[11]:

ClamAV is an open source (GPLv2) anti-virus toolkit, designed especially for e-mail scanning on mail gateways. It provides a number of utilities including a flexible and scalable multi-threaded daemon, a command line scanner and advanced tool for automatic database updates. The core of the package is an anti-virus engine available in a form of shared library.

Clam AntiVirus is highly cross-platform, thus it has been analyzed because it shares some similarities with Snort being in the C/C++ world and using CMake, so it definitely is a good resource and reference for CMake and Win32.



Figure 2.4: ClamAV logo

Chapter 3

Packet Capture

Packet Capture or PCAP (also known as libpcap[12]) is an application programming interface (API) that captures live network packet data from OSI model Layers 2-7.

While PCAP is an abbreviation of packet capture, that is not the API's proper name.

While Unix-like systems implement pcap in the libpcap library, Windows systems relied for several years on a port of libpcap named WinPcap[13] that is no longer supported nor developed. This is why nowadays it has been replaced by a port named Npcap[14] (for Windows 7 and later) that is continuously supported.

The main focus of this section is to analyse the Windows Network Driver, Libpcap and the different ports of libpcap available in Windows:

1. **Libpcap**
 - **MSYS2 Libpcap** and **MinGW Libpcap**.
2. **WinPCAP**: Windows Packet Capture.
3. **Win10PCAP**: WinPcap for Windows 10 (NDIS 6.x driver model).
4. **NPcap**: Windows Packet Capture Library & Driver.
5. **WinDivert**: Windows Packet Divert.

3.1 Windows Network Driver

WinPcap first and Npcap later are architectures for packet capture and network analysis for the Win32 platforms. They are defined as architectures because

packet capture is a low level mechanism that requires a strict interaction with the network adapter and with the operating system, in particular with its networking implementation, so a simple library is not sufficient.

3.1.1 Windows Network Driver Architecture

The architecture (Figure 3.1) includes a kernel-level packet filter, a low-level dynamic link library (`Packet.dll`), and a high-level and system-independent library (`wpcap.dll`).

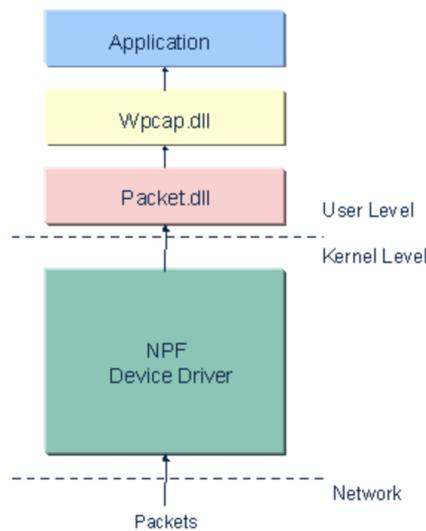


Figure 3.1: Main components of the Windows Network Driver Architecture.

The Windows network driver uses a capture system to bypass the operating systems' protocol stack to access the raw data transiting on the network.

Thus, it uses a *device driver*, called *Netgroup Packet Filter (NPF)*, to directly interact with the network interface drivers. This driver offers basic features like packet capture and injection, as well as more advanced ones like:

- *Filtering System:* To restrict a capture session to a subset of the network traffic.
- *Monitoring Engine:* To obtain statistics on the traffic (e.g. the network load or the amount of data exchanged between two hosts).

The capture system exports an *interface* that user-level applications use to exploit the features provided by the kernel driver. Two different libraries are provided:

- `Packet.dll`: a low-level API to directly access the functions of the driver, with a programming interface independent from the Microsoft OS.

- `wpcap.dll`: a more powerful set of high level capture primitives that are compatible with `libpcap`. These functions enable packet capture in a manner that is independent of the underlying network hardware and operating system.

3.1.2 NPF and NDIS

The NDIS (Network Driver Interface Specification) is a standard that defines the communication between a network adapter (or, better, the driver that manages it) and the protocol drivers (that implement for example TCP/IP). Main NDIS purpose is to act as a wrapper that allows protocol drivers to send and receive packets onto a network (LAN or WAN) without caring about either the particular adapter or the particular Win32 operating system.

NDIS supports four types of network drivers:

- *Miniport drivers*. Miniport drivers directly manage NICs. The miniport drivers interface directly to the hardware at their lower edge and at their upper edge present an interface to allow upper layers to send packets on the network, to handle interrupts, to reset the NIC, to halt the NIC and to query and set the operational characteristics of the driver.

Miniport drivers implement only the hardware-specific operations necessary to manage a NIC, including sending and receiving data on the NIC. Operations common to all lowest level NIC drivers, such as synchronization, is provided by NDIS. Miniports do not call operating system routines directly; their interface to the operating system is NDIS.

- *Intermediate drivers*. Intermediate drivers interface between an upper-level driver such as a protocol driver and a miniport. To the upper-level driver, an intermediate driver looks like a miniport. To a miniport, the intermediate driver looks like a protocol driver. An intermediate protocol driver can layer on top of another intermediate driver although such layering could have a negative effect on system performance. A typical reason for developing an intermediate driver is to perform media translation between an existing legacy protocol driver and a miniport that manages a NIC for a new media type unknown to the protocol driver. An intermediate driver cannot communicate with user-mode applications, but only with other NDIS drivers.
- *Filter drivers*. Filter drivers can monitor and modify traffic between protocol drivers and miniport drivers like an intermediate driver, but are much simpler. They have less processing overhead than intermediate drivers.
- *Transport drivers* or *protocol drivers*. A protocol driver implements a network protocol stack such as IPX/SPX or TCP/IP, offering its services over one

or more network interface cards. A protocol driver services application-layer clients at its upper edge and connects to one or more NIC driver(s) or intermediate NDIS driver(s) at its lower edge.

NPF is implemented as a *filter driver*. In order to provide complete access to the raw traffic and allow injection of packets, it is registered as a modifying filter driver. NPF is able to perform a number of different operations: capture, monitoring, dump to disk, packet injection.

3.1.3 Relationship of WPCAP.DLL and PACKET.DLL

As previously mentioned when moving from UNIX/LINUX PCAP libraries to Windows ones we are also switching from a single library `libpcap.so` to two DLLs, `wpcap.dll` and `Packet.dll`. Some background on `libpcap` is needed to shed some light on the relationship between these two DLLs.

`Libpcap` was originally the code in `tcpdump` that hid from the bulk of `tcpdump` the differences between the mechanisms provided by various flavors of UNIX to allow raw link-layer packets to be transmitted and received; `tcpdump` merely *receives* link-layer packets, and *doesn't send* them, so `libpcap` doesn't have any routines to transmit packets ¹.

If on the one hand, `wpcap.dll` implements the `libpcap` API (plus some extensions) for Win32 systems, on the other hand, `Packet.dll` and the drivers for various Win32 operating systems, provide a Win32-specific raw link-layer packet access mechanism.

- **Wpcap.dll** provides an API that should work on BSD, Linux, Solaris, Windows etc., allowing applications to *capture packets* on a network without themselves having to do that capture differently on different OSes.
- **Packet.dll** provides a Win32-specific API for capturing **and** sending packets, just as the BPF driver on BSD, PF_PACKET sockets on Linux, etc. provide APIs that are somewhat OS-specific for capturing and sending packets on those OSes.

We can therefore identify the `Packet.dll` routines (names beginning with `Packet`) from the `Packet.dll` API and the `wpcap.dll` routines (names beginning with `pcap_`) from the `libpcap` API.

¹There's no reason why it couldn't have those routines; it just doesn't happen to have them.

The libpcap API is a somewhat "higher-level" API, hiding, as it does, various low-level details of BPF or PF_PACKET sockets or DLPI or `Packet.dll` or anything that might make it easier to use. However, this implies that it might also mean that it wouldn't allow you to do some things you could do by directly using the `Packet.dll` API.

The packet capture driver is a device driver that adds to Windows the ability to capture and send raw packets in a way similar to the Berkeley Packet Filter of UNIX kernels.

If it's just a packet capture program, the libpcap API, rather than the raw `Packet.dll` API, may be easier.

3.2 LibPCAP

Libpcap is a portable, open source C/C++ library providing a high level interface for network traffic capture.

Created in 1994 by the tcpdump developers in the Network Research Group at Lawrence Berkeley Laboratory as a part of a research project to improve TCP and Internet gateway performances.

The Packet Capture library provides a high level interface to packet capture systems. All packets on the network, even those destined for other hosts, are accessible through this mechanism. It also supports saving captured packets to a "savefile", and reading packets from a "savefile".

3.2.1 MSYS2 and MinGW-w64 LibPcap

In this section it is analyzed the Libpcap package provided by MSYS2[15] and the one from MinGW-w64[16]. These two Software Distribution and Building Platform for Windows will be analyzed in section 5.1.1 and 5.1.2 for their characteristics and the possibility to complete a successful port of Snort to Windows.

The Libpcap module has been analyzed because after successfully obtaining a binary in MSYS2, it was successfully able to readback pcap files but it was not able to passively sniff an interface. This issue was accompanied with the following error message:

```
Couldn't start DAQ instance: live packet capture not supported|
on this system (-1)
Analyzer: Failed to start DAQ instance
```

This error message is a combination of three different error sources:

- Snort - Couldn't start DAQ instance:

- LibPCAP - live packet capture not supported on this system
- LibDAQ - (-1)

Unfortunately, the (-1) error message from LibDAQ is mapped to the `DAQ_ERROR` macro which is the generic error one, thus it is not providing further information about the issue (See Listing 3.1).

Listing 3.1: LibDAQ `DAQ_ERROR` error code.

```
1 #define DAQ_ERROR      -1 /* Generic error */
```

A deeper analysis of Snort reveals that the function returning the issue is the `daq_pcap_start` one which has the following function hierarchy: Being the libp-

```
pcap_daq_start          // From LibDAQ pcap/daq_pcap.c file
├─ pcap_create          // Create and init the pcap structure
├─ pcap_create_interface // Diff OS env call diff functions
```

Figure 3.2: `daq_pcap_start` function hierarchy.

cap library installed using pacman package manager there is no access to the source code and even debugging Snort there is no concrete possibility to step into `pcap_create()` and see where the error generates from.

