ANALYSIS AND RESEARCH INTO THE PURPOSE AND CAUSE OF 'LOVE' NOISE STORM

Anthony Power
(20098384)

Table of Contents

Declaration

I certify that this assignment is all my own work and contains no plagiarism. By submitting this assignment, I agree to the following terms:
Any text, diagrams or other material copied from other sources (including, but not limited to, books, journals, and the internet) have been clearly acknowledged. and referenced as such in the text by the use of ‘quotation marks’ (or indented italics for longer quotations) followed by the author’s name and date [e.g. (Byrne, 2008)] either in the text or in a footnote/endnote. These details are then. confirmed by a fuller reference in the bibliography.

I have read the sections on referencing and plagiarism in the handbook or in the SETU Plagiarism policy and I understand that only assignments which are free of plagiarism will be awarded marks. I further understand that SETU has a plagiarism policy which can lead to the suspension or permanent expulsion of students in serious cases.

Signed: Anthony Power

Dated: 21 March 2025

Introduction

Overview:

The overall goal of this research project is to determine the cause of these ICMP 'LOVE' noise storms and to also determine the purpose of these noise storms besides the obvious disruption to these providers services. A secondary objective of this research is also to determine the origin of these 'attacks' as initial research from GreyNoise https://github.com/GreyNoise-Intelligence/2024-09-noise-storms has determined that the IP addresses used are spoofed and that the ASN (Associated Serial Number) of some of the packets correspond with *"CDN organization servicing QQ, WeChat, and WePay"* as well as GreyNoise themselves previously writing about a less refined version of these Noise Storms using over 1 million spoofed IPV4 addresses.

Motivation:

The motivation behind this project is one of personal interest in helping to solve the purpose of these Noise Storms and to learn more about them. As this "noise storm" has existed it has been refined from a flood of "≥ 1 million IP addresses in a 24-hour period" according to this notebook by GreyNoise analyst Bob Rudis https://observablehq.com/@greynoise/noise-storms last updated in March 2023, to a concentrated attack of 24 or less spoofed IP addresses sending ICMP packets to a single victim host in its current form. As this attack has effected the service of multiple cloud and network providers there is also motivation in figuring this out to help prevent or defend against future attacks, which according to the notebook linked occur semi-frequently with the notebook stating since it origin, 2020 they had observed 25 separate storms when the notebook was created.

Summary:

This report will go through the discovered technologies used in creation of these noise storms, the intended and non-intended victims of the operation of these storms and the methods used to mask the usage of the technologies used. It will also go through the method used to spoof the IP addresses and through process of elimination on various hypotheses in terms of the usage of the storms and finally present a theory on the current process's required to recreate the behaviour seen in these storms as well as the intended purpose of these storms.

The report will later touch on the research conducted into this topic and as a result of this research some scripts and modified tools to be used to help detect and recreate the conditions seen in the Noise Storm in a controlled testbed. Below will introduce the noise storms themselves and the currently discovered technologies used as part of them.

Introduction to Noise Storms:

As GreyNoise has been reporting, they have sensed these large mainly TCP and ICMP noise storms over the last four years with one of the main linking factors between these is the prevalence of the ASCII string 'LOVE' or hexadecimal string '0x4c4f5645' embedded in these packets, along with seemingly random varying bytes following.

As GreyNoise writes *"Noise Storms are large-scale spoofed packet events"* and the first one they observed was in January 2020, "*coinciding with the U.S. military action against Iranian General Soleimani"*.
This has led some researchers to believe that these 'attacks' are politically motivated but no concreate evidence has been produced yet and the dates in the GreyNoise packet capture do not line up with significant worldwide events as this initial finding suggests.

"They involve millions of spoofed IP addresses, primarily appearing to originate from Brazil in recent months". This is supported by the IP's present in the capture provided by them, as in any packets that fitted the criteria listed had 'spoofed IP's' that originate from Sao Paulo, Brazil.

Other information about this specific noise storm is taken for the GreyNoise GitHub 'Background' section for the repository linked above:

- traffic is mostly TCP (port 443) and ICMP,
- TTL's are between 120 and 200.
- storms have evolved to be more targeted, hitting smaller parts of the internet but with increased intensity.
- They often coincide with significant geopolitical or military events but not always.
- TCP traffic intelligently spoofs window sizes to mimic packets from various operating systems.
- Recent storms have avoided AWS but hit other providers like Cogent, Lumen, and Hurricane Electric
- The ASN for the ICMP traffic is associated with a CDN organization servicing QQ, WeChat, and WePay

The theories that GreyNoise were investigating in terms of the purpose of these storms are as follows *"covert communications, DDoS attempts, misconfigured routers, command and control mechanisms, or attempts to create network congestion for traffic manipulation"*.

Introduction to HiPerConTracer (High-Performance Connectivity Tracer):

From initial investigations conducted by a researcher David Schuetz written in his blog https://darthnull.org/noisestorms/ was that even though when investigating a packet using Wireshark it shows as just a simple ICMP (ping) request packet with a 52 byte payload with 4c4f5645 embedded in the data which is known to be 'LOVE' in ASCII as shown below.

Packet From GreyNoise Capture in Wireshark (v4.4.0-0-g009a163470b5):

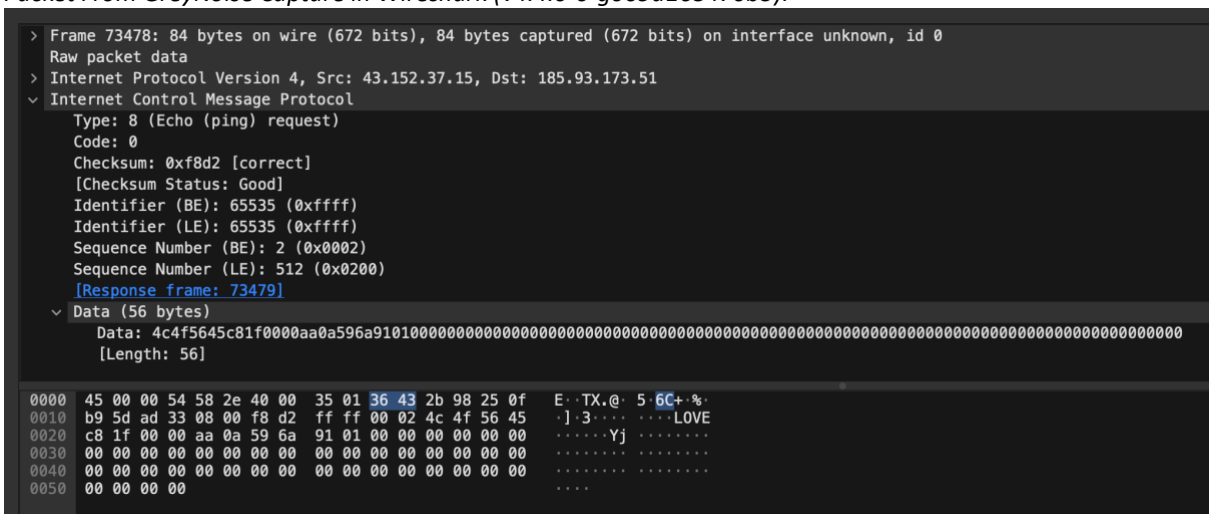

Figure 1: Example Packet From GreyNoise Capture in Wireshark

David Schuetz discovered that if a similar structure packet is analysed using Pyshark, a python wrapper for tshark/Wireshark it produces the following output.

```
(.venv) (base) anthonypower@Anthonys-MBP noise-storm-icmp-brazil % python3 pyshark_test.py
Packet (Length: 84)
Layer RAW
:Layer IP
:       0100 .... = Version: 4
        .... 0101 = Header Length: 20 bytes (5)
        Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
        0000 00.. = Differentiated Services Codepoint: Default (0)
        .... ..00 = Explicit Congestion Notification: Not ECN-Capable Transport (0)
        Total Length: 84
        Identification: 0x5eee (24302)
        010. .... = Flags: 0x2, Don't fragment
        0... .... = Reserved bit: Not set
        .1.. .... = Don't fragment: Set
        ..0. .... = More fragments: Not set
        ...0 0000 0000 0000 = Fragment Offset: 0
        Time to Live: 53
        Protocol: ICMP (1)
        Header Checksum: 0x2f83 [validation disabled]
        Header checksum status: Unverified
        Source Address: 43.152.37.15
        Destination Address: 185.93.173.51
        Stream index: 7
Layer ICMP
:       Type: 8 (Echo (ping) request)
        Code: 0
        Checksum: 0x91bb [correct]
        Checksum Status: Good
        Identifier (BE): 65535 (0xffff)
        Identifier (LE): 65535 (0xffff)
        Sequence Number (BE): 1 (0x0001)
        Sequence Number (LE): 256 (0x0100)
Layer HIPERCONTRACER
:       Magic Number: 0x4c4f5645
        Send TTL: 219
        Round: 25
        Checksum Tweak: 0x0000
        Send Time Stamp: May 23, 2024 07:05:13.287680000 UTC
```

