「又是一本讓人讚嘆的 Python 好書。只需對書中的許多程式稍作調整或修飾就能使用，程式的保質期限至少 10 年，這對於講述安全的書籍來說是很罕見的。」

—Stephen Northcutt,
SANS Technology Institute 創始會長

「一本專述 Python 在 Offensive Security 資安攻防的好書。」

—Andrew Case,
Volatility 核心開發者和《The Art of Memory Forensics》的合著者

「如果您真的有駭客的靈魂，只需要一點火花就能走出自己的路，並成就一些神奇的事情。Justin Seitz 的這本書提供了很多火花。」

—Ethical Hacker

「無論您是想要成為一名專業的駭客/滲透測試專家，還是只是想知道其工作的原理，這本書都是您必需閱讀的。內容尖銳激烈、技術上合理，也讓人大開眼界。」

—Sandra Henry-Stocker, IT World

「絕對是資安和技術安全專業人士的推薦讀物，只要有一點 Python 基礎就能活用。」

—Richard Austin, IEEE Cipher

前言

我為暢銷的第一版《黑帽 Python：給駭客與滲透測試者的 Python 開發指南》寫前言以來，已經過了六年。在這段時間裡世界發生了很多變化，但有一件事沒變：我仍然編寫了大量的 Python 程式。在電腦安全這個領域中，您仍然會依照任務的需要使用以各種語言所編寫的工具。您會看到為了處理 kernel 漏洞所編寫的 C 程式碼、為 JavaScript 模糊測試編寫的 JavaScript 程式碼，或者用 Rust 等較新的「時髦」語言編寫的 proxy 代理伺服器，但 Python 仍是這個行業的主力。在我看來，Python 仍然是最容易上手的語言，而且有大量好用的程式庫，能快速編寫出程式碼以簡單方式執行複雜的任務，Python 一直是最好的選擇。大多數電腦安全工具和漏洞利用（exploit）仍然是用 Python 編寫的。從 CANVAS 等漏洞利用框架到 Sulley 等經典模糊測試的所有內容大都是用 Python 語言所編寫。

在《黑帽 Python：給駭客與滲透測試者的 Python 開發指南》第一版出版之前，我已經用 Python 編寫了許多模糊測試（fuzzer）和漏洞利用（exploit）程式。其中包括針對 Mac OS X 的 Safari、iPhone 和 Android 手機，甚至 Second Life 的漏洞利用和攻擊（您可能需要用 Google 搜尋 Second Life 漏洞攻擊）。

不管怎樣，從那以後，我在 Chris Valasek 的幫助下編寫了一個非常特別的漏洞利用程式，能夠遠端入侵 2014 年的 Jeep Cherokee 和其他汽車。當然，這個漏洞利用程式是用 Python 編寫的，使用了 dbus-python 模組。我們編寫的所有工具（能讓我們遠端控制受感染車輛的轉向、剎車和加速）也是用 Python 編寫

的。在某種程度上來看，您可以說因為 Python 而召回了 140 萬輛 Fiat Chrysler 的汽車。

如果您對修補資訊安全的工作感興趣，Python 是很好的學習語言，因為有大量的反向工程和開發程式庫可供您取用。現在只要等 Metasploit 開發人員意識到要從 Ruby 切換到 Python，我們的社群就會更團結一致了。

在這本深受大家喜愛的經典書籍新版本中，Justin 和 Tim 已將所有程式碼更新到 Python 3 版。就我個人而言，我是個盡可能使用 Python 2 的恐龍，但隨著好用的程式庫完全從 Python 2 遷移到 Python 3 時，即使是我這種恐龍也能很快學會怎麼使用。這本新版書籍涵蓋了積極年輕駭客入門所需的大量主題，從如何讀寫網路資料封包的基礎知識到 Web 應用程式稽核與攻擊所需的相關內容。

總而言之，第二版的《黑帽 Python：給駭客與滲透測試者的 Python 開發指南》是本有趣的讀物，是由經驗豐富的專家編寫，他們願意分享在此過程中所學到的秘密。雖然本書不會立即讓您變成像我一樣的超級駭客，但一定會引領您走上正確的道路。

請記住，腳本小子（script kiddie）和專業駭客（professional hackers）之間的區別在於前者只會使用別人的工具。

而後者可以自己設計編寫。

Charlie Miller
Security Researcher
St. Louis, Missouri
October 2020

序

Python 駭客（hacker）、Python 程式設計師（programmer），這二種角色都能用來描述我們。Justin 花費了很多時間進行滲透測試，這需要具備快速開發 Python 工具的能力，重點是能交付結果（寫出來的程式不一定是好看、最佳化，甚至很穩定）。Tim 的口頭禪是「讓程式能發揮作用、容易理解且快速－程式是依照這樣的要求順序來開發的」。如果您的程式碼具有可讀性，您的分享對象就能理解它，幾個月後自己再查看時也很容易理解。在本書中，您會了解到這就是我們編寫程式碼的方式：駭入破解（hacking）是我們的最終目標，而乾淨、易懂的程式碼是我們用來實現目標的方式。我們希望這種理念和風格也能幫助到您。