However, according to the man page of `pcap_create()` [17]:

```
pcap_create() returns a pcap_t * on success and NULL on failure. If
NULL is returned, errbuf is filled in with an appropriate error message.
errbuf is assumed to be able to hold at least PCAP_ERRBUF_SIZE chars.
```

Indeed, the `pcap_create` is failing and thus the string `errbuf` is filled in with the error message "live packet capture not supported on this system".

Full understanding of the reason why live packet capture is not supported on MSYS2 (and MinGW-w64) can be retrieved reversing the MSYS libpcap lib file (`msys-pcap-1.dll`).

Reverse engineering this `.dll` file it is easy to understand that the analyzed function is just a placeholder returning the error string because MSYS (and MinGW-w64) are *not* supporting Live Packet Capture as it can be seen from Figure 3.3 and 3.4.

3.3 WinPCAP: Windows Packet Capture

For many years, WinPcap has been recognized as the industry-standard tool for link-layer network access in Windows environments, allowing applications to capture and transmit network packets bypassing the protocol stack, and including

```

; Exported entry 96. pcap_create_interface

; Attributes: bp-based frame

public pcap_create_interface
pcap_create_interface proc near

arg_0= qword ptr 10h
arg_8= qword ptr 18h

push rbp
mov rbp, rsp
sub rsp, 20h
mov [rbp+arg_0], rcx
mov [rbp+arg_8], rdx
mov rax, [rbp+arg_8]
mov r8d, 100h
lea rdx, aLivePacketCapt ; "live packet capture not supported on th"...
mov rcx, rax
call strlcpy
mov eax, 0
add rsp, 20h
pop rbp
retn
pcap_create_interface endp

```

Figure 3.3: MSYS2 disassembled pcap_create_interface function.

```

sub_38E6F82A0 proc near
sub rsp, 28h
mov r9, 0FFFFFFFFFFFFFFFh ; MaxCount
lea r8, aLivePacketCapt ; "live packet capture not supported on th"...
mov rcx, rdx ; Destination
mov edx, 100h ; SizeInBytes
call cs:__imp_strncpy_s
xor eax, eax
add rsp, 28h
retn
sub_38E6F82A0 endp

```

Figure 3.4: MinGW disassembled pcap_create_interface function.

kernel-level packet filtering, a network statistics engine and support for remote packet capture.

WinPcap consists of a driver that extends the operating system to provide low-level network access and a library that is used to easily access low-level network layers.

This library also contains the Windows version of the well-known libpcap Unix API.

Inside the Windows kernel, WinPcap runs as a protocol driver. It's at the same level of `tcpip.sys`, and like the TCP/IP stack it receives the packets from the underlying NIC driver, but only when at least one WinPcap-based tool is capturing. This means that when WinPcap is installed but not capturing, the impact on the system is nonexistent.

Note in particular that the WinPcap driver is loaded inside the kernel only when the first capture application opens an adapter after a machine boot.

When WinPcap runs, it doesn't directly interact with TCP/IP. However especially under high network loads, the WinPcap activity (in particular the one at software interrupt level) will impact on TCP/IP responsiveness.

Unfortunately, WinPcap has been unmaintained since 2013, date of the last official WinPcap release[18]:

8 March, 2013

As of today, WinPcap v4.1.3 is available in the download section of the WinPcap website. This release adds support for Windows 8 and Server 2012, and fixes a couple of security issues in the WinPcap driver that could cause an OS crash.

WinPcap itself has identified its successor and passed the torch to Npcap in 2018 with an official notice[19] on their "News and Release" website section:

15 September 2018

WinPcap, though still available for download (v4.1.3), has not seen an upgrade in many years and there are no road map/future plans to update the technology. While community support may persist, technical oversight by Riverbed staff, responses to questions posed by Riverbed resources, and bug reporting are no longer available.

Gordon Lyon, Nmap project founder, has created Npcap, a packet capture library for Windows, that includes WinPcap compatibility and may be a suitable replacement for WinPcap and WinPcap Pro. Information can be found at <https://nmap.org/npcap/>.

When Microsoft will remove NDIS 5 or cease the grandfathering of older less secure driver signatures, WinPcap will cease working.

3.4 Win10PCAP: WinPcap for Windows 10

Win10Pcap[20] is a new WinPcap-based Ethernet packet capture library developed by Daiyuu Nobori at University of Tsukuba.

Differently from the original WinPcap - which is based on the NDIS 5.x driver model - Win10PCAP integrated the WinPcap codes into the NDIS 6.x driver model to work stably with Windows 10. Indeed, it is compatible with Windows 10 on both x86 and x64 platforms.

Win10PCAP also added the code to support the capability to capture IEEE802.1Q VLAN tags in Ethernet frames which the original WinPcap was not supporting.

Despite the great potential of Win10PCAP, we are not going to explore this alternative further because the source code has not seen an updated for years².

The kernel-mode Win10Pcap device driver has obtained the "Windows 10 Compatible" logo from Microsoft on June 8, 2015 (Figure 3.5).



Compatible with Windows 10 on both x86 and x64 platforms.

Figure 3.5: Win10Pcap device driver "Windows 10 Compatible" logo from Microsoft.

3.5 NPcap: Windows Packet Capture Library & Driver

Npcap is an *architecture* for packet capture and network analysis for Windows operating systems from the *Nmap Project*[21]. Npcap began in 2013 as improvement to WinPcap, but has begun the Windows packet capture library "par excellence" since then with hundreds of releases improving Npcap's speed, portability, security, and efficiency.

²The last commit is dated 8 October 2015.

Npcap supports all Windows architectures (x86, x86-64, and ARM) and all versions of Windows and Windows Server that Microsoft themselves still support. Npcap works on Windows 7 and later by making use of the NDIS 6 Light-Weight Filter (LWF) API which is faster than the deprecated NDIS 5 API used by WinPcap.

Npcap provides facilities to:

- capture raw packets, both the ones destined to the machine where it's running and the ones exchanged by other hosts (on shared media)
- filter the packets according to user-specified rules before dispatching them to the application
- transmit raw packets to the network
- gather statistical information on the network traffic
- Loopback Packet Capture and Injection: Npcap is able to sniff loopback packets through the Windows Filtering Platform (WFP). Packet injection works as well with the `pcap_inject()` function.

This set of capabilities is obtained by means of a device driver, which is installed inside the networking portion of the Windows kernel, plus a couple of DLLs (from the SDK).

However, Npcap receives and sends the packets independently from the host protocols, like TCP/IP. This means that it isn't able to block, filter or manipulate the traffic generated by other programs on the same machine: it simply "sniffs" the packets that transit on the wire. Therefore, it does not provide the appropriate support for applications like traffic shapers, QoS schedulers and personal firewalls.

3.5.1 NPcap License

The Npcap project is funded by selling *NPCAP OEM*. This special version of Npcap includes enterprise features such as the *silent installer* and special license rights allowing to redistribute Npcap with their products or to install it on more systems within the organization with easy enterprise deployment. Unluckily, the Npcap free license only allows five installs and does not allow for any redistribution.

3.6 WinDivert: Windows Packet Divert

WinDivert[22], developed by the *ReQrypt*[23] organization, is a user-mode packet capture-and-divert package for Windows 2008, Windows 7, Windows 8, Windows 10 and Windows 2016.



Figure 3.6: Npcap Silent Installation Error Banner.

WinDivert is dual-licensed under the GNU Lesser General Public License (LGPL) Version 3 or the GNU General Public License (GPL) Version 2 and features a silent installation, two characteristics that Npcap has not.

WinDivert allows user-mode applications, with a simple yet powerful API, to:

- capture (and modify) network packets
- filter/drop network packets
- sniff network packets
- (re)inject network packets
- fully support IPv6
- support loopback (localhost) traffic

For all of these reasons WinDivert can be used to implement user-mode packet filters, packet sniffers, firewalls, NAT, VPNs, tunneling applications, etc.

Indeed, WinDivert is at the basis of well-known projects such as *Suricata* (Network threat detection engine) and *mitmproxy* (Interactive SSL-capable intercepting HTTP proxy).

3.6.1 WinDivert Architecture

The `WinDivert.sys` driver is installed below the Windows network stack. The following sequence of actions occurs:

1. A new packet enters the network stack and is intercepted by the `WinDivert.sys` driver.
2. (a) If the packet matches the PROGRAM-defined filter, it is diverted.

- (b) If the packet does not match the filter, the packet continues as normal.
3. PROGRAM either *drops*, *modifies*, or *re-injects* the packet.

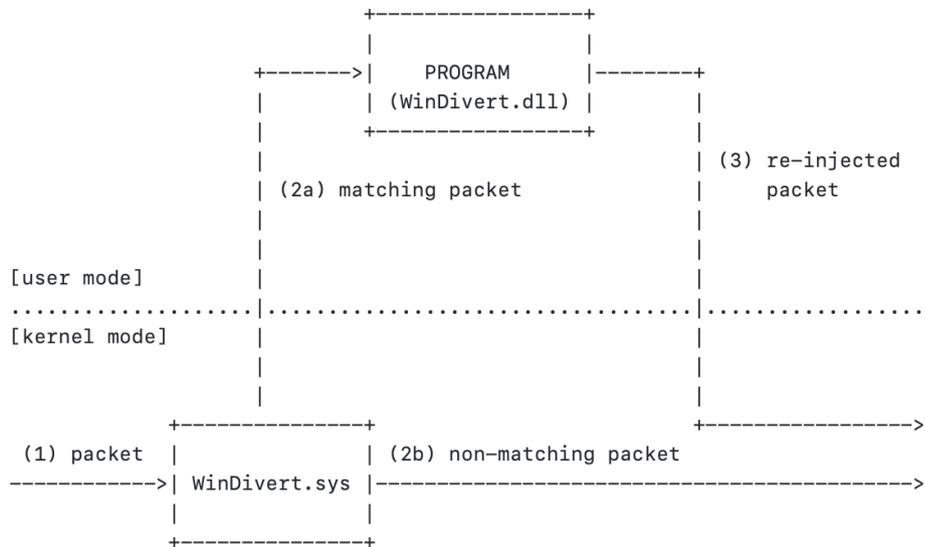


Figure 3.7: The basic architecture of WinDivert.