Figure 2: Output of Packet in PyShark

This program shows that the packet contains a 'HIPERCONTRACER' layer. By inputting this term into a search engine the top results found are for a traceroute service created by Thomas Dreibholz, in 2012 as an advanced Ping/Traceroute measurement tool that performs "regular Ping and Traceroute measurements over IPv4 and IPv6 among sites" https://www.nntb.no/~dreibh/hipercontracer/ and then the results can be exported to a file, csv or into an SQL database. The source code for this program is available on GitHub at https://github.com/dreibh/hipercontracer.

As of February 2025, the HiPerConTracer (High-Performance Connectivity Tracer) tool has been packaged into a framework of the same name. The core functionality of the tool remains the same however the framework introduces additional tools for helping to obtain, process, collect, store, and retrieve measurement data which was previously done using the tool itself. Variants of these additional tools existed in previous versions of the program such as the "Reverse Tunnel Tool" https://github.com/dreibh/hipercontracer?tab=readme-ov-file - the-hipercontracer-reverse-tunnel-tool but they have now been officially launched.

Through initial investigation into this program Thomas determined that the version of this tool used to produce the packets in the GreyNoise capture was a modified version of the initial source code in where the 'Magic Number' is always set to '4c4f5645' or 'LOVE' and the send timestamps are spoofed to hide the actual time these are sent and they are also converted to milliseconds from UTC and stored in little-endian format.

The reason the programs usage was discovered is quite interesting. As covered in this article from the Medium entitled “A Detour In Time: Meta-Analysis of GreyNoise Ping Storms” https://medium.com/@colin.l.morrell/a-detour-in-time-meta-analysis-of-greynoise-ping-storms-6b9262d0daea the author Colin Morell details the reasons why this program was discovered from some packets instead of others. Ultimately it came down to the structure of the HiPerConTracer dissector for Wireshark.

```
96      // Check for plausible send time stamp:
97      // * After:  01.01.2016 00:00:00.000000
98      // * Before: 31.12.2099 23:59.59.999999
99      // Time stamp is microseconds since 29.09.1976 00:00:00.000000.
100     sendTimeStamp += UINT64_C(212803200000000);
101     if ( (sendTimeStamp < UINT64_C(1451602800000000)) ||
102          (sendTimeStamp > UINT64_C(4102441199999999)) )
103       return false;
```

Figure 3: Timestamp Check in Wireshark Dissector

The image above shows an extract from the dissector in question and checks for a plausible timestamp and also adds a static value to the timestamp which is discussed earlier in the article as the ‘Dreibh offset’ being the time in which is the epoch date of the program’s author Thomas Dreibholz’s suspected birthdate. This was why when the timestamp where converted back to microseconds or in more recent versions of the program nanoseconds and converted to little endian, it was still of as it still had this epoch applied to it. This is only done internally and the timestamp is converted to Unix timestamp before it is sent. Looking at the dissector the author determined that there three conditions for a double plausible Unix epoch timestamp that would be incorrectly parsed by the dissector leading to the discovery of the program.

1 — (t) timestamp is an accurately generated timestamp with millisecond precision stored in big-Endian format,
2 — r(t) the reverse of the timestamp is the little-Endian representation of the original timestamp, and
3 — d(r(t)) the reversed timestamp in integer form with the ‘dreibh epoch’ added is a plausible, microsecond-precision Dreibholz timestamp when parsed by the HiPerConTracer dissector.

If all three of these conditions are met then the dissector generates a false positive result and the output of the program is dissected as it is a valid HiPerConTracer header.

$$
\begin{aligned}
t_{datetime} &= \texttt{2024-09-15 01:25:34} \\
t_{int} &= \texttt{1726363534848} \\
t &= \texttt{0000 0191 F349 0600} \\
r(t) &= \texttt{0006 49F3 9101 0000} \\
r(t)_{int} &= \texttt{1770160318906368} \\
d(r(t)) = r(t)_{int} + dreib &= \texttt{1982963518906368} \\
d(r(t))_{datetime} &= \texttt{2032-11-01 23:11:58}
\end{aligned}
$$

Figure 4: Example of Reversing Timestamp

The image above from the article shows this in action with t=timestamp, r=reversed and d=’dreib epoch’
Here it can be seen that all three of the conditions are satisfied and the header is parsed as it would normally due to this false positive.

Literature Review

2.1 HiPerConTracer Traceroute/Ping service:

As previously discussed in the earlier section the suspected program used to generate the ICMP packets that were found in the 'noise storm' was a traceroute/ping service called HiPerConTracer created by Thomas Dreibholz in 2012. This program polls selected destination IPs from a source IP and collects data on the round time for a response and the number of hops or devices the packet travels through before being returned and timestamps among other information. The structure of the unmodified 'traceservice' header it creates can be seen below taken from David Schuetz blog post on the subject.

```
/* Setup list of header fields */
static hf_register_info hf[] = {
  { &hf_magic_number,   { "Magic Number",    "hipercontracer.magic_number",   FT_UINT32, BASE_HEX, NULL,                0x0, "An identifier chosen by the sender upon startup",
HFILL } },
  { &hf_send_ttl,       { "Send TTL",        "hipercontracer.send_ttl",       FT_UINT8,  BASE_DEC, NULL,                0x0, "The IP TTL/IPv6 Hop Count used by the sender",
HFILL } },
  { &hf_round,          { "Round",           "hipercontracer.round",          FT_UINT8,  BASE_DEC, NULL,                0x0, "The round number the packet belongs to",
HFILL } },
  { &hf_checksum_tweak, { "Checksum Tweak",  "hipercontracer.checksum_tweak", FT_UINT16, BASE_HEX, NULL,                0x0, "A 16-bit value to ensure a given checksum for the
ICMP/ICMPv6 message", HFILL } },
  { &hf_seq_number,     { "Sequence Number", "hipercontracer.seq_number",     FT_UINT16, BASE_DEC, NULL,                0x0, "A 16-bit sequence number",
HFILL } },
  { &hf_send_timestamp, { "Send Time Stamp", "hipercontracer.send_timestamp", FT_ABSOLUTE_TIME, ABSOLUTE_TIME_UTC, NULL, 0x0, "The send time stamp (microseconds since September
29, 1976, 00:00:00)", HFILL } }
};
```

As can be seen there is six desired fields, these being the 'Magic Number', send ttl, round number, checksum tweak, sequence number and the send timestamp. The 'Magic Number' is by default set by a random calculation as in the snippet below.

```
    ResultsMap(resultsMap),
    SourceAddress(sourceAddress),
    SourcePort(sourcePort),
    DestinationPort(destinationPort),
    NewResultCallback(newResultCallback),
    MagicNumber( ((std::rand() & 0xffff) << 16) | (std::rand() & 0xffff) )
{
  Identifier       = 0;
  TimeStampSeqID   = 0;
  PayloadSize      = 0;
  ActualPacketSize = 0;
}
```

Figure 5: Setting of Header in HiPerConTracer Program

In the theorised modified version of this script this is set to 0x4c4f5645 or 'LOVE' in ASCII. The checksum tweak in the modified program is always 0x0000 and the sequence number is missing but it is possible that this sequence number field is used to define the sequence number for the ICMP header and not this header itself as even when running the unmodified version of this program below it does not appear in the header itself.