自從本書第一版問世以來，Python 世界發生了很多變化。Python 2 版本已於 2020 年 1 月結束它的使命了。Python 3 版變成程式編寫和開發平台推薦的版本。因此，本書第二版重構了書中的程式碼，並使用最新的套件和程式庫將其移植到 Python 3 版。這裡使用了 Python 3.6 版和更高版本所提供的語法，例如 Unicode 字串、Context managers 和 f-strings 等。最後，我們在第二版書中新增了編寫程式碼和網路概念的解釋說明，例如 Context managers 的使用、Berkeley Packet Filter 語法以及 ctypes 和 struct 程式庫的比較。

隨著您閱讀本書的進展，您會意識到我們不會深入探討某個主題，這是故意的。我們希望提供您基礎的知識，給您一些快速體驗，以便您獲得駭客工具開發領域的原理知識。考慮到這一點，我們在整本書中放置了很多解釋說明、概

念想法和習題作業，幫助您朝著目標前進。我們鼓勵您深入探索這些概念想法，我們很樂意聽到您自己有能力完成任何一項工具。

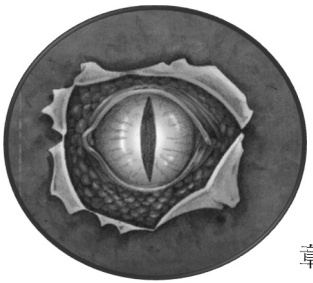
與其他技術書籍一樣，不同程度水準的讀者會對本書有不同的體驗。某些讀者可能直接翻到與他最近工作相關的章節主題，而某些讀者可能會從頭到尾閱讀。如果您是初階到中階的 Python 程式設計師，我們建議您從本書的開頭按順序閱讀各個章節。閱讀與學習的路上您會發掘到一些好的積木讓您在未來可以搭建完整的應用。

首先，我們在第 2 章中介紹了網路基礎知識，然後在第 3 章中慢慢探究原始通訊端（raw socket），並在第 4 章中使用 Scapy 來製作一些更有趣的網路工具。本書的後半部分會從侵入 Web 應用程式談起，第 5 章是介紹自訂工具，然後第 6 章擴充很多人用的 Burp Suite。隨後會花很多時間討論木馬，第 7 章是使用 GitHub 來執行命令和控制，一直到第 10 章會介紹一些提升 Windows 許可權限的技巧。最後一章是介紹關於 Volatility 記憶體鑑識分析的程式庫，這套工具能幫助您了解防守方的思維，並展示如何利用這套工具進行攻擊。

我們盡量讓程式碼的範例簡短而切題，其解釋說明也是如此。如果您對 Python 還不熟悉，建議您自己手動輸入每一行程式指令，這樣更能熟悉和記憶編寫程式碼所有流程。本書的所有程式碼範例都能在 https://nostarch.com/black-hat-python2E 上取得。

讓我們一起邁向學習的旅程吧！

第 3 章
製作 sniffer



網路封包分析器（sniffer，或譯封包監聽器）允許您查看進入和離開某台目標機器的封包內容。因此，這種工具在漏洞入侵的前後都很實用。在某些情況下，您可以使用現成的 sniffer 工具，如 Wireshark（https://wireshark.org/），或是使用 Pythonic 風格的解決方案，如 Scapy（會在下一章探討）。儘管如此，知道如何開發自己的速成封包分析程式以觀察和解碼網路流量是很有用處的。

編寫開發這種程式也會讓您深刻感謝現有成熟的工具，您不需要付出什麼努力，因為它們能輕鬆地處理所有細節的問題。您還能學習一些新的 Python 技術，同時對網路低階的底層工作原理有更深入的理解。

前一章我們介紹了如何使用 TCP 和 UDP 來發送和接收資料，這種方式可能也是您與大多數網路服務互動的方式。但是在這些高階層的協定底下其實就是網路封包發送和接收的基礎建制區塊。您會使用 raw socket 存取較低階層的網路資訊，例如 raw Internet Protocol（IP）和 Internet Control Message Protocol（ICMP）的標頭。我們在本章不會解碼任何 Ethernet 資訊，但如果您打算執行

低階的攻擊，例如 ARP 毒化，或是開發無線網路評估工具，您就要非常熟悉 Ethernet frames 及其使用方法。

讓我們先從如何發現網段上還在活動中的主機開始。

建制 UDP 主機探索工具

我們的 sniffer 程式的主要目標是探索發覺目標網路上的主機。攻擊者希望能夠看到網路上的所有潛在目標，以便能專注於偵察和嘗試入侵漏洞。

我們會使用大多數作業系統已知的操作來確定特定 IP 位址上是否有還在活動的主機。當我們把 UDP 封包發送到主機上的關閉埠號時，該主機通常會回發一條 ICMP 訊息，指示該埠號無法抵達。這個 ICMP 訊息告知有一台活著的主機，如果沒有主機，我們不會收到對 UDP 封包的任何回應。因此，我們必須挑選一個不太使用的 UDP 埠號，為了獲得最大的覆蓋範圍，我們可以探測多個埠號，確保不會存取到已啟用的 UDP 服務。

為什麼是 UDP 呢？嗯，在整個子網路中散布訊息並等待 ICMP 回應抵達，並不會造成什麼負擔。這是個非常簡單的掃描程式，因為大部分的處理工作都用在解碼和分析各種網路協定標頭（header）。我們會在 Windows 和 Linux 實作這支主機掃描程式，好讓我們能在企業環境中更有可能使用到它。