Feature	WinPCAP	Win10PCAP	NPcap	WinDivert
Actively maintained and supported	No	No	Yes	Yes
License	BSD-style	GPL	Free for personal use	LGPLv3/GPLv2
Capture Loopback traffic	No	?	Yes	Yes
Inject Loopback traffic	No	?	Yes	Yes
Protocol Driver	NDIS 5	NDIS 6	NDIS 6	?
Block, Filter, Manipulate traffic	Yes	Yes	No	Yes
Silent Installer	Yes	Yes	No	Yes

Table 3.1: Summary of the PCAP Libraries and their key features.

3.7 Which PCAP Module for Snort 3?

As highlighted from the PCAP modules available for Windows we are able to draw some conclusion for one of the cores of the Snort NIDS routine.

automate its installation.

As further evidence for the great fit of WinDivert there is the proof that Suricata (analyzed in Section 2.2) in layer 4 inline mode uses WinDivert on Windows [24].

Chapter 4

LibDAQ: The Data Acquisition Library

Snort 2.9 introduced the Data Acquisition library (LibDAQ)[25], for packet I/O. Since then the Data Acquisition library has become the most important dependency of Snort, no matter the version.

4.1 LibDAQ Introduction

The DAQ replaces direct calls into packet capture libraries like libPCAP with an abstraction layer to make it effortless to add additional software or hardware packet capture implementations. DAQ supports AFPacket, BPF, Divert, FST, Netmap, NFQ, PCAP, Savefile, Trace, and DUMP which is used for testing.

The DAQ is essentially an abstraction layer coming with a suite of pluggable modules that can be selected at run-time to interact with a data source (traditionally a network interface or network data plane). This makes switching from passive to inline mode easy, and does not require a recompile of the snort core.

The advantage of having an abstraction layer is that it makes Snort agnostic of how it gets its data. The Data Acquisition module will handle the data acquisition part so Snort itself can remain generic regardless of architecture (or reading pcaps).

The main focus of this section is to analyse the LibDAQ library and how it can be ported to Windows while preserving its core features, the DAQ Modules.

4.2 LibDAQ Configuration

Users can select and configure the DAQ when Snort can operate in three different DAQ modes namely *passive (tap)*, *inline*, and *read-file*:

- **Inline:** When Snort is in Inline mode, it acts as an IPS allowing drop rules to trigger. Snort can be configured to run in inline mode using the command line argument `-Q` and snort config option `daq-mode` as follows:

```
snort -Q
config daq-mode inline
```

- **Passive:** When Snort is in Passive mode, it acts as a IDS. Snort can be configured to passive mode using the snort config option `daq-mode` as follows:

```
config daq-mode passive
```

- **Read-File:** Read-File mode is used to read packet capture files. It is useful to test and troubleshoot Snort. Snort can be configured to run in read-file mode using the command line option (`-enable-inline-test`) or using the snort config option `daq-mode` as follows:

```
snort -r
config daq-mode read-file
```

If the mode is not set explicitly, `-Q` will force it to inline, and if that hasn't been set, `-r` will force it to read-file, and if that hasn't been set, the mode defaults to passive. Also, `-Q` and `-daq-mode inline` are allowed, since there is no conflict, but `-Q` and any other DAQ mode will cause a fatal error at start-up.

4.3 DAQ Modules Features

Applications using LibDAQ use the library API defined in `daq.h` to load, configure, and interact with pluggable DAQ modules.

On top of the base DAQ library we can plug-in many, so called, Modules.

Each DAQ module implements some or all parts of the DAQ module API depending on its type and capabilities. There are two main classes of DAQ modules:

- Base modules: provide a full-fledged and independently usable implementation of the DAQ module API.

- Wrapper modules: provide a subset of the API that is applied in a decorator pattern when combined with a base module.

Different DAQ modules types¹ can be remarked:

- *Readback* : the module can read read from a PCAP file;
- *Live* : the module can open live interfaces;
- *Inline* : the module can form an inline bridge;
- *Multi (Instance)* : the module can be instantiated multiple times;
- *Unpriv* : the module can run unprivileged;
- *Wrapper* : the module must decorate another DAQ module;

4.4 DAQ Modules

The most influential DAQ modules will be now analyzed in more detail. The FST module will not be analyzed because it is currently in an early and highly experimental state. Table 4.1 summarizes the main features of every DAQ module.

4.4.1 BPF Module

The BPF module is a *wrapper* DAQ module that implements filtering on packet reception when given a Berkeley Packet Filter (BPF) to operate with.

It adds the BPF capability to the module stack that it is part of and will update the filtered count in the DAQ statistics. Filtered packet messages will be immediately finalized with a PASS verdict.

Since this module uses the BPF implementation from LibPCAP the only requirement is a version of libpcap greater or equal than 1.0.0.

4.4.2 Dump Module

The Dump module is a wrapper DAQ module that presents the configuration stack as inline-interface- and injection-capable. All packet messages that are finalized with a passing verdict (PASS, REPLACE, WHITELIST, IGNORE) or injected will be written to a PCAP savefile. By default, the packet capture file will be named `inline-out.pcap` in the current directory.

¹taken from libdaq/api/daq_common.h

The Dump DAQ module also supports capturing received packets to a separate PCAP savefile. This is disabled by default, but can be enabled with the `dump-rx` variable. The `dump-rx` variable takes an optional argument for the filename to dump received packets to; it defaults to `inline-in.pcap` if no argument is given.

When running with multiple instances, the both the TX and RX output filenames will be mangled to start with the instance ID followed by an underscore. For example, the default TX output filename would be `2_inline-out.pcap` for the second instance. Both the TX and RX output filenames must be bare (no directory structure, relative nor absolute) in such a configuration.

The only requirement for this particular DAQ module is having a libpcap version greater or equal than 1.0.0.

4.4.3 Netmap Module

The Netmap DAQ module is built on top of the netmap project[26], a framework for very high speed packet I/O. It is available on both FreeBSD and Linux with varying amounts of preparatory setup required.

Although on the Netmap Project Github repository it is reported that "*Netmap has been ported to Windows in summer 2015 [...]*"[27], this is an old and unmaintained project. This is why it is better to look to some other windows packet intercept toolkit for a Windows port of this module (perhaps some framework used to build VPNs).

4.4.4 PCAP Module

This is the *default* DAQ Module used by Snort. The PCAP module is built around the LibPCAP library to support both read-file and passive interface sniffing modes. Since the input specifications are directly passed to LibPCAP the input is perfectly mapped to the `-i` or `-r` options of TCPDUMP.

The PCAP DAQ module defaults to listening in promiscuous mode, meaning that can intercept and read in its entirety each network packet that arrives. Moreover, the PCAP DAQ module defaults to using immediate (less-buffered or unbuffered) delivery mode.

The only requirement for this particular DAQ module is having a libpcap version greater or equal than 1.5.0.

4.4.5 Savefile Module

The PCAP DAQ module is not the only one able to perform Readback operations. Indeed, the Savefile DAQ module is designed for performance-optimized readback of traditional pcap savefiles².

The savefile DAQ module will map an entire pcap savefile into memory and then directly access the contents to acquire DAQ message data. Compared to the PCAP DAQ module, this eliminates both the overhead of the libpcap API interface itself as well as the copying of packet data into the DAQ message pool's data buffers.

However, one drawback of this module, as mentioned above, is that the contents of the entire pcap savefile will be mapped into memory and thus will not be released until the DAQ module is stopped.

This is why it is of paramount importance to make sure to avoid loading a file that is too large to fit in memory because this would run the system out of memory once all of the packets in the file have been accessed and everything has been paged into active memory.

Only pcap savefiles with Ethernet data link types (`DLT_EN10MB`) are supported. Another key aspect to take into account is that the beginning of message data in messages received by the application can easily be positioned at unaligned memory addresses. This is a major issue for architectures that cannot handle unaligned memory accesses.

4.4.6 Trace Module

It is a *wrapper* DAQ module that records information about packet message verdicts, injected packet messages, and IOCTL calls that it intercepts to a text file (by default, `inline-out.txt`). The Trace module presents the configuration stack as being capable of inline interface operation, blocking, and injection.

Injected packet messages and unrecognized IOCTLs will have their contents dumped in hexadecimal format. On the other hand, recognized IOCTLs will get more of a pretty-print style output. Verdicts on packet messages will be recorded on a single line together with some identifying information from the packet message header.

²Only pcap savefile format version 2.4 is supported.

Module \ Type	Readback	Live	Inline	Multi	Unpriv	Wrapper
AFPacket		X	X	X	X	
BPF			X		X	X
Divert		X	X	X		
Dump			X	X	X	X
FST					X	X
Netmap		X	X	X		
NFQ		X	X	X		
PCAP	X	X		X	X	
Savefile	X		X	X		X
Trace			X		X	X

Table 4.1: Summary of the DAQ modules and associated DAQ types.

Chapter 5

Porting Snort3 to Windows

The requirement is to have a binary distribution of Snort3 that is able to run on Windows. In the following section are investigated the main alternatives that are feasible to port a Linux program to Windows, how the Snort 3 dependencies can be installed and the core changes that the codebase needs in order to have at least readback and live interface sniffing capabilities.

5.1 Running Linux programs on Windows

To run a Linux program on Windows, the following options are available[28]:

- Run the program as-is on the Windows Subsystem for Linux (WSL)[29]. In WSL your program executes directly on the machine hardware, not in a virtual machine.
- Run the program as-is in a Linux virtual machine[30] or Docker container[31], either on the local machine or on Azure.
- Compile the program using gcc or clang in the MinGW or MinGW-w64 environments, which provide a compile-time translation layer from Linux to Windows system calls.
- Compile and run the program using gcc or clang in the Cygwin[32] environment, which provides a more complete Linux environment on Windows compared to MinGW or MinGW-w64.
- Manually port your code from Linux and compile for Windows using Microsoft C++ (MSVC)[33]. This involves refactoring platform-independent code into separate libraries, and then re-writing the Linux-specific code to use Windows-specific code (for example, Win32 API).