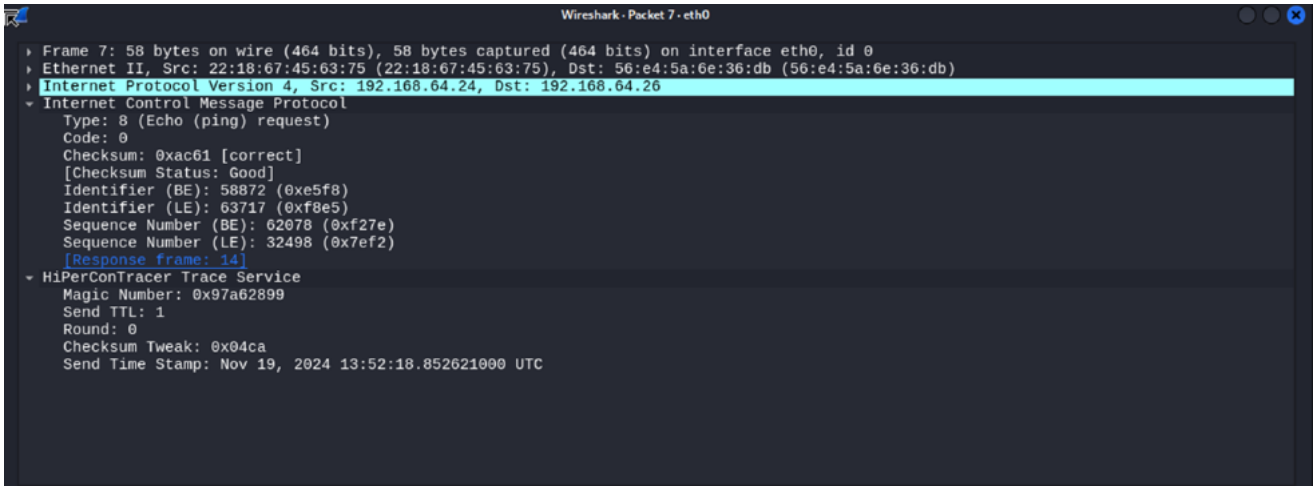

Figure 6: Capture of Packet from Unmodified HiPerConTracer Program

2.2 PcapPlusPlus:

For the detection script that will later be detailed the goal of the script was it could run through a large packet capture file and pick out the source and destination hosts as well as a sample packet summary from each source host in a timely manner. To begin, the initial prototyping of this script was conducted using Pyshark similar to David Schuetz's testing. This would provide the desired results but was inefficient and would take up to 15 mins to run to completion on the merged ICMP packet capture provided by GreyNoise.

As the plan for this script was to be ran in a timely manner this was deemed too slow and the focus was shifted to a lower-level language as the time taken to translate the Python code to C was what was deemed the main inefficiency in the program. After looking at alternatives such as libcap, the decision was chosen to use Pcap++ https://pcapplusplus.github.io/ a C++ library for capturing and parsing packets. This was chosen due to the fact It could obtain raw packet data from the packets which was needed to obtain the raw hex value of fields in the header such as the 'Magic Number'.

Pcap++ is a combination of three separate C++ libraries that link into each other. This can be seen from the dependency model taken from their *"API Reference"* page on their website https://pcapplusplus.github.io/docs/api.

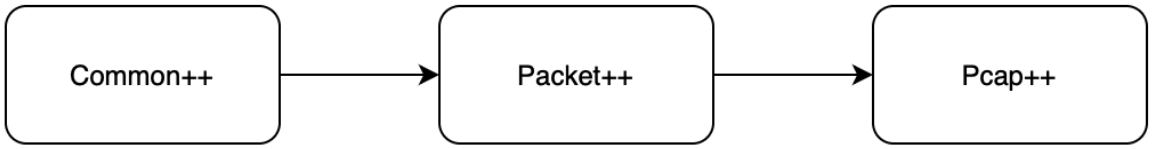

Figure 7: Combination of Libraries for PcapPlusPlus

Common++ - a library with common code utilities which in this case is used by the other two libraries.
Packet++ - a library for parsing, creation and editing of network packets.
Pcap++ - a C++ wrapper for packet capture engines such as libcap.

Pcap++ was also chosen due to its built-in acknowledgement of unknown layers as in layers not recognised by the library itself such as the previously discussed 'HIPERCONTRACER' layer. Having an unknown layer as part of the captured packet means the contents of that layer can be accessed directly and the packet does not have to accessed in its raw format before then accessing that layer's 'payload'. This was an initial issue had with Pyshark when analysing an initial packet as it would not allow direct access to this layer as it was unknown to the program, so the only way to access it was to access the raw packet and get the data from said raw packet.

This was an issue of convenience and also efficiency as it would require loading nearly twice as many packets as the ones in the actual capture impacting performance.

2.3 IP in IP Tunnelling:

IP in IP tunnelling is the protocol that encapsulates one IP packet into another IP packet like the image shown below from RedHat's developer blog https://developers.redhat.com/blog/2019/05/17/an-introduction-to-linux-virtual-interfaces-tunnels - ipip_tunnel.

Figure 8: IP Packet Encapsulation for IPIP tunnel

As can be seen the packet is spilt into four headers, Ethernet, IPv4 Header, Inner IPv4 Header and the Payload. The payload itself is the remaining headers in the packet such as for a normal non-modified HiPerConTracer packet would be two additional headers in the ICMP header and HiPerConTracer header. In ideal operation of a IP in IP tunnel it would be used to create a direct connection between two devices on separate networks that are both connected to an intermediary network such as the internet.

Too create these tunnel connections at least on Linux require the ipip module to be loaded using `modprobe `which requires the package `kmod` installed with a package manager and when that module is loaded it will allow creation of ipip devices and a default ipip0 will be created on the device with local=any and remote=any which are the devices for the source 'local' address and 'destination' address of the tunnel.

Following the steps from the RedHat blog on Server A and the opposite on Server B (the two servers that need connecting), will setup the tunnel connection.

```
On Server A:
# ip link add name ipip0 type ipip local LOCAL_IPv4_ADDR remote REMOTE_IPv4_ADDR
# ip link set ipip0 up
# ip addr add INTERNAL_IPV4_ADDR/24 dev ipip0
Add a remote internal subnet route if the endpoints don't belong to the same subnet
# ip route add REMOTE_INTERNAL_SUBNET/24 dev ipip0
```

Figure 9: Steps to Create IPIP tunnel interface on Server

This tunnel is just designed for connecting two IPv4 subnets through a public IPv4 internet and do not protect against IP spoofing which is why it was chosen. If a user wants to send traffic through the tunnel it sends to the INTERNAL_IPV4_ADDR of Server B and it will take that traffic and decapsulate it at the destination. The lack of security in the usage of this protocol allows an actor to easily spoof a host IP address as will be seen in the next section and the later explanation and setup of the developed theory.

2.4 Tunnelled Temporal Lensing Attacks:

This is relatively new attack vector that was discovered during the investigation into possible theories. From this academic paper from DistriNet, KU Leuven https://papers.mathyvanhoef.com/usenix2025-tunnels.pdf it explains a 'temporal lensing attack' as *"an amplification effect is achieved by concentrating packets in time. For instance, the attacker sends packets for 10 seconds and uses protocol properties to ensure they arrive at the victim in a window of less than one second, resulting in an amplification factor of at least 10".* This acts as a Denial of Service or DoS attack where the timing of the packets sent to a network are concentrated to reach the destination all within a short time span of each other for a selection usually sent over a much longer time period. The tunnelled version of this attack adds the extra step of tunnelling the packets through an IPIP tunnel to spoof the IP addresses which means that the victim of the attack or the router on the victim's network must in the case of an ICMP request route that traffic to the server that corresponds to the spoofed IP address.

As part of the research in this paper they scanned multiple tunnelling protocols and determined vulnerable hosts. A host considered vulnerable if they

1. Decapsulate arbitrary traffic
2. Forward traffic with their source IP

For their scan they chose three methodologies Standard, ICMP Echo/Reply and TTL Expired and scanned the selected hosts they discovered among other that they could conduct two attacks using Echo/Reply where the source of the encapsulated packet is the same as the destination address or the source address of the encapsulated packet is a spoofed IP address and the reply is sent to the spoofed IP address instead of the original host. The second method is more in line with the type of attack seen as it was replying to an IP address associated with myqcloud.com which is part of Tencent's Content Distribution Network or CDN according to the Shodan.io search below.