我們還可以在掃描程式中加入額外的處理邏輯，方便我們在探索到的主機上啟動完整的 Nmap 埠號掃描。如此一來，就能判斷是否具有潛在的網路攻擊漏洞。這裡留給讀者的自己練習，我們很期待能聽到各種擴充這個核心概念的創意。讓我們開始吧！

在 Windows 和 Linux 監聽偵測封包

在 Windows 中存取 raw socket 的過程與在 Linux 中略有不同，但我們希望能夠更有彈性地把同一支 sniffer 程式部署到多種平台。為了解決這個問題，我們會建立一個 socket 物件，然後判斷是在哪個平台上執行。Windows 需要透過 socket 輸入/輸出控制（IOCTL）設定一些額外的旗標（flags），從而在網路介

面上啟用混雜模式（promiscuous mode）。輸入/輸出控制（IOCTL）是 user space 程式與 kernel 模式元件進行溝通的方法。詳細內容請參考：http://en.wikipedia.org/wiki/Ioctl。

在本章的第一個範例中，我們會簡單編寫出 raw socket 的 sniffer 程式，能讀取單個封包，然後退出：

```
import socket
import os

# 要監聽的主機
HOST = '192.168.1.203'

def main():
    # 建立 raw socket、bin 到公共界面
    if os.name == 'nt':
        socket_protocol = socket.IPPROTO_IP
    else:
        socket_protocol = socket.IPPROTO_ICMP

 ❶ sniffer = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket_protocol)
    sniffer.bind((HOST, 0))
    # 在捕捉時引入 IP 標頭
 ❷ sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

 ❸ if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    # 讀入一個封包
 ❹ print(sniffer.recvfrom(65565))

    # 如果是在 Windows，則關掉混雜模式
 ❺ if os.name == 'nt':
        sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

if __name__ == '__main__':
    main()
```

我們一開始會把 Host IP 定義為自己機器的位址，並使用在網路介面上監聽偵測封包所需的參數來建構 socket 物件❶。Windows 和 Linux 的區別在於，Windows 不受協定限制允許監聽偵測所有傳入的封包，而 Linux 則強制指定監聽偵測 ICMP 封包。請留意，我們使用的是混雜模式（promiscuous mode），這需要 Windows 的 administrator 許可權或 Linux 的 root 權限。混雜模式允許我們監聽偵測網卡看到的所有封包，甚至包括不是發送到指定主機的封包。接著是設定 socket 選項❷，在擷取的封包中要引入 IP 標頭。下一步❸是判定是否在 Windows，如果是，則執行附加步驟，把 IOCTL 發送到網卡驅動程式以啟用混

雜模式。如果您在虛擬機器中執行 Windows，可能會收到 gust 作業系統啟用混雜模式的通知，請同意這個請求。接下來我們準備實際執行一些監聽偵測，在這個範例中，我們只是印出整個 raw 封包❹，但沒有解碼。這只是為了確保監聽偵測程式碼的核心工作是否有正確執行。在監聽偵測單個封包後，再次檢測 Windows，然後在退出腳本程式之前先停用混雜模式❺。

試用與體驗

開啟新的終端機或 Windows 的 cmd.exe 模式，並執行以下命令：

```
python sniffer.py
```

在另一個終端機或 shell 視窗中，可選擇要 ping 的主機。在這裡我們會用 ping nostarch.com：

```
ping nostarch.com
```

在執行 sniffer 程式的第一個視窗中，您應該看到一些與以下內容非常相似的亂碼輸出：

```
(b'E\x00\x00T\xad\xcc\x00\x00\x80\x01\n\x17h\x14\xd1\x03\xac\x10\x9d\x9d\x00\
x00g,\rv\x00\x01\xb6L\x1b^\x00\x00\x00\x00\xf1\xde\t\x00\x00\x00\x00\x00\x10\
x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./01234567', ('104.20.209.3', 0))
```

您可以看到這裡已經捕捉到最初發送到 nostarch.com 的 ICMP ping 請求（是根據在輸出尾端的 nostarch.com IP 位置 104.20.209.3 判斷）。如果您在 Linux 上執行此範例，也會接收到來自 nostarch.com 的回應。

只監聽偵測單個封包是沒什麼用的，所以讓我們新增一些功能來處理更多封包並解碼其內容。

解碼 IP 層

在目前形式中，我們的 sniffer 程式會接收所有 IP 標頭，包括任何更高階層的協定，例如 TCP、UDP 或 ICMP。資訊會被打包成二進位形式，如前面所示，以肉眼很難理解。讓我們對封包的 IP 部分進行解碼，以便從中提取有用的資訊，例如協定的類型（TCP、UDP 或 ICMP）以及來源和目的 IP 位址。這會是將來更深入協定解析的基礎。