As a first consideration, using WSL or WSL2 is not suitable for our purpose since WSL is just a Hyper-V[34] VM that has its own virtual Hyper-V network adapter. The WSL architecture uses virtualized networking components[35]. This means that in initial preview builds WSL 2 will behave more similarly to a virtual machine, e.g: WSL 2 will have a different IP address than the host machine. It is not visible the "network namespace" of the windows host, but more something like inside a docker container on Linux. The same holds for a Linux VM or Docker container.

Therefore, as an initial starting point Snort has been compiled using gcc in the MSYS2 and MinGW-w64 environment. After encountering the libpcap issue referenced in Section 3.2.1 a Windows port of Snort in Windows using MSVC (capable of reading pcap files and sniffing interactively an interface) has been achieved.

5.1.1 MSYS2

MSYS2 is a collection of tools and libraries providing an easy-to-use environment for building, installing and running native Windows software.

It consists of the mintty command line terminal, bash, git and subversion (for version control systems), and even build systems like autotools, all based on a modified version of Cygwin. Despite being heavily based on Cygwin, the main focus of MSYS2 is to provide a build environment for native Windows software and the Cygwin-using parts are kept at a minimum. MSYS2 provides up-to-date native builds for GCC, mingw-w64, CMake, OpenSSL just to name a few.

To provide easy installation of packages and a way to keep them updated it features a package management system called Pacman (the package manager of Arch Linux).

5.1.2 MinGW-w64

Mingw-w64 is an advancement of the original mingw.org project, created to support the GCC compiler on Windows systems. It has forked it in 2007 in order to provide support for 64 bits and new APIs. It has since then gained widespread use and distribution.

5.1.3 Microsoft Visual C++ (MSVC)

Microsoft Visual C++ (MSVC) is a compiler for the C and C++ programming languages by Microsoft. Microsoft Visual C++ is the de facto standard for native Windows applications.

For this reason and the fact that two of the main dependencies of Snort, LibDNET

and Npcap, are available in Windows only as a MSVC compiled libraries using Microsoft Visual C++ was an easy pick.

5.2 Installing Snort Dependencies for MSVC

Snort dependencies have been installed using a combination of packages installed with:

- [Chocolatey](#)[36]: A *software management automation* tool for Windows that wraps installers, executables, zips, and scripts into compiled packages.
- [Vcpkg](#)[37]: A free C/C++ *package manager* for acquiring and managing libraries.

Those packages that could not be installed using one of these two package managers have been installed from source and will be analyzed individually. In [Table 5.1](#) summarizes the required packages for building Snort and LibDAQ in MSVC, and their installation method.

Dependency	Installation method
CMake	<code>choco install cmake</code>
DAQ	Source code (ported to Windows)
DNET	Nmap Libdnet-stripped
FLEX	<code>choco install winflexbison3</code>
HwLoc	<code>vcpkg install hwloc</code>
LuaJIT	<code>vcpkg install luajit</code>
OpenSSL	<code>choco install openssl</code>
PCAP	Npcap SDK + Npcap Driver
PCRE	<code>vcpkg install pcre</code>
pkgconfig	<code>choco install pkgconfiglite</code>
zlib	<code>vcpkg install zlib</code>

Table 5.1: Snort dependencies and relative installation method on Windows.

5.2.1 Npcap Install

In order to install Npcap[38] it is needed to:

- run the *Npcap 1.60 executable installer* for Windows 7/2008R2, 8/2012, 8.1/2012R2, 10/2016, 2019 (x86, x64, and ARM64),
- and build Snort 3 against the *Npcap SDK*.

5.2.2 Libdnet Install

Libdnet provides a simplified, portable interface to several low-level networking routines, including network address manipulation, kernel arp cache and route table lookup and manipulation, network firewalling, network interface lookup and manipulation, IP tunnelling, and raw IP packet and Ethernet frame transmission.

Despite the fact that Libdnet comes with a developers package, it has the issue of being only compatible with 32-bit applications. Starting from April 2005 Dug Song's Libdnet networking library has been stripped down for inclusion within Nmap and is now available as an MSBuild project[39] whose settings are stored in the XML project file *libdnet-stripped.vcxproj*.

The Microsoft Build Engine[40] is a platform for building applications. This engine, which is also known as MSBuild, provides an XML schema for a project file that controls how the build platform processes and builds software. Visual Studio uses MSBuild, but MSBuild doesn't depend on Visual Studio. By invoking *msbuild.exe* on a project or solution file, there is the possibility to orchestrate and build products in environments where Visual Studio isn't installed.

5.2.3 Vcpkg Install

Vcpkg is a free C/C++ *package manager* for acquiring and managing libraries.

It uses CMake internally as a build scripting language. This is because CMake is very popular for C++ projects and for cross-platform open source libraries.

The best way to use installed libraries with cmake is via the toolchain file `scripts\buildsystems\vcpkg.cmake`. To use this file, it is simply need to add it onto the CMake command with `-DCMAKE_TOOLCHAIN_FILE`. Vcpkg has been used to build both LibDAQ and Snort3. The preliminary dependencies installed using vcpkg are:

- *dirent*: a C/C++ programming interface that allows programmers to retrieve information about files and directories under Linux/UNIX.
- *getopt*: An implementation of the `getopt()` function to parse the command-line arguments.
- *threads*: pthreads for windows.

- *dlfcn-win32*: A set of functions that allows runtime dynamic library loading.

The other mandatory dependencies required by Snort installed via `vcpkg` are *hwloc*, *luajit*, *zlib*, and *liblzma*.

5.3 Porting LibDAQ to Windows

LibDAQ needs to be installed from source. From the `README.md` "Build and Install" section on their GitHub page[41]:

LibDAQ is a standard autotools project and builds and installs as such:

```
./configure
make
make install
```

If building from git, you will need to do the following to generate the configure script prior to running the steps above:

```
./bootstrap
```

This will build and install both the library and modules.

In order to install LibDAQ in Windows the build system has been switched from Autotools[42] to CMake[43] because the existing one would not work in Windows.

As already highlighted LibDAQ requires the `vcpkg` referenced in TODO to build successfully. In terms of code refactoring the main changes concern the use of the WinSock API rather than BSD sockets which are analyzed in breadth and depth in section 5.4.1.

The DAQ modules that have been ported to Windows are: BPF, Dump, PCAP, Savefile, and Trace. Since as described in section 4.4.5 the *Savefile* module maps a pcap savefile into memory it needs the `<sys/mman.h>` header and associated functions (`mmap`, etc).

It is available in Windows through the `mmap-win32` library[44], available with `vcpkg`, which implements a wrapper for `mmap` functions around the memory mapping Windows API. This required change reflects only in the imported header (See Listing 5.1).

Listing 5.1: Cross-platform code for `<sys/mman.h>` header support.

```
1 #ifdef _WIN32
2   #include <mman/sys/mman.h>
3 #else
```

```

4 | #include <sys/mman.h> // memory management declarations
5 | #endif // _WIN32

```

5.4 Re-writing Linux specific code into Windows specific code

A huge portion of Snort 3 code base (e.g., socket networking and signal management) differs substantially for POSIX and Windows. This is why placing the Windows code along side the existing POSIX code, wrapping with preprocessor conditionals as necessary is not a good alternative and was used only in few situations.

Oppositely, to handle these compatibility issues new abstractions have been created to build separate implementations for each platform.

5.4.1 Porting BSD sockets with Winsock API

While Winsock is often somewhat compatible with Berkeley sockets, it is not strictly compatible, and therefore must be treated differently. Moving from UNIX sockets to Windows sockets is fairly simple. Windows programs require a different set of include files, need initialization and deallocation of WinSock resources, use `closesocket()` instead of `close()`, and use a different error reporting facility. However, the meat of the application is identical to UNIX.

Different header and lib

To use the Winsock API it is needed to include the Winsock 2 header files. The `Winsock2.h` header file contains most of the Winsock functions, structures, and definitions. The `Ws2tcpip.h` header file contains definitions introduced in the WinSock 2 Protocol-Specific Annex document for TCP/IP that includes newer functions and structures used to retrieve IP addresses.[45]

Listing 5.2: UNIX socket headers.

```

1 | #include <sys/types.h>
2 | #include <sys/socket.h>
3 | #include <netinet/in.h>
4 | #include <arpa/inet.h>
5 | #include <netdb.h>
6 | #include <unistd.h>
7 | #include <errno.h>
8 | #include <sys/un.h>

```

Listing 5.3: WinSock headers.

```

9 | #include <WinSock2.h>
10 | #include <Windows.h>
11 | #include <Ws2tcpip.h>
12 | #include <afunix.h>
13 |
14 | #pragma comment(lib, "Ws2_32.lib")

```

Moreover, the build environment must link to the Winsock Library file `Ws2_32.lib`.

The `pragma comment` indicates to the linker that the `Ws2_32.lib` file is needed. Despite the use of the 32 designator, the 64-bit library shares the same name.

Socket Representation

While under BSD sockets, socket descriptors are normal *file descriptors* (small non-negative integers), under WinSock, socket descriptors are (usually) normal NT kernel object *handles*.

This means, that instead of `int`, sockets need to be of type `SOCKET` and the classic comparison with zero to detect error conditions does not hold anymore. While sockets are supposed to be interchangeable with instances of `HANDLE` in many APIs, Microsoft choose to use a different representation so casting is necessary when sockets are used with normal Win32 function calls.

Socket Error Handling

UNIX socket calls report their error status by returning `-1` and setting the value of the variable `errno`. The set of errors they can report are fully enumerated as constants in `<errno.h>` header.