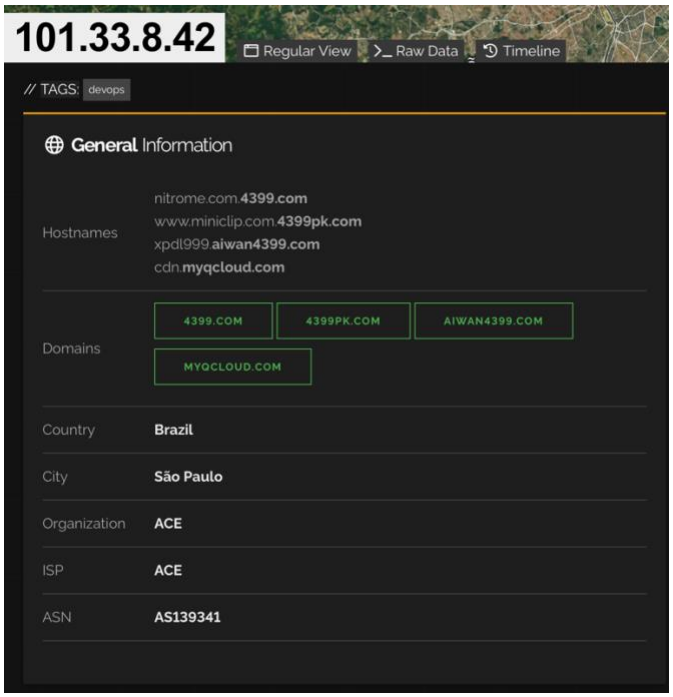

Figure 10: Shodan.io Output for 101.33.8.42

Looking further into paper a table can be seen of the most vulnerable scanned hosts for each tunnelling protocol as seen below.

Protocol	Total Hosts	Top 3 ports			Top 3 domains		
		Top 1 port	Top 2 port	Top 3 port	Top 1 domain	Top 2 domain	Top 3 domain
IPIP	10,000	HTTP (54%)	HTTPS (53%)	SNMP (23%)	fbcdn (2156)	myqcloud (881)	aiwan4399 (577)
IP6IP6	3099	HTTPS (79%)	HTTP (67%)	SSH (7%)	fbcdn (2015)	whatsapp (357)	infinitum (65)
GRE	10,000	BGP (37%)	GTP-C (18%)	GTP-U (18%)	telenet (310)	windstream (221)	163data (194)
GRE6	24	HTTP (42%)	SSH (29%)	BGP (25%)	N/A	N/A	N/A
6in4	10,000	NTP (51%)	SNMP (46%)	HTTP (5%)	proxad (620)	49-tataidc (476)	14-tadaidc (327)
4in6	1258	SNMP (43%)	NTP (16%)	HTTP (14%)	bbtec (384)	synology (29)	byteark (27)

Figure 11: List of Top Ports & Domains for Selected Protocols

As can be seen the IPIP, GRE and 6in4 had the most hosts and interestingly myqcloud is the top 2 domain for the IPIP protocol with a selection of 881 hosts vulnerable to tunnelling via IPIP.

Methodology

3.1 Treating 'LOVE' packets as isolated traffic:

This was one of the main problems with deciphering the Noise Storm traffic in the first place, as the storm itself was so loud as in the number of packets sent the concern was that the packets were a smokescreen for other activity on the network which was hiding for lack of a better term in 'the eye of the storm'. In David Schuetz's initial investigation in the blog post titled "Update on August Ping Storms" https://darthnull.org/noise-storm-update/ he splits the ICMP traffic in the capture into five different ping payloads as seen in the table below.

Number	Payload fragment (hex)	Payload start (ASCII)	My Name
2,851,686	4c4f5645 605a0000	LOVE`Z..	LOVE
5,706	20212223 24252627	.!"#$%&'	Linux Ping
4,432	17ec4e6a 6f4c7354	(nothing legible)	Seventeens
399	6c69626f 70696e67	liboping	liboping
4	6abcccf9 59b1d941	(nothing legible)	~~oddballs~~ Ramius

Figure 12: Grouping of Different Ping Payloads

The table shows the disparity in the number of packets sent in each payload with everything else being dwarfed by the LOVE ping payload. The only one of note was the Seventeens which according to his research later in that blog appears to another high precision ping test.

Looking at the TCP packets in the capture they appeared to be a SYN-flood attack targeting a cloud router on that network with it having to repeatedly retransmit TCP handshake packets using up its resources which ultimately does not seem related to the LOVE payload as it attacks different individuals.

Finally, the decision to treat the LOVE ping payload as isolated was due to by treating it as isolated if any connection to the other ping types appeared during the research they could be noted and investigated but there would be no bias in terms of looking for a connection with the other packets in the investigation if they were all treated as separate instances.

3.2 Noise Storm Detection Script:

The language that was used to write this script was C++ and the PCAP++ package available in it as one of the main requirements for this script were performance as with the size of the merged ICMP packet capture form GreyNoise and intended future captures from Geant and the testbed would be large in scale and require iterating over most packets in the capture. To help confirm this, as initial prototyping for this script was done using Python and Pyshark, the time taken to run through all packets would increase exponentially depending on the size, with the merged packet capture taking upwards of 15 mins to analyse fully. This would however produce easily readable results, but the time spent was too much of a trade-off for these results, so the program was re-written in C++ using PCAP++.

As C++ uses pointer to store its variables such as bytes for a payload or header the program had to have a method to store this variable as a vector and then print this vector out.

```
31
32  vector<char> getFullPayloadAsVector(pcpp::Layer* curLayer)
33  {
34      const uint8_t* payload = curLayer->getLayerPayload();
35      const size_t payload_size = curLayer->getLayerPayloadSize();
36      vector<char> full_payload;
37      for (int i=0; i < payload_size; i++) {
38          full_payload.push_back(payload[i]);
39      }
40      return full_payload;
41  }
42
```

Figure 13: GetFullPayloadAsVector Function

This function takes in the desired layer (in this case the 'HIPERCONTRACER' or to PCAP++ 'unknown' layer) and gets the pointer to its payload and the size of this payload and then creates a blank vector. It then reads each byte of the payload and adds it to this vector and then returns the completed vector which is printed in the main function.

This main function prints details for a single packet from all the source addresses found in the packet capture and then it will print the source and destination addresses of the all the packets that have 'LOVE' embedded in its 'payload' or as we have already discovered has 'LOVE' as its 'Magic Number'.

```
60      while (reader.getNextPacket(rawPacket)) {
61
62      // parse the raw packet into a parsed packet
63          pcpp::Packet parsedPacket(&rawPacket);
64          pcpp::IPv4Layer * ipLayer = parsedPacket.getLayerOfType<pcpp::IPv4Layer>();
65
66          // verify the packet is ICMP
67              if (parsedPacket.isPacketOfType(pcpp::ICMP)) {
68              uint8_t header_type = parsedPacket.getLayerOfType<pcpp::IcmpLayer>()->getIcmpHeader()->type;
69              uint16_t checksum = parsedPacket.getLayerOfType<pcpp::IcmpLayer>()->getIcmpHeader()->checksum;
70
71              if ((srcAddresses.find(ipLayer->getSrcIPv4Address()) == srcAddresses.end()) && (int(header_type) == 8) && (destAddresse
72              // extract source and dest IPs
73                  pcpp::IPv4Layer * ipLayer = parsedPacket.getLayerOfType<pcpp::IPv4Layer>();
74                  pcpp::IPv4Address srcIP = ipLayer->getSrcIPv4Address();
75                  pcpp::IPv4Address destIP = ipLayer->getDstIPv4Address();
76                  pcpp::IcmpLayer * icmp_layer = parsedPacket.getLayerOfType<pcpp::IcmpLayer>();
77                  vector<char> vec = getFullPayloadAsVector(parsedPacket.getFirstLayer());
78                  string payload(vec.begin(), vec.end());
79                  if (payload.find("LOVE") != std::string::npos) {
80                      srcAddresses.insert(srcIP);
81                      destAddresses.insert(destIP);
82                      std::cout
83                      << "Packet type: '" << int(header_type) << "'; (8 is Request, 0 is Reply) \n"
84                      << "Source IP is '" << srcIP << "'; \n"
85                      << "Dest IP is '" << destIP << "'; \n"
86                      << "IP ID: 0x" << std::hex << pcpp::netToHost16(ipLayer->getIPv4Header()->ipId) << "; \n"
87                      << "Checksum: 0x" << std::hex << checksum << "; \n"
88                      << "TTL: " << (int)ipLayer->getIPv4Header()->timeToLive << "; \n"
89                      << "Packet payload is '" << payload << "'; \n"
90                      << "\n" << std::endl;
91                  }
92              }
93          }
94      }
```