如果我們檢查網路上的實際封包的樣貌，您應該會理解要怎麼解碼傳入的封包。有關 IP 標頭的組成，請參見圖 3-1。

<table>
<tr><th colspan="6">Internet Protocol</th></tr>
<tr><td>Bit offset
（位移量）</td><td colspan="2">0-3</td><td>4-7</td><td>8-15</td><td>16-18</td><td>19-31</td></tr>
<tr><td>0</td><td colspan="2">Version
（版本）</td><td>HDR length
（HDR 長度）</td><td>Type of service
（服務類型）</td><td colspan="2">Total length
（總長度）</td></tr>
<tr><td>32</td><td colspan="4">Identification（識別）</td><td>Flags
（旗標）</td><td>Fragment offset
（片段位移量）</td></tr>
<tr><td>64</td><td colspan="3">Time to live
（存活時間）</td><td>Protocol
（協定）</td><td colspan="2">Header checksum
（標頭檢查碼）</td></tr>
<tr><td>96</td><td colspan="6">Source IP address（來源 IP 位址）</td></tr>
<tr><td>128</td><td colspan="6">Destination IP address（目的 IP 位址）</td></tr>
<tr><td>160</td><td colspan="6">Options（選項）</td></tr>
</table>

圖 3-1：典型的 IPv4 標頭結構

我們會解碼整個 IP 標頭（選項欄位除外）並提取協定類型、來源和目的 IP 位址。這表示我們會直接使用二進位檔案來進行相關處理，而且要思考一種以 Python 來分隔 IP 標頭每個部分的策略。

在 Python 中，有幾種方法可以把外部二進位資料轉換為資料結構。我們可以用 ctypes 模組或 struct 模組來定義資料結構。ctypes 模組是 Python 的外部程式庫。此模組為以 C 為基礎的語言提供了橋樑，讓我們能夠在共享程式庫中使用與 C 相容的資料型別和呼叫函式。另一方面，struct 會轉換 Python 值和 C 結構，表

示成為 Python 位元組物件。換句話說，ctypes 模組除了提供許多功能之外還能處理二進位資料型別，而 struct 模組主要是處理二進位資料。

當您在 Web 上瀏覽工具倉庫時，都會看到這兩種方法。本小節會展示這兩種方法是怎麼讀取網路的 IPv4 標頭。二選一由您決定，兩者都能正常運作。

ctypes 模組

以下程式碼片段定義了新的類別 IP，它能讀取封包並把標頭內容解析到單獨的欄位：

```
from ctypes import *
import socket
import struct

class IP(Structure):
    _fields_ = [
         ("version",      c_ubyte, 4),          # 4 bit unsigned char
         ("ihl",          c_ubyte, 4),          # 4 bit unsigned char
         ("tos",          c_ubyte, 8),          # 1 byte char
         ("len",          c_ushort, 16),        # 2 byte unsigned short
         ("id",           c_ushort, 16),        # 2 byte unsigned short
         ("offset",       c_ushort, 16),        # 2 byte unsigned short
         ("ttl",          c_ubyte, 8),          # 1 byte char
         ("protocol_num", c_ubyte, 8),          # 1 byte char
         ("sum",          c_ushort, 16),        # 2 byte unsigned short
         ("src",          c_uint32, 32),        # 4 byte unsigned int
         ("dst",          c_uint32, 32)         # 4 byte unsigned int
    ]
    def __new__(cls, socket_buffer=None):
        return cls.from_buffer_copy(socket_buffer)

    def __init__(self, socket_buffer=None):
        # 人類可讀的 IP 位址
        self.src_address = socket.inet_ntoa(struct.pack("<L",self.src))
        self.dst_address = socket.inet_ntoa(struct.pack("<L",self.dst))
```

這個類別建立了一個 _fields_ 結構來定義 IP 標頭的每一部分。該結構使用在 ctypes 模組中定義的 C 型別。例如，c_ubyte 型別是個 unsigned char、c_ushort 型別是個 unsigned short … 等等。您可以看到每個欄位都與圖 3-1 中的 IP 標頭內容匹配相符。每個欄位描述採用三個引數：欄位名稱（例如 ihl 或 offset）、它採用的值的型別（例如 c_ubyte 或 c_ushort）以及該欄位的位元寬度（例如 ihl 和 version 是 4）。能夠指定位元寬度是很方便的，因為這樣可以讓我們自由

指定需要的任意長度，而不僅僅只能用位元組等級（位元組等級的規範會強制我們定義的欄位只能是 8 位元的倍數）。

IP 類別繼承自 ctypes 模組的 Structure 類別，該類別指定我們必須在建立任何物件之前要定義一個 _fields_ 結構。要填入 _fields_ 結構，Structure 類別會使用 __new__ 方法，該方法把類別參照當作第一個引數，然後建立並返回類別的物件，該物件會傳給 __init__ 方法。當我們建立 IP 物件時，我們會像往常一樣進行，但在底層 Python 呼叫 __new__ 方法建立物件之前（呼叫 __init__ 方法時）立即填入 _fields_ 資料結構。只要您事先定義了結構，就可以把外部網路封包傳給 __new__ 方法來處理，這些欄位應該會很神奇地變成物件的屬性。

您現在已經了解如何把 C 資料型別與 IP 標頭的值相對應。在轉換為 Python 物件時以 C 程式碼作為參考是很有用的，因為這可以無縫轉換為純 Python 語法。有關使用此模組的完整詳細資訊，請參閱 ctypes 的說明文件。

struct 模組

struct 模組提供了可用於指定二進位資料結構的格式字元。在下面的範例中，我們會再定義一個 IP 類別來存放標頭資訊。但在這裡我們會使用格式字元來表示標頭的各個部分：

```
import ipaddress
import struct

class IP:
    def __init__(self, buff=None):
        header = struct.unpack('<BBHHHBBH4s4s', buff)
     ❶ self.ver = header[0] >> 4
     ❷ self.ihl = header[0] & 0xF

        self.tos = header[1]
        self.len = header[2]
        self.id = header[3]
        self.offset = header[4]
        self.ttl = header[5]
        self.protocol_num = header[6]
        self.sum = header[7]
        self.src = header[8]
        self.dst = header[9]

        # 人類可讀的 IP 位址
        self.src_address = ipaddress.ip_address(self.src)
```

```
        self.dst_address = ipaddress.ip_address(self.dst)

        # 對應協定常數到它們的名稱
        self.protocol_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
```