However, under Windows, WinSock is forced to use a different reporting mechanism since `errno` is part of the compiler runtime and not a system API.

As with BSD sockets, `-1` is returned on error but because a WinSock handle is not an integer, one cannot simply check the sign when examining the return values from functions like `socket` or `accept`. Instead, the program is expected to check for equality with `INVALID_SOCKET` or `SOCKET_ERROR`.

Once an error is detected, the application calls `WSAGetLastError()` to retrieve the actual error code.

From the "Handling Winsock Errors" section in the Microsoft Documentation^[46]:

Most Windows Sockets 2 functions do not return the specific cause of an error when the function returns. Some Winsock functions return a value of zero if successful. Otherwise, the value `SOCKET_ERROR` (`-1`) is returned and a specific error number can be retrieved by calling the `WSAGetLastError` function. For Winsock functions that return a handle, a return value of `INVALID_SOCKET` (`0xffff`) indicates an error and a specific error number can be retrieved by calling `WSAGetLastError`. For Winsock functions that return a pointer, a return value of `NULL` indicates an error and a specific error number can be retrieved by calling the `WSAGetLastError` function.

The runtime provided by Visual Studio has the full set of POSIX error numbers provided in its copy of `<errno.h>` but these are disjoint from the codes actually used by WinSock. Instead, the error codes used by WinSock are prefixed with `WSA` (e.g. `WSAEWOULDBLOCK`).

In the following are provided two example to show how Windows Sockets 2 functions that return an integer value (See Listing 5.4) and functions that return a handle (See Listing 5.5). Moreover, it is possible to notice that in both cases the socket descriptor is cast to a `SOCKET` handle.

Listing 5.4: Windows Sockets 2 function to wrap UNIX `send` function

```

1 #define send w32_send
2
3 ssize_t w32_send(int sockfd, const void *buf, size_t len, int flags)
4 {
5     int ret = send((SOCKET)sockfd, (const char *)buf, (int)len, flags);
6     if (ret == SOCKET_ERROR) {
7         wsock2errno();
8         return -1;
9     }
10    return (ssize_t)ret;
11 }

```

Listing 5.5: Windows Sockets 2 function to wrap UNIX `socket` function

```

1 #define socket w32_socket
2
3 int w32_socket(int domain, int type, int protocol)
4 {
5     SOCKET s = socket(domain, type, protocol);
6     if (s == INVALID_SOCKET) {
7         wsock2errno();
8         return -1;
9     }
10    return (int)s;
11 }

```

Unix Sockets

Unix sockets^[47] allow inter-process communication (IPC) between processes on the same machine.

Support for the unix socket has existed both in BSD and Linux for the longest time, but, on Windows it is a quite recent feature the possibility to use the unix socket (`AF_UNIX`) address family on Windows to communicate between Win32 processes.

To use unix sockets in Windows it is firstly needed to replace the `sys/un.h` header with the `afunix.h` one. Then it is possible to write a Windows unix socket winsock application as one would write any other unix socket application, but, using Winsock API's.

This is mainly possible because the `sockaddr_un` structure, used for defining the address of a unix socket, in the Windows implementation of the unix socket has kept the same name, definition and semantics of the unix socket address in Linux, to make cross-platform development easier.

Different `in6_addr` between Win and Linux

The `IN6_ADDR` structure specifies an IPv6 transport address. Unfortunately in Linux and Windows there is a discrepancy between the two structures used to describe the IPv6 address in Linux (`in6.h`) and in Windows (`in6addr.h`).

In Windows we have a union that contains two different representations of the IPv6 transport address, an array (`Byte`) that contains 16 `UCHAR`-typed values and an array that contains eight `USHORT`-typed values (See Listing 5.6).

On the other hand, in Linux there is an additional member (`u6_addr32`) that is not mapped in Windows (See Listing 5.7).

Listing 5.6: Windows IPv6 address structure

```

1 typedef struct in6_addr {
2     union {
3         UCHAR Byte[16];
4         USHORT Word[8];
5     } u;
6 } IN6_ADDR, *PIN6_ADDR, *LPIN6_ADDR;
```

Listing 5.7: Linux IPv6 address structure

```

1 struct in6_addr
2 {
3     union
4     {
5         __u8           u6_addr8[16];
6         __be16        u6_addr16[8];
7         __be32        u6_addr32[4];
8     } in6_u;
9     #define s6_addr           in6_u.u6_addr8
10    #define s6_addr16        in6_u.u6_addr16
11    #define s6_addr32        in6_u.u6_addr32
12 };
```

In order to cope with this mismatch between the two data structures the change proposed in Listing 5.8 needs to be performed. This is already taking into account that both Linux and Windows use a Network Byte Order (big-endian) specification. Linux:

Be aware the `IN6ADDR_` constants and `in6addr_*` externals are defined in network byte order, not in host byte order as are the IPv4 equivalents.*

Windows:

All members of the `IN6_ADDR` structure must be specified in network-byte-order (big-endian).

Listing 5.8: Solution to error: class "in6_addr" has no member "__u6_addr"

```

1 #ifdef _WIN32
2     if ( src_ip && (!IN6_IS_ADDR_V4MAPPED(src_ip) || (src_ip->u.Word[6] &&
3         src_ip->u.Word[7])))
4 #else
5     if (src_ip && (!IN6_IS_ADDR_V4MAPPED(src_ip) || src_ip->s6_addr32[3]))
6 #endif // _WIN32

```

5.4.2 Porting UNIX Signals in Windows

The UNIX operating system supports a wide range of signals (software interrupts that catch or indicate different types of events). Windows on the other hand supports only a small set of signals that is restricted to exception events only. Consequently, converting Snort3 code to Win32 requires the replacement of some UNIX signals.

The Windows signal implementation is limited to the following signals (Table 5.2):

Snort Signals

The Snort signal implementation exploits the following signals (Table 5.3):

To understand why Snort uses these signals and maps them in this way it is important to dig deeper into the logic behind packet analysis.

Snort from a high level is using an *Analyzer thread*. This analyzer is nothing more than a Finite State Machine (See Figure 5.1). It will start in the `NEW` state and will transition to the `INITIALIZED` state once the object is called as part of spinning off a packet thread. Further transitions will be prompted by commands from the main thread.

Signal	Meaning
SIGABRT	Abnormal termination
SIGFPE	Floating-point error
SIGILL	Illegal instruction
SIGINT	CTRL+C signal
SIGSEGV	Illegal storage access
SIGTERM	Termination request

Table 5.2: Signals supported in Windows and their meanings.

Snort Signal	Mapping	Meaning
SIGINT	SIGINT	Shutdown normally
SIGQUIT	SIGQUIT	Shutdown normally
SIGTERM	SIGTERM	shutdown as if started with <code>-dirty-pig</code>
SIGNAL_SNORT_DUMP_STATS	SIGUSR1	Dump stats to stdout
SIGNAL_SNORT_ROTATE_STATS	SIGUSR2	Rotate stats files
SIGNAL_SNORT_RELOAD	SIGHUP	Reload config files
SIGNAL_SNORT_READ_ATTR_TBL	SIGURG	Reload hosts file
SSIGNAL_SNORT_CHILD_READY	SIGCHLD	Confirmation child was started ok

Table 5.3: Signals used by Snort and their meanings.

It will transition from `INITIALIZED` to `STARTED` via the `START` command. Later it will go from `STARTED` to `RUNNING` through the `RUN` command.

Finally, it will end up in the `STOPPED` state when the Analyzer object has finished its execution. This state can be either be prompted by the `STOP` command or if the Analyzer finishes its operation for other reasons (such as encountering an error condition).

The other state the Analyzer can be in is `PAUSED`, which is triggered when it receives the `PAUSE` command while in the running state. To go back from `PAUSED` to `RUNNING` the command `RESUME` is needed.

The other two commands currently available in the source code (but not yet mapped in Windows) are `SWAP` and `ROTATE`. The `ROTATE` command will cause open (per-thread) output files to be closed, rotated, and reopened anew. The `SWAP` command will swap to a new configuration at the earliest convenience.

All in all, the most important signals for a Snort binary able to perform ReadBack operations and Live traffic Capture are the first two. This is because while for the Readback feature in which once we are done reading the pcap file the Analyzer

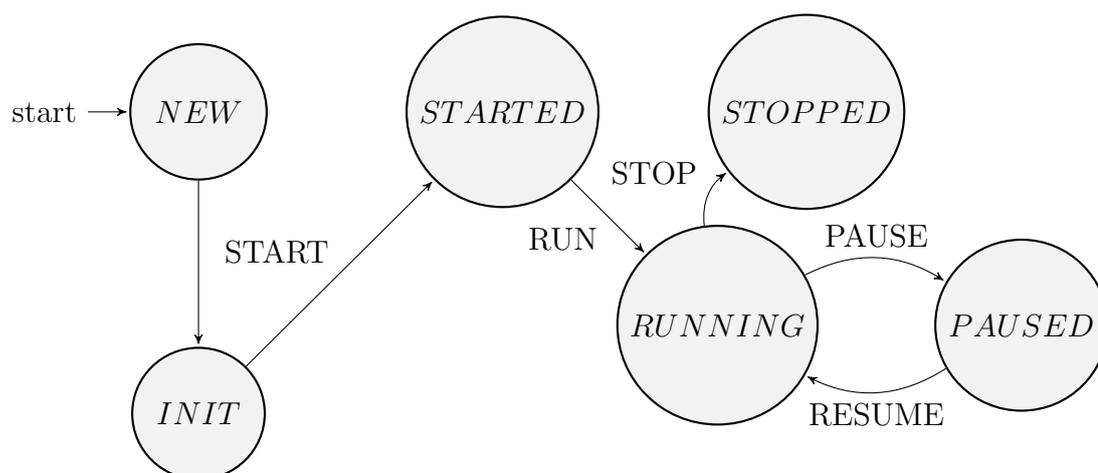


Figure 5.1: Snort Analyzer's Finite State Machine.

will automatically find the EOF and switch from the `RUNNING` to `STOPPED` state in the case of the Interactive mode live interface sniffing we must explicitly stop the execution of Snort and consequently the Analyzer, hitting CTRL+C (which maps to SIGINT).