Figure 14: Main Function of Detection Script

The final output of this program for the merged packet capture is as follows.

```
Packet type: '8'; (8 is Request, 0 is Reply)
Source IP is '43.152.17.16';
Dest IP is '185.93.173.51';
IP ID: 0x9fc4;
Checksum: 0xf998;
TTL: 33;
Packet payload is ����LOVE�     ^Yj^;


Packet type: '8'; (8 is Request, 0 is Reply)
Source IP is '43.152.0.40';
Dest IP is '185.93.173.51';
IP ID: 0x37f5;
Checksum: 0x1043;
TTL: 34;
Packet payload is C^^LOVE^^Zj^;


Source IP Addresses:
43.152.0.24 43.152.0.26 43.152.0.40 43.152.0.50 43.152.0.51 43.152.17.16 43.152.17.24 43.152.36.45 43.152.37.15 43
.152.37.40 43.152.130.19 43.152.132.14 43.152.132.34 101.33.8.42 101.33.8.156 101.33.8.157 101.33.22.105 101.33.22
.151 101.33.22.152 101.33.24.102 101.33.24.240 148.253.45.4 163.171.175.5 187.19.162.157

Destination IP Addresses:
185.93.173.51
```

Figure 15: Output of Noise Storm Detection Script

Here two individual packets can be seen from ’43.152.17.16’ and ’43.152.0.40’ that both have a destination IP of ‘185.93.173.51’.

3.3 HiPerConTracer Research:

As previously mentioned when discussing the research made by David Schuetz on the GreyNoise packet capture he had discovered that these packets were generated using a service coined HiPerConTracer that is a Ping/Traceroute service but it was a modified version.

Before modifying the program to similarly match the one used to generate the packets seen in the noise storm, it was necessary to gain an understanding of how the service works. For this, three virtual machines were set up on a closed VM network. One of these VMs was a Kali Linux machine with the other ones being Ubuntu 22.04 VMs one with Snort configured with the latest rules snapshot at the time ‘31470’ and the other with the HiPerConTracer service installed on it a compiled from source. The Kali VM was used in conjunction with Hexinject https://hexinject.sourceforge.net to inject a reply packet onto the network to reply to a ICMP request with false information to see if it has any effect on the output of the service.

Closed Network Setup:

IP	Operating System	Tool(s) on Machine
192.168.64.24	Ubuntu 22.04	HiPerConTracer
192.168.64.25	Ubuntu 22.04	Snort
192.168.64.26	Kali Linux	Hexinject & Wireshark

Running the HiPerConTracer service with this command `hipercontracer --source 192.168.64.24 --destination 192.168.64.26 -T --tracerouteinterval 10000 --traceroutеduration 1000 -v`. To target the Kali Linux machine and then using this command with Hexinject `hexinject -s -i eth0 -c 1 -f 'arp' | replace '06 04 00 01'  '06 04 00 02' | hexinject -p -i eth0 `, which will scan the network interface for the term ‘arp’ replace the string found in the found packet and then inject that modified packet onto the network. This changes the Opcode from ‘01’ or request to ‘02’ which is reply so the packet is treated as a reply packet and the rest of the packet remains the same. The result of this can be seen in Wireshark.

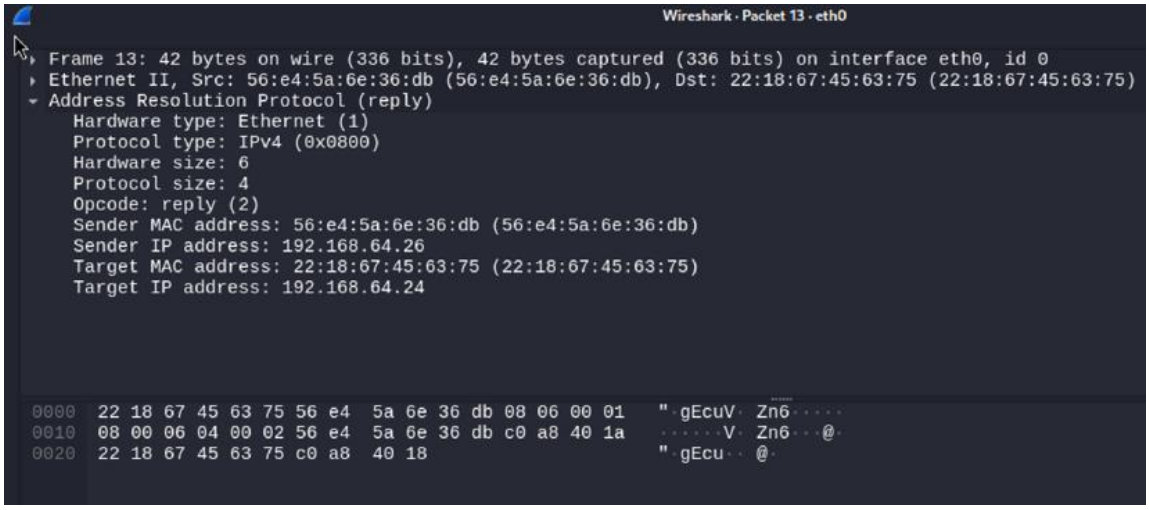

Figure 16: Modified Packet in Wireshark

This has little to no effect on the HiPerConTracer service as it only is concerned if it gets a reply from the destination address and not the content of the reply. As can be seen the outputted csv file from the analysis conducted it only affected the response time from the destination as they were among the fastest to respond.