第一個格式字元（範例中是 < ）通常是指定資料的位元組順序（endianness，或譯端序、尾序），或者二進位數字中的位元組順序。C 型別是以機器的原生格式和位元組順序來表示。在本範例中，我們使用的 Kali（x64）是小端序（little-endian）。在小端序機器中，最低有效位元組存放在較低的位址，最高有效位元組則存放在最高的位址。

第二個格式字元代表標頭的個別部分。struct 模組提供了幾種格式字元。對於 IP 標頭，我們只需要格式字元 B（1-byte 無符號字元）、H（2-byte 無符號短整數型別）和 s（需要位元組寬度規範的 byte 陣列；4s 表示 4-byte 字串）。請留意我們的格式字串是怎麼與圖 3-1 的 IP 標頭結構匹配對應的。

請記住，使用 ctypes 可以指定各個標頭部分的位元寬度。若使用 struct，nybble（4-bit 資料單元，也稱為 nibble）沒有格式字元，因此我們必須進行一些操作，從標頭的第一部分取得 ver 和 hdrlen 變數。

在接收到的標頭資料的第一個位元組中，我們只想把**高順位** nybble（位元組中的第一個 nybble）指定給 ver 變數。取得位元組高順位 nybble 的典型方法是把位元組**右移**四位，這相當於在位元組前面加入 4 個 0，導致最後 4 個位元會脫落❶。這樣原始位元組只剩下第一個 nybble。Python 程式碼主要執行了以下的處理：

```
0   1   0   1   0   1   1   0   >> 4
----------------------------
0   0   0   0   0   1   0   1
```

我們想要把**低順位** nybble 或位元組的最後 4 位指定到 hdrlen 變數。要獲取位元組的第二個 nybble 的典型方法，是使用布林 AND 運算子來處理 0xF（0000 1111）❷。布林運算處理後，使得 0 AND 1 產生 0（因為 0 等於 FALSE，而 1 等於 TRUE）。若想要讓表示式為 TRUE，左右兩邊都必須為 TRUE。因此，這項操作會刪除前 4 位元，因為任何與 0 進行 AND 運算的內容都會變成 0。它保留最後 4 位元不變，因為任何與 1 進行 AND 運算的內容都會返回原本的值。本質上，Python 程式是按照如下方式操作位元組：

```
      0   1   0   1   0   1   1   0
AND   0   0   0   0   1   1   1   1
---------------------------
      0   0   0   0   0   1   1   0
```

您不需要非常了解二進位操作就能解碼 IP 標頭，但您會看到某些解碼模式，例如在探索其他駭客的程式碼時會很常見到移位和 AND 的運用，所以很值得深入了解這些技術。

在需要進行位元移位（bit-shifting）的情況下，解碼二進位資料需要花費一翻功夫。但在大多數情況下（例如讀取 ICMP 訊息），設定還算簡單：ICMP 訊息的每一部分都是 8 位元的倍數，struct 模組提供的格式字元也是 8 位元的倍數，所以不需要將一個位元組拆分為單獨的 nybbles。在圖 3-2 所示的 Echo Reply ICMP 訊息中，可以看到 ICMP 標頭的每個參數都能定義到一個具有現成格式字母（BBHHH）的結構中。

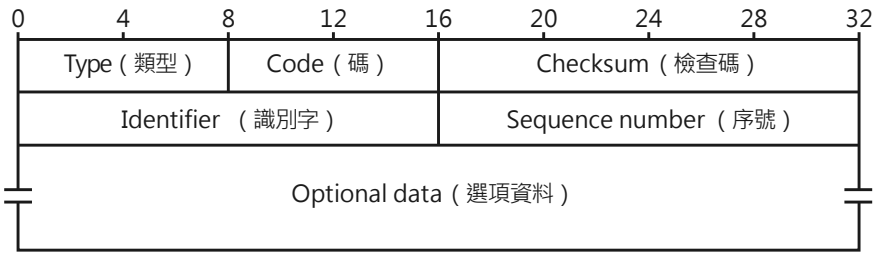


圖 3-2：Echo Reply ICMP 訊息的範例

有一種快速解析此訊息的方法是直接把 1 byte 指定給前兩個屬性，將 2 bytes 指定給後續的三個屬性：

```
class ICMP:
    def __init__(self, buff):
        header = struct.unpack('<BBHHH', buff)
        self.type = header[0]
        self.code = header[1]
        self.sum = header[2]
        self.id = header[3]
        self.seq = header[4]
```

請閱讀 struct 說明文件（https://docs.python.org/3/library/struct.html）來取得使用此模組的完整相關詳細資訊。

使用 ctypes 模組或 struct 模組都可以讀取和解析二進位資料。無論採用哪種方法，您都可以像下列這樣實例化（instantiate）這個類別：

```
    mypacket = IP(buff)
    print(f'{mypacket.src_address} -> {mypacket.dst_address}')
```