5.4.3 Data structure alignment

Another important aspect to deal with when porting Snort3 to Windows is the **Data structure alignment**, which is the way data is arranged and accessed in memory. It consists of two different but related issues:

- data alignment: putting the data at a memory offset equal to a multiple of the word size, which increases the system's performance due to the way the CPU handles memory.
- data structure padding: inserting some meaningless bytes between the end of the last data structure and the start of the next to align the data.

Snort native code uses the `__attribute__((packed, aligned(X)))` mechanism to insist particular (X) sized padding, where X should be powers of two. Even though this mechanism is a feature of GCC (GNU C compiler), it was cleverly designed in a way to make it easy to quietly eliminate it if used on platforms other than GNU C (like MSVC).

It can be replaced with another mechanism, `pragma pack`, which is originally a Visual C++ compiler specific extension, which has been implemented in GCC as well for VC++ compatibility.

It is important to highlight that `pragma pack` takes effect from the point of definition and until another `pragma pack`, while `__attribute__((packed))` is effective only for the definition it's attached to.

In order to make this available in Snort we need to:

1. Check the system *Struct Packing*.
2. If the system does not have the `__attribute__` mechanism, elide it.
3. Push the alignment value to stack using `pragma pack(1)`. This will be done for all the structures until we stop it with `pragma pack()`.

Listing 5.9: Cross-Platform Data Structure Alignment.

```

1  #ifndef HAVE_ATTRIB_PACKED
2      #define __attribute__(x)
3  #endif
4
5  #ifdef HAVE_PRAGMA_PACK
6      #pragma pack(1)
7  #endif
8
9  ...
10 // add each packed struct individually tagged with the attribute in between: __attribute__
    ((packed))
11 ...
12
13 #ifdef HAVE_PRAGMA_PACK
14     #pragma pack()
15 #endif
16 ...

```

5.4.4 Thread Local Storage (TLS)

All threads of a process share its virtual address space. The local variables of a function are unique to each thread that runs the function. However, the static and global variables are shared by all threads in the process.

Thread local storage (TLS)[\[48\]](#) is the method through which each thread in a given multithreaded process can allocate locations in which it can store thread-specific data which are accessible to the process using a global index.

It is important to notice that various compiler implementations provide specific ways to declare thread-local variables:

- C++11: The `thread_local` storage class specifier is the recommended way to specify thread-local storage for objects and class members.
- GNU C: The `__thread` storage class marks a static variable as having thread-local storage duration. This means that, in a multi-threaded application, a unique instance of the variable is created for each thread that uses it, and destroyed when the thread terminates.
- MSVC: Provides a Microsoft-specific attribute, `thread`, as extended storage class modifier. A thread variable can be declared with `__declspec(thread)` keyword.

While on the one hand it seems sufficient to declare the `THREAD_LOCAL` macro used by Snort to `__declspec(thread)` instead of the `__thread` keyword, on the other hand in MSVC we have an issue when a data is declared with the `thread` attribute and with the DLL interface.

This issue is very frequent in Snort which makes heavy usage of the combination `SO_PUBLIC - THREAD_LOCAL`. For instance, the following piece of code:

```

1 extern SO_PUBLIC THREAD_LOCAL PacketCount pc;
2 // extern __declspec(dllimport) __declspec(thread) PacketCount pc;

```

Will lead to the C2492 MSVC compiler error, from the Microsoft documentation[49]:

'pc': data with thread storage duration may not have dll interface.

The variable is declared with the `thread` attribute and with the DLL interface. The address of the `thread` variable is not known until run time, so it cannot be linked to a DLL import or export.

This is the reason why for this particular issue in the Snort 3 port to Windows an hybrid (and working) approach has been adopted. For cases in which data is declared with the `thread` attribute and with the DLL interface, the thread locality is dropped. For all other cases in which thread locality is not in conflict with the DLL interface, it is kept.

5.4.5 `SO_PUBLIC` and `SO_PRIVATE` Snort Types

Snort 3 defines two symbols, `SO_PUBLIC` and `SO_PRIVATE` to hide the definition of the `dllexport` and `dllimport` storage-class attributes[50] which are used to export and import functions, data, and objects to or from a DLL. There were two main issues with these Snort types:

1. the symbol `SO_PRIVATE` was not defined if the system was Windows.

2. the symbol `SO_PUBLIC` must be associated with `__declspec(dllexport)` rather than with `__declspec(dllimport)`.

These enhancements are shown in Listing 5.10.

Listing 5.10: Declaration of Snort Types.

```

1 #ifndef SO_PUBLIC
2 #if defined _WIN32 || defined __CYGWIN__
3 #   define SO_PRIVATE    // elides SO_PRIVATE on Windows
4 #   ifdef __GNUC__
5 //#   define SO_PUBLIC __attribute__((dllimport))
6 #   define SO_PUBLIC __attribute__((dllexport))
7 #   else
8 //#   define SO_PUBLIC __declspec(dllimport)
9 #   define SO_PUBLIC __declspec(dllexport)
10 #   endif
11 #   define DLL_LOCAL
12 #else
13     ...
14 #endif
15 #endif

```

5.4.6 Miscellaneous MSVC Fixes

Find First Set vs Leading Zeros Count

In Linux the `ffs()` function[51] returns the position of the first (least significant) bit set in a word. On the contrary Windows counts the number of leading zeros in a 16-, 32-, or 64-bit integer through the `__lzcnt16`, `__lzcnt`, `__lzcnt64` compiler intrinsic functions (respectively)[52].

We simply need to associate these functions as shown in Listing 5.11.

Listing 5.11: Mapping Linux Find First Set to Windows.

```

1 #define ffs __lzcnt

```

MSVC does not support logical operator keywords

The Microsoft Visual C++ compiler does not support the keywords like `and`, `not`, `or`, etc. They must be replaced by the more commonly used operators like `&&` instead of `and`, `||` instead of `or` as reported in table 5.4.

The `min/max` macros clash between C++ and Windows

The Windows header file `WinDef.h`[53] - brought inside by `Windows.h` - defines two macros `min` and `max` which result in conflicts and compiler errors.

Keyword	Logical Operator
and	&&
and_eq	&=
bitand	&
bitor	
compl	~
not	!
not_eq	!=
or	
or_eq	=
xor	^
xor_eq	^=

Table 5.4: MSVC unsupported keywords and their logical operators counterparts.

Any C++ source code including this Windows header will likely head into problems if `min` or `max` are used together with the Standard C++ library functions `std::min()`/`std::max()` as defined in the `<algorithm>` header.

The solutions to this can be multiple:

- Define the `NOMINMAX` macro to instruct `WinDef.h` to avoid the definition of the `min/max` macros. For example updating the Visual C++ compiler options with `/D NOMINMAX` or inserting `define NOMINMAX`.
- Redefine `min/max` in the abstraction layer. (See Listing 5.12).

Listing 5.12: Redefinition of the `min/max` macros.

```

1 #ifndef MIN
2 #define MIN(a, b)  (((a) < (b)) ? (a) : (b))
3 #endif
4 #ifndef MAX
5 #define MAX(a,b)  (((a) > (b)) ? (a) : (b))
6 #endif

```

Reversing the Order of Bytes

The Byteswap functions from both Linux[54] and Windows[55] allow to *reverse order of bytes* in an integer (E.g.: `0x123456789abcdef` -> `0xefcdab8967452301`).

These are essential routines to move from big endian processors (bytes are numbered from most-significant to least-significant in a multi-byte word) to little endian ones (bytes are numbered from least-significant to most-significant), and viceversa.

Listing 5.13: Mapping Linux `bswap_*` to Windows.

```
1 #ifdef _WIN32
2 #include <stdlib.h>
3 #define bswap_16(x) _byteswap_ushort(x)
4 #define bswap_32(x) _byteswap_ulong(x)
5 #define bswap_64(x) _byteswap_uint64(x)
6 #endif // _WIN32
```

Compare two strings ignoring case

While on Linux this job is performed by the `strcasecmp` and the `strncasecmp` (`<strings.h>`)[\[56\]](#) on Windows it is achieved through the `lstrncmpi` (`<Winbase.h>`)[\[57\]](#) and the `_strnicmp` (`<string.h>`)[\[58\]](#) functions.

Listing 5.14: Mapping Linux string comparison functions to Windows.

```
1 #ifdef _WIN32
2 #define strcasecmp lstrncmpi
3 #define strncasecmp _strnicmp
4 #endif // _WIN32
```


Chapter 6

Automating Snort 3 Installation

After having achieved a successful Snort 3 binary distribution for Windows the main aim of this section is to simplify and automate the installation of Snort 3 on Windows systems through the portability feature provided by Docker.

The focus of this chapter revolves around 4 main points:

- The choice of the Windows container base image.
- The installation of Chocolatey and Vcpkg to install the necessary LibDAQ and Snort 3 dependencies.
- The installation of Visual Studio Build Tools into a Windows container to build LibDAQ and Snort 3.
- The installation of Npcap and the associated drivers to support network packet capture.

6.1 Windows Container Base Images

Windows offers four container base images (see Figure 6.1) that users can build from. Each base image is a different type of the Windows or Windows Server operating system, has a different on-disk footprint, and has a different set of the Windows API set [59].

After careful consideration the chosen base image is *Windows Server Core*[60] with .NET Framework 4.8 (See Listing 6.1).