Class	PacketSize	Checksum	SourcePort	DestinationP	StatusFlags	PathHash	TAB	SendTimestamp	HopNumber	ResponseSiz	Status	TimeSource	Delay.AppSe	Delay.Queuir	Delay.AppRe	RTT.App	RTT.SW	RTT.HW	HopIP
0	44	e8fa	0	0	200	5c9924248e	~	180825bbac4fa6e8	1	0	255	0	-1	-1	-1	4130000	-1	-1	192.168.64.26
0	44	900b	0	0	200	5c9924248e	~	180825be13020c58	1	0	255	0	-1	-1	-1	1591000	-1	-1	192.168.64.26
0	44	752	0	0	200	5c9924248e	~	180825c0822dae00	1	0	255	0	-1	-1	-1	1476000	-1	-1	192.168.64.26
0	44	dc64	0	0	200	5c9924248e	~	180825c31ecabbe8	1	0	255	0	-1	-1	-1	1424000	-1	-1	192.168.64.26
0	44	e030	0	0	200	5c9924248e	~	180825c58fb97220	1	0	255	0	-1	-1	-1	1522000	-1	-1	192.168.64.26
0	44	a20f	0	0	200	5c9924248e	~	180825c811499e80	1	0	255	0	-1	-1	-1	1422000	-1	-1	192.168.64.26
0	44	d346	0	0	200	5c9924248e	~	180825ca75cef678	1	0	255	0	-1	-1	-1	1124000	-1	-1	192.168.64.26
0	44	c6e5	0	0	200	5c9924248e	~	180825cd39048a70	1	0	255	0	-1	-1	-1	1202000	-1	-1	192.168.64.26
0	44	4da	0	0	200	5c9924248e	~	180825cfd7efe588	1	0	255	0	-1	-1	-1	1262000	-1	-1	192.168.64.26
0	44	8b09	0	0	200	5c9924248e	~	180825d262392db0	1	0	255	0	-1	-1	-1	430000	-1	-1	192.168.64.26
0	44	ea41	0	0	200	5c9924248e	~	180825d4d5aac220	1	0	255	0	-1	-1	-1	457000	-1	-1	192.168.64.26
0	44	24a2	0	0	200	5c9924248e	~	180825d75d342ec8	1	0	255	0	-1	-1	-1	510000	-1	-1	192.168.64.26
0	44	3c8c	0	0	200	5c9924248e	~	180825d9d1bc4be8	1	0	255	0	-1	-1	-1	1151000	-1	-1	192.168.64.26
0	44	6065	0	0	200	5c9924248e	~	180825dc3675db10	1	0	255	0	-1	-1	-1	950000	-1	-1	192.168.64.26
0	44	6306	0	0	200	5c9924248e	~	180825defd58cb50	1	0	255	0	-1	-1	-1	1413000	-1	-1	192.168.64.26
0	44	59dc	0	0	200	5c9924248e	~	180825e182020ef0	1	0	255	0	-1	-1	-1	3019000	-1	-1	192.168.64.26
0	44	fe04	0	0	200	5c9924248e	~	180825e43e9e0fa8	1	0	255	0	-1	-1	-1	1486000	-1	-1	192.168.64.26
0	44	fc04	0	0	200	5c9924248e	~	180825e6b38b66d8	1	0	255	0	-1	-1	-1	1178000	-1	-1	192.168.64.26
0	44	e560	0	0	200	5c9924248e	~	180825e97eb908b8	1	0	255	0	-1	-1	-1	745000	-1	-1	192.168.64.26
0	44	62f8	0	0	200	5c9924248e	~	180825ec10f3dad0	1	0	255	0	-1	-1	-1	1586000	-1	-1	192.168.64.26
0	44	f6d4	0	0	200	5c9924248e	~	180825ee9eefb348	1	0	255	0	-1	-1	-1	1408000	-1	-1	192.168.64.26
0	44	3.00E+45	0	0	200	5c9924248e	~	180825f1070638b0	1	0	255	0	-1	-1	-1	833000	-1	-1	192.168.64.26
0	44	1.70E+06	0	0	200	5c9924248e	~	180825f3a7797be8	1	0	255	0	-1	-1	-1	1471000	-1	-1	192.168.64.26
0	44	29c	0	0	200	5c9924248e	~	180825f647a9fd48	1	0	255	0	-1	-1	-1	2105000	-1	-1	192.168.64.26
0	44	f64a	0	0	200	5c9924248e	~	180825f8fb3f6260	1	0	255	0	-1	-1	-1	1535000	-1	-1	192.168.64.26
0	44	d53a	0	0	200	5c9924248e	~	180825fbbec5c1f0	1	0	255	0	-1	-1	-1	1008000	-1	-1	192.168.64.26
0	44	d01	0	0	200	5c9924248e	~	180825fe80f11d90	1	0	255	0	-1	-1	-1	1597000	-1	-1	192.168.64.26

Figure 17: Output of HiPerConTracer csv

It can be seen however from this output the information the service collects such as number of hops, response time, addresses visited and the path taken to reach the destination to name a few.

3.4 Selecting Theory to investigate:

This section will explain the relevant theories regarding the purpose of the Noise Storms and try to disprove some of these theories based on the evidence gathered from the initial research of the capture and based on observations from the research conducted. The popular theories that will be covered are as follows, covert communications/number stations, DDoS attempts, misconfigured routers, command, and control mechanisms, or attempts to create network congestion for traffic manipulation and temporal lensing.

As previously stated, the temporal lensing attack vector which was investigated is a type of Denial-of-Service attack and is relevant for the reasons mentioned in the section on temporal lensing which will be reiterated here.

- Attack used tunnelling to spoof IP address as discovered from Snort alerts when capture was analysed and can be seen in the fact that the packets have the ‘Don’t Fragment’ flag set in the IPv4 header as this is set when using a tunnelling protocol.
- Attack used IP addresses owned by mqcloud.com which was discovered to be the top 2 domain vulnerable for IPIP tunnelling with 881 hosts vulnerable.

- The timing of the packets being sent where 20 packets sent every minute for a week with it being refreshed every day at 12am for all 24 hosts which indicates a concentration of the packets being sent from the hosts.
- GreyNoise reported that the attack had affected the usability of the affected network which would indicate a denial-of-service attack.

Next the rest of theories will be run through in terms of feasibility and relevance to the attack.

Misconfigured Routers: This is unlikely as which will be seen later routing the packets through the network is one of the main reasons this attack worked, is that the attack dealt with mainly ICMP Echo/Reply packets which were sent to their intended destination. On the actor side the distribution of the packets across different hosts and the effort to spoof the IP addresses indicates that deliberate action was taken in setting up the attack and the likelihood of a misconfigured router causing this exact traffic is highly unlikely.

Covert Communication/Number Stations: This theory was mainly born off the fact that the packets which were unknown at the time to be HiPerConTracer packets contained 'LOVE' in ASCII in the payload. This theory hinged on the 'LOVE' was so the packet could be picked out from the network traffic and the following bytes when combined with from multiple packets were form a encoded message similar to the number stations used in World War I with radio waves. According to Wikipedia the traditional definition of 'number stations' are *"a shortwave radio station characterized by broadcasts of formatted numbers, which are believed to be addressed to intelligence officers operating in foreign countries."* This could be attributed to network traffic, but this theory fell apart when David Schuetz discovered that the payload was from the HiPerConTracer header and the subsequent bytes after 'LOVE' were the round number, checksum tweak and timestamp.

Control Mechanisms: This is the same as the above in the fact that now the true nature of the packet's payload is known this makes this theory a lot less plausible as there is no payload besides the HiPerConTracer header in the packets.

Attempts at Network Congestion: This theory works in tandem with the temporal lensing and DoS in the fact that the effect of the attack is multiplied depending on the number of servers involved in the attack and in the attempt of a denial-of-service attack an attacker would want to create network congestion on the destination network to use up the router's resources. This did not seem to be used to manipulate traffic in transit on the network, but the traffic was manipulated to be routed through the same path for each server for reasons that will be explained later.

After all this consideration a hypothesis was formed for the attack vector. ***The attack was a denial-of-service attack using the HiPerConTracer program and IPIP tunnelling to send packets using spoofed IP addresses to the victim in a concentrated manor to conduct a temporal lensing attack.*** Below is a diagram created to help map out the operation of the attack in accordance with this hypothesis.

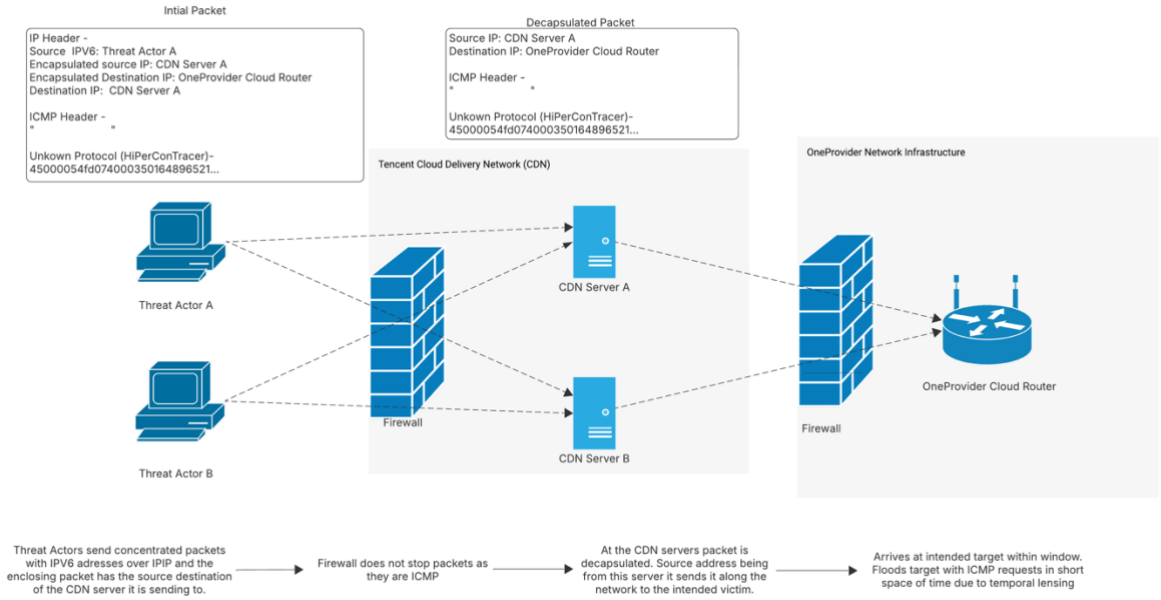

Figure 18: Diagram Explaining the Theorised Attack Vector & Method of Attack

Note: At the time of designing this diagram the consensus was that the packets were sent over 6in4 tunnelling and the actors used IPv6 addresses. This is still a possibility and is theoretically possible, but the IPIP tunnelling seems to be the more likely used here considering the evidence gathered. The operation of the IPIP tunnel would be the same it with just have IPv4 addresses for the actors instead of IPv6.