在上面的範例中，您使用 buff 變數中的封包資料來實例化 IP 類別。

編寫 IP 解碼程式

讓我們將剛剛建立的 IP 解碼程式實作到一個名為 sniffer_ip_header_decode.py 的檔案中，如下所示：

```
import ipaddress
import os
import socket
import struct
import sys

❶ class IP:
    def __init__(self, buff=None):
        header = struct.unpack('<BBHHHBBH4s4s', buff)
        self.ver = header[0] >> 4
        self.ihl = header[0] & 0xF

        self.tos = header[1]
        self.len = header[2]
        self.id = header[3]
        self.offset = header[4]
        self.ttl = header[5]
        self.protocol_num = header[6]
        self.sum = header[7]
        self.src = header[8]
        self.dst = header[9]

      ❷ # 人類可讀的 IP 位址
        self.src_address = ipaddress.ip_address(self.src)
        self.dst_address = ipaddress.ip_address(self.dst)

        # 對應協定的常數到名稱
        self.protocol_map = {1: "ICMP", 6: "TCP", 17: "UDP"}
        try:
            self.protocol = self.protocol_map[self.protocol_num]
        except Exception as e:
            print('%s No protocol for %s' % (e, self.protocol_num))
            self.protocol = str(self.protocol_num)

    def sniff(host):
        # 與前面範例很類似
```

```
        if os.name == 'nt':
            socket_protocol = socket.IPPROTO_IP
        else:
            socket_protocol = socket.IPPROTO_ICMP

        sniffer = socket.socket(socket.AF_INET,
                                socket.SOCK_RAW, socket_protocol)
        sniffer.bind((host, 0))
        sniffer.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)

        if os.name == 'nt':
            sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

        try:
            while True:
                # 讀取封包
             ❸ raw_buffer = sniffer.recvfrom(65535)[0]
                # 從前 20 bytes 建立 IP 標頭
             ❹ ip_header = IP(raw_buffer[0:20])
                # 印出偵測的協定和主機
             ❺ print('Protocol: %s %s -> %s' % (ip_header.protocol,
                                                 ip_header.src_address,
                                                 ip_header.dst_address))
        except KeyboardInterrupt:
            # 如果是在 Windows 則關掉混雜模式
            if os.name == 'nt':
                sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
            sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    sniff(host)
```

程式一開始是 IP 類別的定義❶，這裡定義了一個 Python 結構，此結構把接收到緩衝區前 20 bytes 的資料對應成好處理的 IP 標頭資料。如您所見，我們識別的所有欄位都與標頭結構有很好地對應匹配。接著會做一些處理來產生方便閱讀的輸出結果，這些結果指示正在使用的協定和連線中涉及的 IP 位址❷。使用了新建的 IP 結構之後，接著編寫處理邏輯來繼續讀取封包並解析其資訊。我們讀入封包❸，隨後傳入前 20 bytes ❹來初始化 IP 結構，接著就是印出獲取的資訊❺。讓我們動手試一試吧！

試用與體驗

讓我們測試一下之前的編寫的程式碼，看看從發送的原始封包中提取了什麼樣的資訊。我們強烈建議讀者在 Windows 機器上進行這項測試，因為這樣就能看到 TCP、UDP 和 ICMP，讓您可以進行一些有趣簡潔的測試（例如，打開瀏覽器）。如果您只有 Linux 可用，請執行前面的 ping 測試來查看其執行的情況。

請開啟終端機並鍵入如下內容：

```
python sniffer_ip_header_decode.py
```

由於 Windows 的對談回應很快，所以您可能會馬上看到輸出結果。我們透過打開 Internet Explorer 並連到 www.google.com 來測試此腳本程式，以下是腳本程式的輸出：

```
Protocol: UDP 192.168.0.190 -> 192.168.0.1
Protocol: UDP 192.168.0.1 -> 192.168.0.190
Protocol: UDP 192.168.0.190 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 192.168.0.187 -> 74.125.225.183
Protocol: TCP 74.125.225.183 -> 192.168.0.187
Protocol: TCP 192.168.0.187 -> 74.125.225.183
```

因為我們並沒有對這些封包進行深入檢查，所以只能猜測這個串流大概是用來做什麼的。我們的猜測是，前幾個 UDP 封包是 DNS 查詢，用來確定 google.com 的位址，隨後的 TCP session 是我們的機器的實際連線，以及從 Web 伺服器下載的內容。