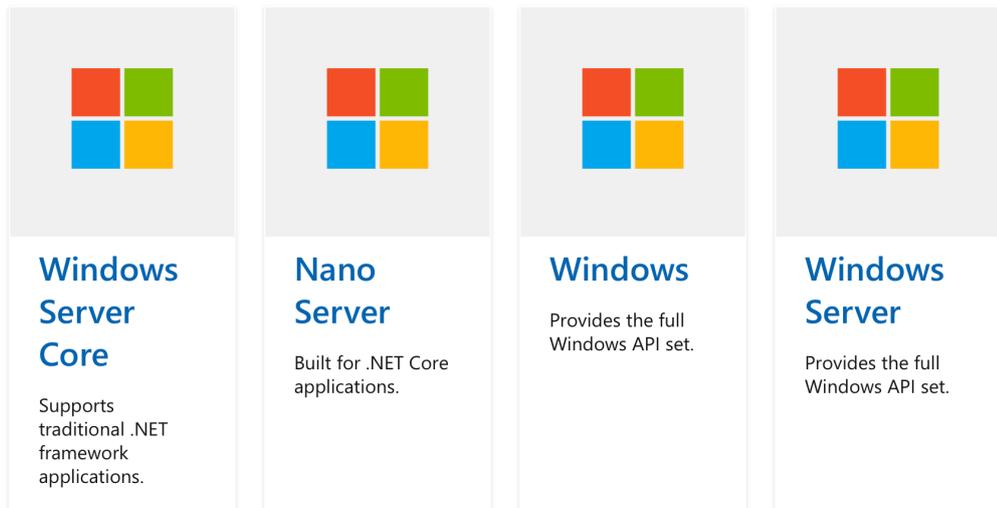


Figure 6.1: Windows container base images.

Listing 6.1: Windows Base Image Identification.

```

1 # escape='
2
3 ARG FROM_IMAGE=mcr.microsoft.com/dotnet/framework/sdk:4.8-windowsservercore-
  ltsc2019
4 FROM ${FROM_IMAGE}

```

6.2 Installing Snort Dependencies in a Windows Container

As discussed in section 5.2 the dependencies of Snort 3 are mainly installed using Chocolatey and Vcpkg package managers.

Listing 6.2: Installing Chocolatey and Vcpkg in a Windows Container.

```

1 SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop'; $ProgressPreference =
  'SilentlyContinue;"]
2
3 # Install Chocolatey
4 RUN [Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12; '
  iex(iwr -useb https://chocolatey.org/install.ps1);
5
6
7 # Download and install vcpkg
8 RUN mkdir C:\dev && cd C:\dev && '
  git clone https://github.com/Microsoft/vcpkg.git && '
  .\vcpkg\bootstrap-vcpkg.bat -disableMetrics
9
10

```

6.3 Installing Visual Studio Build Tools in a Windows Container

The installation of Visual Studio Build Tools into a Windows container to build LibDAQ and Snort 3 is bound to the installation, through Chocolatey, of:

- The Visual Studio 2019 Build Tools 16.11.8.0 that allow to build native and managed MSBuild-based applications without requiring the Visual Studio IDE. [61]
- The Visual C++ build tools workload for VS2019 Build Tools 1.0.1. [62]
- The Desktop development with C++ workload for VS2019. [63]

Listing 6.3: Installing Visual Studio Build Tools in a Windows Container.

```

1 SHELL ["cmd", "/S", "/C"]
2
3 # Visual Studio 2019 Build Tools 16.11.8.0
4 RUN cinst -y --no-progress visualstudio2019buildtools
5 # Visual C++ build tools workload for VS2019 Build Tools 1.0.1
6 RUN cinst -y visualstudio2019-workload-vctools
7 # Install Desktop development with C++ workload for VS2019
8 RUN cinst -y --no-progress visualstudio2019-workload-nativedesktop || echo "ignore
  reboot request"

```

6.4 Installing Npcap in a Windows Container

The installation steps for Npcap in a Windows container are slightly different from the ones discussed in section 5.2.1 because currently there is no way to install drivers in Windows containers. However containers on Windows can use drivers that are present on host OS.

For this reason in the Windows container is sufficient to install the Npcap SDK (against which Snort 3 will be built) and to copy the Npcap DLLs from the host to the container (See Listing 6.4).

Listing 6.4: Copying Npcap DLLs to the Windows Container.

```

1 COPY wpcap.dll C:\\Windows\\System32\\wpcap.dll
2 COPY Packet.dll C:\\Windows\\System32\\Packet.dll

```


Chapter 7

Conclusion and Future Work

The main purpose of this thesis work is to create a Proof-of-Concept Malware Detection Engine for Cisco Secure Endpoint (SE), formerly known as Advanced Malware Protection (AMP).

To achieve this objective an IDS/IPS as Snort 3 has been taken into consideration and, together with its required dependencies, has been ported to Windows to provide a future engine for the Windows Secure Endpoint.

Being Snort 3 heavily dependent on Linux's Libpcap library, the available Windows PCAP modules have extensively been analyzed. From this research has emerged that even though WinPcap used to be recognized as the industry-standard tool for link-layer network access in Windows environments nowadays it has been supplanted by Npcap.

Although Npcap features many issues like the lack of a silent installer, the free license only allows five installs (does not allow for any redistribution) and the poor driver performances it is a great choice for a Proof-of-Concept given also the fact that the Libdnet-Striped depends on Npcap.

Using Npcap which is the natural Windows port of Libpcap there has been the possibility to work with LibDAQ (the Data AcQuisition library, for packet I/O). The first change that was needed was to change the build system from Autotools to CMake in order to build on Windows. Out of the currently 10 existing modules a successful Windows port of 5 modules has been achieved.

Among the Five modules available in Windows there is the possibility to use the PCAP one which allows both the *Readback* and *Live* DAQ Modes which were the two main objectives of this project.

This means that Snort 3 for Windows is not only capable of reading packet capture files but also of opening a network traffic interface and sniff packets and display them in a continuous stream on the console. It must be highlighted, though, that the Live interface sniffing features are achieved only using a native Windows port through Microsoft Visual C++ compiler and not using MSYS2 nor MinGW-w64.

Among the different options available to port Snort 3 to Windows the most effective was found to be Microsoft Visual C++ compiler (MSVC). Not only MSVC is the State of the art compiler in terms of native Windows ports but also the lack of live interface sniffing capabilities for MinGW-w64 and MSYS2 libpcap's was an incentive to build Snort 3 using MSVC.

This choice was supported by the fact that both the Npcap SDK and the Libdnet-striped were compiled using MSVC and thus only compatible for a native build using the same compiler (MinGW-w64/MSYS2 are incompatible with MSVC libraries).

To cope with the main incompatibilities present when running a Linux executable in Windows an abstraction layer has been implemented in order to avoid a huge amount of changes in the original codebase and only the changes which were absolutely specific were threatened using some preprocessor definitions. Those features which are completely extraneous to a Windows program (for example, `syslog.h`, `pwd.h`, `grp.h`) were safely elided.

Given the difficulties in installing all the dependencies of Snort 3 in Windows (but also in Linux), the installation of Snort 3 in Windows has been made even easier with a containerized application that can be deployed to any other system where Docker is running to be sure that Snort 3 will perform exactly as it should on Windows. This work has lead to many discoveries in terms of Windows containers that can be exploited in the future for similar situations.

7.1 Future Work

Since this thesis work was a Proof-of-Concept there are some aspects that will need to be handled in future.

The main one is to investigate the possibility to implement a LibDAQ module based on WinDivert to overcome the limitations shown by Npcap. In this regard, it is good to notice once more the strength of Snort 3 which relying on this abstraction layer for data acquisition will remain untouched by these changes.

Bibliography

- [1] *What is Snort?* <https://www.snort.org>. [Online]. 2021 (cit. on p. 5).
- [2] *Snort FAQ - What are the differences in the rule sets?* <https://www.snort.org/faq/what-are-the-differences-in-the-rule-sets>. [Online] (cit. on p. 8).
- [3] Ian Firms. *Barnyard2*. <https://github.com/firnsy/barnyard2>. [Online]. 2016 (cit. on p. 10).
- [4] Google. *FlatBuffer*. <https://google.github.io/flatbuffers/>. [Online]. 2021 (cit. on p. 10).
- [5] The Zeek Project. *Zeek*. <https://zeek.org>. [Online]. 2021 (cit. on p. 12).
- [6] *Zeek on Windows, Issue 951*. <https://github.com/zeek/zeek/issues/951>. [Online]. 2021 (cit. on p. 13).
- [7] The Open Information Security Foundation. *Suricata*. Version 6.0.3. 2021. URL: <https://suricata.io> (cit. on p. 13).
- [8] Microsoft. *Microsoft Defender for Identity*. <https://www.microsoft.com/en-us/security/business/threat-protection/identity-defender>. [Online]. Sept. 2021 (cit. on p. 13).
- [9] David Curwin, Kent Sharkey, and M. Baldwin. *What is Microsoft Defender for Identity?* <https://docs.microsoft.com/en-us/defender-for-identity/what-is>. [Online]. 2021 (cit. on p. 14).
- [10] David Curwin and Kent Sharkey. *Microsoft Defender for Identity FAQ: WinPcap and Npcap drivers*. <https://docs.microsoft.com/en-us/defender-for-identity/technical-faq#winpcap-and-npcap-drivers>. [Online]. Oct. 2021 (cit. on p. 14).
- [11] Cisco and/or its affiliates. *ClamAV Introduction*. <https://docs.clamav.net/Introduction.html#clamav>. [Online]. 2021 (cit. on p. 14).
- [12] The Tcpdump Group. *LibPCAP*. Version 1.10.1. 2021. URL: <https://www.tcpdump.org> (cit. on p. 17).