3.5 Modifying HiPerConTracer Service:

The packet seen in the example below is from the GreyNoise packet capture they had uploaded to GitHub.

```
> Frame 73478: 84 bytes on wire (672 bits), 84 bytes captured (672 bits) on interface unknown, id 0
  Raw packet data
> Internet Protocol Version 4, Src: 43.152.37.15, Dst: 185.93.173.51
v Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0xf8d2 [correct]
    [Checksum Status: Good]
    Identifier (BE): 65535 (0xffff)
    Identifier (LE): 65535 (0xffff)
    Sequence Number (BE): 2 (0x0002)
    Sequence Number (LE): 512 (0x0200)
    [Response frame: 73479]
  v Data (56 bytes)
      Data: 4c4f5645c81f0000aa0a596a9101000000000000000000000000000000000000000000000000000000000000000000000000000000000000
      [Length: 56]

0000  45 00 00 54 58 2e 40 00  35 01 36 43 2b 98 25 0f   E··TX.@· 5·6C+·%·
0010  b9 5d ad 33 08 00 f8 d2  ff ff 00 02 4c 4f 56 45   ·]·3···· ····LOVE
0020  c8 1f 00 00 aa 0a 59 6a  91 01 00 00 00 00 00 00   ·····Yj ········
0030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ········ ········
0050  00 00 00 00                                       ····
```

Figure 19: Packet from GreyNoise Capture

The initial source of confusion of program used to generate the packets was that Wireshark would display the “traceservice” header as a payload for the ICMP packet. Attempting to access this payload would result in a null as it does not exist. This is in contrast to a packet generated by the unmodified HiPerConTracer service which produces this output.

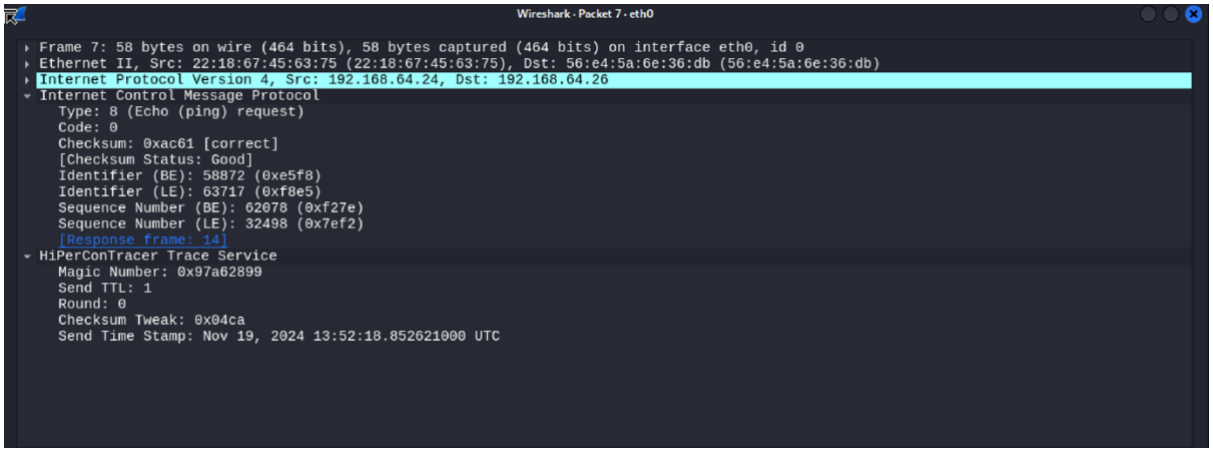

Figure 20: Regular Packet from Non-Modified HiPerConTracer Service

From this disparity it was theorised that the version of the program used was modified in ways to mask its usage and true purpose. Referring again to David Schuetz's findings he discovered that the send timestamp in the GreyNoise packets were in milliseconds, stored in little-endian format and where in UNIX epoch time.

Looking at the source code for the program, it can be seen below that the timestamp is set in "src/traceserviceheader.h" as seen below.

```
}
inline void sendTimeStamp(const ResultTimePoint& timeStamp) {
    // For HiPerConTracer packets: time stamp is microseconds since 1976-09-29.
    sendTimeStamp(std::chrono::duration_cast<std::chrono::milliseconds>(
                      timeStamp - HiPerConTracerEpoch).count());
}
```

Figure 21: Send Time-stamp being set in src/traceserviceheader.h

This can be changed from microseconds to milliseconds, like what is seen. Then if this modified service is ran it produces the following output.

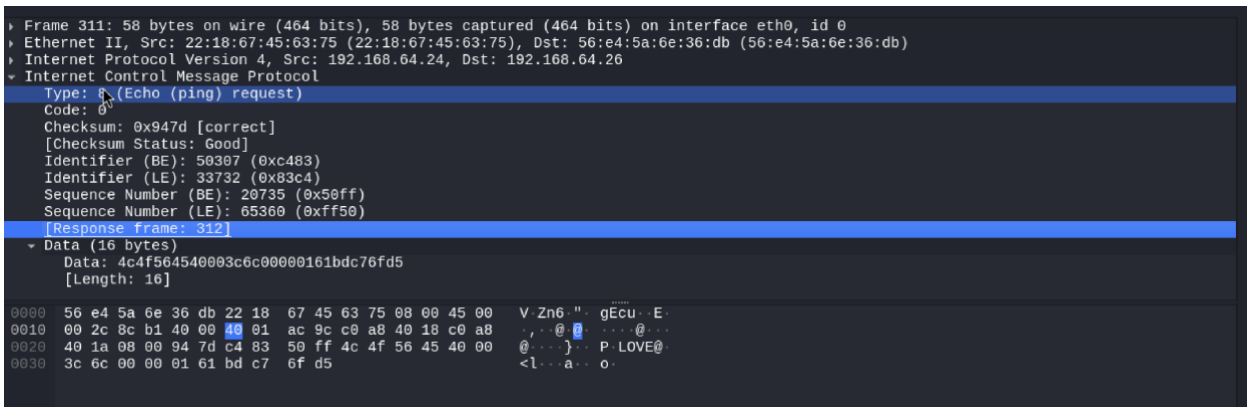

Figure 22: Modified HiPerConTracer Service Output after Modified Time-stamp

This leads to Wireshark not recognising the header, similar to the one seen in the GreyNoise packet capture. The main difference of this packet compared to the one seen in the original capture is the lack of padding at the end of the packet which was used to mask the programs usage further but is unnecessary for this testing as it contains no information.

The finally part of the modification of the HiPerConTracer program was the editing of the Checksum Tweak for the program. As David Schuetz discovered *"The "Checksum Tweak" is, according to what I've been reading, an arbitrary 16-bit field that gets automatically adjusted to ensure that all packets in the same "session" have the*

same ICMP checksum. This should fool network gear into thinking they're the same packet, so they get routed over the same path, for consistency in timing." This was used to ensure that all packets were routed through the same path over the network ensuring they arrive at the host at the same time.