若想要在 Linux 上執行相同的測試，我們可以 ping google.com，結果看起來會像下列的內容：

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
```

您已經可以看到限制了：這裡只能看到回應，且只針對 ICMP 協定。不過因為我們的目標就是要建構主機探索掃描程式，所以這是完全可以接受的。接下來是把前面解碼 IP 標頭的相同技巧套用到 ICMP 訊息的解碼。

解碼 ICMP

現在我們已經可以完全解碼監聽到的封包的 IP 層了，我們還需要解碼掃描程式從發送 UDP 封包到關閉埠號所引發的 ICMP 回應。ICMP 訊息的內容會有很大差異，但每條訊息都包含三個一致的元素：type（類型）、code（代碼）和 checksum（檢查碼）。type 和 code 欄位告知接收主機送達的是哪種 ICMP 訊息，然後指示要如何正確解碼。

以我們的掃描程式來說，要尋找 type 值 3 和 code 值 3。這對應到 ICMP 訊息的 Destination Unreachable，而 code 值 3 表示引發 Port Unreachable 錯誤。Destination Unreachable ICMP 訊息的示意圖請參閱圖 3-3。

Destination Unreachable 訊息		
0-7	8-15	16-31
Type = 3	Code（代碼）	Header checksum（標頭檢查碼）
Unused（未使用）		Next-hop MTU
IP 標頭和原始資料包的前 8 bytes 內容		

圖 3-3：Destination Unreachable ICMP 訊息的結構

如圖所見，前 8 bits 是類型，後 8 bits 包含 ICMP 代碼。需要注意的是，當主機發送這些 ICMP 訊息時，實際上也包含了生成回應的原始訊息的 IP 標頭。我們還需要再次檢查發送的原始資料包（datagram）的 8 bytes 內容，以確保掃描程式生成了 ICMP 回應。為此，我們只需切掉接收緩衝區的最後 8 bytes 內容，提取掃描程式發送的魔法字串（magic string）。

讓我們為之前的 sniffer 程式加入更多程式碼，以俱備解碼 ICMP 封包的功能。請把之前的檔案另存為 sniffer_with_icmp.py 檔，並加入以下程式碼：

```
import ipaddress
import os
import socket
import struct
import sys

class IP:
--省略--
```

```
❶ class ICMP:
      def __init__(self, buff):
          header = struct.unpack('<BBHHH', buff)
          self.type = header[0]
          self.code = header[1]
          self.sum = header[2]
          self.id = header[3]
          self.seq = header[4]

      def sniff(host):
      --省略--

              ip_header = IP(raw_buffer[0:20])
              # 如果是 ICMP，則使用它
           ❷ if ip_header.protocol == "ICMP":
                  print('Protocol: %s %s -> %s' % (ip_header.protocol,
                          ip_header.src_address, ip_header.dst_address))
                  print(f'Version: {ip_header.ver}')
                  print(f'Header Length: {ip_header.ihl} TTL: {ip_header.ttl}')

                  # 從 ICMP 封包開始的位置計算
               ❸ offset = ip_header.ihl * 4
                  buf = raw_buffer[offset:offset + 8]
                  # create our ICMP structure
               ❹ icmp_header = ICMP(buf)
                  print('ICMP -> Type: %s Code: %s\n' %
                       (icmp_header.type, icmp_header.code))

          except KeyboardInterrupt:
          if os.name == 'nt':
              sniffer.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
          sys.exit()

  if __name__ == '__main__':
      if len(sys.argv) == 2:
          host = sys.argv[1]
      else:
          host = '192.168.1.203'
      sniff(host)
```

這段程式碼是在現有的 IP 結構下建立了 ICMP 結構❶。當封包接收主迴圈確定收到了 ICMP 封包時❷，會計算原始封包中 ICMP 本體所在的 offset 位移量❸，然後建立緩衝區❹並印出 type 和 code 欄位的內容。長度計算是以 IP 標頭 ihl 欄位為基準，該欄位指出 IP 標頭中包含的 32-bits word（4-byte 區塊）的數量。因此，把這個欄位乘以 4 就能知道 IP 標頭的大小，從而知道下一個網路層（在本範例中是 ICMP）的起始位置。

如果使用之前典型的 ping 測試來快速執行這段程式碼，得到的輸出應該會略有不同：

```
Protocol: ICMP 74.125.226.78 -> 192.168.0.190
ICMP -> Type: 0 Code: 0
```

這表示 ping（ICMP Echo）回應有正確接收和解碼。我們現在準備實作最後一部分的處理邏輯，那就是發送 UDP 封包並解譯其結果。

現在加入 ipaddress 模組的使用，以便透過主機探索掃描的動作能覆蓋整個子網路。請將 sniffer_with_icmp.py 腳本程式檔另存為 scanner.py 檔，並加入以下程式碼：

```
import ipaddress
import os
import socket
import struct
import sys
import threading
import time

#子網路
SUBNET = '192.168.1.0/24'
#我們要在 IMCP 回應中檢查的魔法字串（magic string）
MESSAGE = 'PYTHONRULES!' ❶

class IP:
--省略--

class ICMP:
--省略--

#這裡會使用我們的魔法訊息掃出 UDP 資料包
def udp_sender(): ❷
    with socket.socket(socket.AF_INET, socket.SOCK_DGRAM) as sender:
        for ip in ipaddress.ip_network(SUBNET).hosts():
            sender.sendto(bytes(MESSAGE, 'utf8'), (str(ip), 65212))

class Scanner: ❸
    def __init__(self, host):
        self.host = host
        if os.name == 'nt':
            socket_protocol = socket.IPPROTO_IP
        else:
            socket_protocol = socket.IPPROTO_ICMP

        self.socket = socket.socket(socket.AF_INET,
                                    socket.SOCK_RAW, socket_protocol)
        self.socket.bind((host, 0))
        self.socket.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
```