- [13] Riverbed Technology. *WinPcap*. Version 4.1.3. 2013. URL: <https://www.winpcap.org> (cit. on p. 17).
- [14] Nmap Software LLC. *Npcap*. Version 1.55. 2021. URL: <https://nmap.org/npcap/> (cit. on p. 17).
- [15] MSYS2. *Libpcap*. Version 1.10.0-1. 2021. URL: https://packages.msys2.org/package/libpcap?repo=msys&variant=x86_64 (cit. on p. 21).
- [16] MSYS2. *MinGW-w64-x86_64-libpcap*. Version 1.10.0-2. 2021. URL: https://packages.msys2.org/package/mingw-w64-x86_64-libpcap?repo=mingw64 (cit. on p. 21).
- [17] The Tcpdump Group. *Man page of PCAP_CREATE - RETURN VALUE*. https://www.tcpdump.org/manpages/pcap_create.3pcap.html. [Online]. Nov. 2021 (cit. on p. 22).
- [18] Riverbed Technology. *WinPcap v4.1.3*. <https://www.winpcap.org/news.htm>. [Online]. Mar. 2013 (cit. on p. 24).
- [19] Riverbed Technology. *WinPcap End of Life*. <https://www.winpcap.org/default.htm>. [Online]. Sept. 2018 (cit. on p. 24).
- [20] Daiyuu Nobori, University of Tsukuba, Japan. *Win10Pcap*. 2015. URL: <https://www.win10pcap.org> (cit. on p. 25).
- [21] Nmap Software LLC. *Nmap*. Version 7.92. 2021. URL: <https://nmap.org> (cit. on p. 25).
- [22] ReQrypt.org. *WinDivert*. Version 2.20. 2019. URL: <https://reqrypt.org/windivert.html> (cit. on p. 26).
- [23] Basil. *ReQrypt.org*. <https://reqrypt.org/home.html>. [Online]. 2017 (cit. on p. 26).
- [24] *Setting up Suricata IPS/inline for Windows*. <https://suricata.readthedocs.io/en/latest/setting-up-ipsinline-for-windows.html#setting-up-ips-inline-for-windows>. [Online]. 2021 (cit. on p. 30).
- [25] Cisco Systems, Inc. *LibDAQ*. Version 3.0.5. 2021. URL: <https://github.com/snort3/libdaq> (cit. on p. 31).
- [26] Luigi Rizzo. *Netmap - the fast packet I/O framework*. Version 11.3. URL: <http://info.iet.unipi.it/~luigi/netmap/> (cit. on p. 34).
- [27] *Netmap (Windows)*. <https://github.com/luigirizzo/netmap#windows>. [Online]. 2015 (cit. on p. 34).
- [28] Colin Robertson, Kent Sharkey, Nick Schonning, Jak Koke, Mike Jones, Mike Blome, Gordon Hogenson, and Saisang Cai. *Running Linux programs on Windows*. <https://docs.microsoft.com/en-us/cpp/porting/porting-from-unix-to-win32?view=msvc-170>. [Online]. Mar. 2021 (cit. on p. 37).

-
- [29] Craig Loewen et al. *What is the Windows Subsystem for Linux?* <https://docs.microsoft.com/en-us/windows/wsl/about>. [Online]. Sept. 2021 (cit. on p. 37).
- [30] *Virtual Machines*. <https://azure.microsoft.com/en-gb/services/virtual-machines/>. [Online]. 2021 (cit. on p. 37).
- [31] *Docker on Azure*. <https://docs.microsoft.com/en-us/azure/docker/>. [Online]. 2021 (cit. on p. 37).
- [32] *Cygwin*. <https://cygwin.com>. [Online]. 2021 (cit. on p. 37).
- [33] *Microsoft C++ (MSVC)*. <https://docs.microsoft.com/en-us/cpp/build/reference/compiling-a-c-cpp-program?view=msvc-170>. [Online]. Mar. 2021 (cit. on p. 37).
- [34] *Introduction to Hyper-V on Windows 10*. <https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/>. [Online]. 2019 (cit. on p. 38).
- [35] Craig Loewen, v-chmcl, Matt Wojciakowski, and v-surgos. *Will WSL 2 be able to use networking applications?* <https://docs.microsoft.com/en-gb/windows/wsl/faq#will-wsl-2-be-able-to-use-networking-applications->. [Online]. Dec. 2021 (cit. on p. 38).
- [36] Chocolatey Software, Inc. *Chocolatey*. Version 0.11.3. 2021. URL: <https://chocolatey.org> (cit. on p. 39).
- [37] Microsoft. *Vcpkg*. Version 0.11.3. May 12, 2021. URL: <https://vcpkg.io/en/index.html> (cit. on p. 39).
- [38] Nmap Software LLC. *Downloading and Installing Npcap Free Edition*. <https://nmap.org/npcap/>. [Online]. 2021 (cit. on p. 39).
- [39] Colin Robertson. *.vcxproj and .props file structure*. <https://docs.microsoft.com/en-us/cpp/build/reference/vcxproj-file-structure?view=msvc-170>. [Online]. Aug. 2021 (cit. on p. 40).
- [40] Gordon Hogenson, Dave Thomas, Theano Petersen, Terry G. Lee, Genevieve Warren, Mike Jacobs, Nick Schonning, Mike Jones, and Maira Wenzel. *MS-Build*. <https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild?view=vs-2022>. [Online]. Oct. 2021 (cit. on p. 40).
- [41] Inc. Cisco Systems. *LibDAQ README.md*. <https://github.com/snort3/libdaq/blob/master/README.md>. [Online]. 2021 (cit. on p. 41).
- [42] GNU.org. *Autotools*. https://www.gnu.org/software/automake/manual/html_node/Autotools-Introduction.html. [Online] (cit. on p. 41).
- [43] Kitware. *CMake*. Version 3.22.0. 2021. URL: <https://cmake.org> (cit. on p. 41).

- [44] Steven Lee. *mman-win32*. <https://github.com/alitrack/mman-win32>. [Online]. Oct. 2019 (cit. on p. 41).
- [45] Steven White, Kent Sharkey, Adam Clouthier, and Michael Satran. *Creating a Basic Winsock Application*. <https://docs.microsoft.com/en-us/windows/win32/winsock/creating-a-basic-socks-application>. [Online]. July 2021 (cit. on p. 42).
- [46] Steven White, Kent Sharkey, and Michael Satran. *Handling Winsock Errors*. <https://docs.microsoft.com/en-us/windows/win32/winsock/handling-socks-errors>. [Online]. July 2021 (cit. on p. 43).
- [47] *Unix - sockets for local interprocess communication*. <https://man7.org/linux/man-pages/man7/unix.7.html>. [Online]. Mar. 2021 (cit. on p. 44).
- [48] Tyler Whitney, Kent Sharkey, Colin Robertson, Xinye Tao, Mike Jones, Mike Blome, Gordon Hogenson, and Saisang Cai. *Thread Local Storage (TLS)*. <https://docs.microsoft.com/en-us/cpp/parallel/thread-local-storage-tls?view=msvc-170>. [Online]. Aug. 2021 (cit. on p. 49).
- [49] Colin Robertson, Kent Sharkey, Mike Jones, Mike Blome, Gordon Hogenson, and Saisang Cai. *Compiler Error C2492*. <https://docs.microsoft.com/en-us/cpp/error-messages/compiler-errors-1/compiler-error-c2492?view=msvc-160>. [Online]. Mar. 2021 (cit. on p. 50).
- [50] Colin Robertson, Kent Sharkey, Nick Schonning, Mike Jones, Mike Blome, Gordon Hogenson, and Saisang Cai. *dllexport, dllimport*. <https://docs.microsoft.com/en-us/cpp/cpp/dllexport-dllimport?view=msvc-170>. [Online]. Mar. 2021 (cit. on p. 50).
- [51] *ffs(3) — Linux manual page*. <https://man7.org/linux/man-pages/man3/ffs.3.html>. [Online]. 2021 (cit. on p. 51).
- [52] *__lzcnt16, __lzcnt, __lzcnt64*. <https://docs.microsoft.com/en-us/cpp/intrinsics/lzcnt16-lzcnt-lzcnt64?view=msvc-160>. [Online]. 2021 (cit. on p. 51).
- [53] Microsoft. *windef.h header*. <https://docs.microsoft.com/en-us/windows/win32/api/windef/>. [Online]. May 2021 (cit. on p. 51).
- [54] *bswap(3) — Linux manual page*. https://man7.org/linux/man-pages/man3/bswap_16.3.html. [Online]. June 2021 (cit. on p. 52).
- [55] Tyler Whitney, Kent Sharkey, Maira Wenzel, Mike Blome, Mike Jones, Gordon Hogenson, Saisang Cai, and Colin Robertson. *__byteswap_uint64, __byteswap_ulong, __byteswap_ushort*. <https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/byteswap-uint64-byteswap-ulong-byteswap-ushort?view=msvc-170>. [Online]. Mar. 2021 (cit. on p. 52).

- [56] *strcasecmp(3)* — *Linux manual page*. <https://man7.org/linux/man-pages/man3/strcasecmp.3.html>. [Online]. 2021 (cit. on p. 53).
- [57] *lstrcmpiA function (winbase.h)*. <https://docs.microsoft.com/en-us/windows/win32/api/winbase/nf-winbase-lstrcmpia>. [Online]. 2021 (cit. on p. 53).
- [58] *_strnicmp function (string.h)*. <https://docs.microsoft.com/en-us/cpp/c-runtime-library/reference/strnicmp-wcsnicmp-mbsnicmp-strnicmp-l-wcsnicmp-l-mbsnicmp-l?view=msvc-170>. [Online]. 2021 (cit. on p. 53).
- [59] Susan Boher, Heidi Lohr, Elizabeth Ross, Vidush Vishwanath, Krzysztof Krak, Craig Wilhite, Maira Wenzel, and Patrick Lang. *Container Base Images*. <https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/container-base-images>. [Online]. Nov. 2021 (cit. on p. 55).
- [60] DockerHub. *Windows Server Core by Microsoft*. https://hub.docker.com/_/microsoft-windows-servercore (cit. on p. 55).
- [61] Chocolatey Software, Inc. *Visual Studio 2019 Build Tools 16.11.8.0*. 2021. URL: <https://community.chocolatey.org/packages/visualstudio2019buildtools> (cit. on p. 57).
- [62] Chocolatey Software, Inc. *Visual C++ build tools workload for Visual Studio 2019 Build Tools*. 2021. URL: <https://community.chocolatey.org/packages/visualstudio2019-workload-vctools> (cit. on p. 57).
- [63] Chocolatey Software, Inc. *Desktop development with C++ workload for Visual Studio 2019*. 2021. URL: <https://community.chocolatey.org/packages/visualstudio2019-workload-nativedesktop> (cit. on p. 57).