```
        if  os.name == 'nt':
            self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)

    def sniff(self): ❹
        hosts_up = set([f'{str(self.host)} *'])
        try:
            while True:
                # 讀取一個封包
                print('.',end='')
                raw_buffer = self.socket.recvfrom(65535)[0]
                # 從前 20 bytes 來建立 ip 標頭
                ip_header = IP(raw_buffer[0:20])
                if ip_header.protocol == "ICMP":
                    offset = ip_header.ihl * 4
                    buf = raw_buffer[offset:offset + 8]
                    icmp_header = ICMP(buf)
                    # 檢查 TYPE 和 CODE 是否為 3
                    if icmp_header.code == 3 and icmp_header.type == 3:
                        if ipaddress.ip_address(ip_header.src_address) in ❺
                                        ipaddress.IPv4Network(SUBNET):

                            # 確定有我們的魔法訊息
                            if raw_buffer[len(raw_buffer) - len(MESSAGE): ] == ❻
                                        bytes(MESSAGE, 'utf8'):
                                tgt = str(ip_header.src_address)
                                if tgt != self.host and tgt not in hosts_up:
                                    hosts_up.add(str(ip_header.src_address))
                                    print(f'Host Up: {tgt}') ❼
        # 處理 CTRL-C
        except KeyboardInterrupt: ❽
            if  os.name == 'nt':
                self.socket.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)

            print('\nUser interrupted.')
            if hosts_up:
                print(f'\n\nSummary: Hosts up on {SUBNET}')
            for host in sorted(hosts_up):
                print(f'{host}')
            print('')
            sys.exit()

if __name__ == '__main__':
    if len(sys.argv) == 2:
        host = sys.argv[1]
    else:
        host = '192.168.1.203'
    s = Scanner(host)
    time.sleep(10)
    t = threading.Thread(target=udp_sender) ❾
    t.start()
    s.sniff()
```

最後這小段程式應該很容易理解。我們定義了一個簡單的字串簽章❶，以便測試回應是否來自我們最初發送的 UDP 封包。我們的 udp_sender 函式❷只會接收在腳本程式頂端指定的子網路，遍訪該子網路中的所有 IP 位址，然後向它們發送 UDP 資料包。

隨後我們定義一個 Scanner 類別❸。為了進行初始化，我們把一個主機當作引數傳給它。當它進行初始化時，我們會建立一個 socket，如果是在 Windows 執行，則打開混雜模式，並把 socket 變成為 Scanner 類別的一個屬性。

sniff 方法❹會探索監聽網路，其步驟與前面的範例相同，但這裡會記錄有哪些主機已啟動。如果檢測到預期的 ICMP 訊息，我們會先檢查以確保 ICMP 回應是來自我們的目標子網路❺。接著執行最後的檢查，以確保 ICMP 回應中有包含我們的魔法字串❻。如果所有這些檢查都通過，就會印出 ICMP 訊息起源的主機 IP 位址❼。當我們按下 CTRL-C 結束監聽過程時，會進行鍵盤中斷的相關處理❽。也就是說，如果在 Windows 上，我們會關閉混雜模式並印出所有還在活動主機的排序清單。

__main__ 區塊是處理設定的工作：它會建立 Scanner 物件，接著只休眠幾秒鐘，隨後在呼叫 sniff 方法之前，在單獨的執行緒❾中生成 udp_sender 以確保我們沒有干擾監聽回應。讓我們動手試一試這支程式吧！

試用與體驗

現在讓我們使用這支掃描程式在本機網路執行。您可以在 Linux 或 Windows 上執行，因為結果是相同的。在作者的執行範例中，我們所在的本機 IP 位址是 192.168.0.187，因此我們在掃描程式中把子網路設為 192.168.0.0/24。如果執行掃描程式所得到的輸出太雜亂，注釋掉所有 print 陳述句，只留下最後一條告知主機正在回應的陳述句。

```
python.exe scanner.py
Host Up: 192.168.0.1
Host Up: 192.168.0.190
Host Up: 192.168.0.192
Host Up: 192.168.0.195
```

IPADDRESS 模組

我們的掃描程式會用到 ipaddress 程式庫，它允許我們輸入子網路遮罩，例如 192.168.0.0/24，並讓掃描程式能正確地處理。

ipaddress 模組讓處理子網路和定址變得非常容易。舉例來說，您可以使用 Ipv4Network 物件執行如下簡單的測試：

```
ip_address = "192.168.112.3"

if ip_address in Ipv4Network("192.168.112.0/24"):
    print True
```

或者，如果您想把封包發送到整個網路，可建立簡單的迴圈來處理：

```
for ip in Ipv4Network("192.168.112.1/24"):
    s = socket.socket()
    s.connect((ip, 25))
    # 傳送 mail 封包
```

在想要一次處理整個網路時，這個功能會大幅簡化您的程式設計工作，也非常適合用在我們的主機探索工具。

以我們前面執行的快速掃描來說，只需要幾秒鐘就能得到結果。把結果的 IP 位址與家中路由器的 DHCP 表進行交叉比對，就能驗證結果是否正確。您可以輕鬆擴充本章所學到的內容，進一步解碼 TCP 和 UDP 封包以及開發其他掃描工具。這支掃描程式對於我們在第 7 章建構的木馬框架也很有用，它允許部署的木馬掃描本機網路來尋找額外的目標。

現在您已經了解網路在高階和低階運作的基礎原理，接下來讓我們探索一套非常成熟的 Python 程式庫 Scapy。