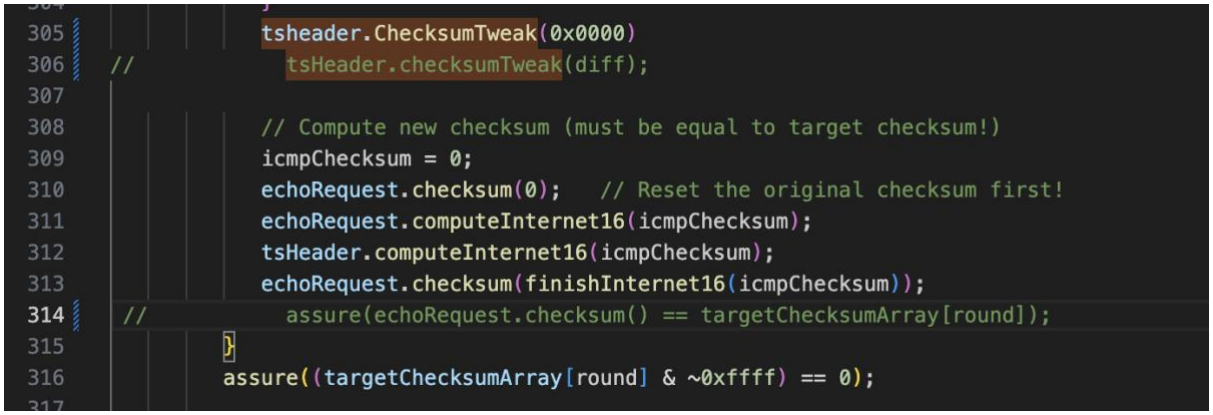

Figure 23: Checsum Tweak being set in src/iomodule-icmp.cc

As can be seen above this was achieved by changing the Checksum tweak and removing the check for it being the same as the checksum round. The finished modified-hipercontracer program is available on the projects GitHub at https://github.com/anthonypower0923/FYP/tree/main/modified-hipercontracer.

Results

4.1 Software Produced:

During the course of this project not much software was produced due to the nature of a research project, however two main pieces of software were produced, these being a detection script for evidence of the LOVE noise storm with a script written in C++. This would output the structure of the HiPerConTracer header for each source address in the packet capture as well as at the bottom a list of source IP addresses and destination IP addresses for the capture.

The other main piece of software produced was a modified version of Thomas Dreibholz HiPerConTracer program with it being modified to match as closely the version seen in the packet capture provided by GreyNoise. This version of the program is not using the same version HiPerConTracer as was used in the capture and is instead the most updated version available as of 19th of April 2025 and modified to match the conditions of the one used in the capture. The version of the one used in the capture is suspected to be version 1.x of the HiPerConTracer program based on this comment from the traceserviceheader.h.

```
127     }
128     inline void sendTimeStamp(const ResultTimePoint& timeStamp) {
129         // For HiPerConTracer packets: time stamp is nanoseconds since 1976-09-29.
130         // NOTE: HiPerConTracer 1.x used microseconds here. Simple heuristic:
131         // < 2*32*1e6: HiPerConTracer 1.x in microseconds
132         // Otherwise:  HiPerConTracer 2.x in nanoseconds
133         sendTimeStamp(std::chrono::duration_cast<std::chrono::miliseconds>(
134                                 timeStamp - HiPerConTracerEpoch).count());
135     }
136
```

Figure 24: Comment stating the different units used of versions of HiPerConTracer

This line comment signifies that HiPerConTracer 1.x used microseconds for the timestamp which is the same as the timestamp David Schuetz was able to extrapolate from one of the packets in the capture which matched the sniff timestamp in the packets frame. The other reason for this being the case is that version 2.0 of the HiPerConTracer program only came out on the December 20 2024, four months after the date of the packet capture.

In addition to the these artefacts are two python scripts being mqtt_to_influx.py which creates a MQTT client and listens on the specified topic and parses the output into an InfluxDB that is either local or remote. The other script is an unfinished packet parser for parsing the output of the packet capture and reversing the time stomping of the timestamp reverting it back to its original timestamp and parsing out the HiPerConTracer header and placed in a separate sheet for each packet capture. This is unfinished because it was not created to handle the other packets in the capture and currently produces some false positives. It was decided to leave it in this unfinished state and to focus on further research considering the time needed to complete this program and test it. It maybe revisited in the future.

4.2 Initial Investigation:

The initial investigation of this phenomenon was mainly focused on the purpose and usage of the HiPerConTracer program used and if there was anything that it does that was not intended by the author that could be exploited. To test this program a simple testbed was setup between three machines as mentioned in the HiPerConTracer research section and the results of the Snort alert logs did not flag any major issues with the program. Snort flagged mainly the an ‘arp_spoof’ attempt in its logs below.

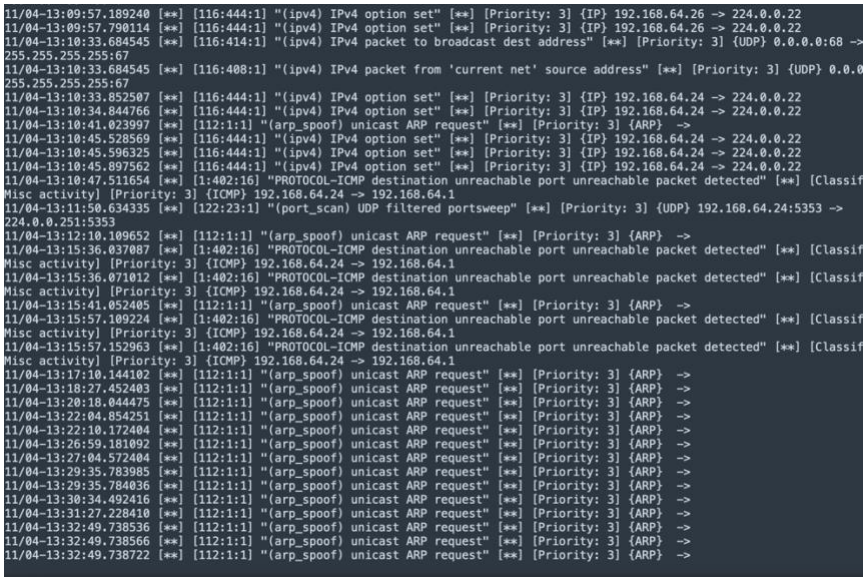

Figure 25: Snort Logs For Normal Run of HiPerConTracer Service

The alert that is of interest here is “unicast ARP request” or ARP cache poisoning attack which “is a technique by which an attacker sends (spoofed) Address Resolution Protocol (ARP) messages onto a local area network” according to Wikipedia. https://en.wikipedia.org/wiki/ARP_spoofing

If we look at the Snort ARP Spoof Inspector Rules we can see that this is caused by a Unicast ARP request as it matches the SID 112:1 in the table below from Cisco.

GID:SID	Rule Message
112:1	unicast ARP request
112:2	ethernet/ARP mismatch request for source
112:3	ethernet/ARP mismatch request for destination
112:4	attempted ARP cache overwrite attack

Figure 26: SID To Rule Table for Snort Alerts

This is caused due to the usage of the traceroute option on the program and is caused because the service is looking for addresses on the network and when this is ran in solely ‘ping’ mode most of these alerts stop.

4.3 Testing Formed Hypothesis:

This section is relation to testing the hypothesis formed in the previous section. This hypothesis was as followed “***The attack was a denial-of-service attack using the HiPerConTracer program and IPIP tunnelling to send packets using spoofed IP addresses to the victim in a concentrated manor to conduct a temporal lensing attack.”***

As this attack would be conducted over a long period of time it was preferable to procure cloud resources so the attack could be left running and collecting data. OpenShift was used because due to previous connections with the Walton Institute it could be used without accruing fees and as one of the requirements for the testbed would be an isolated network and initially creating containers in a namespace on OpenShift would satisfy these criteria, however due to the locked down nature of OpenShift it was not possible to use this setup as network interfaces needed to be created such as IPIP tunnel interfaces to conduct this experiment. This was not possible considering the restrictions imposed by OpenShift in relation to containers, so the decision was made to setup the testbed using Docker on a created VM in OpenShift and place this VM of a publicly accessible network as it at the time due to the cyberattack on the college the main college network was down. The VM created was an ‘Ubuntu 24.04.2 LTS’ VM.

As the VM was publicly accessible the first thing done on it was to setup default firewall rules on the VM to deny incoming and allow outbound and then set up the relevant rules to allow access to the VM such as port 22 access for the network and public access to ports 1883 and 3000 for MQTT and Graphana respectively as can be seen in the below image.

```
root@ubuntu-tomato-marmoset-42:~# ufw status verbose
Status: active
Logging: on (low)
Default: deny (incoming), allow (outgoing), deny (routed)
New profiles: skip

To                         Action      From
--                         ------      ----
22/tcp                     ALLOW IN    86.46 [redacted]
1883                       ALLOW IN    Anywhere
3000/tcp                   ALLOW IN    Anywhere
1883 (v6)                  ALLOW IN    Anywhere (v6)
3000/tcp (v6)              ALLOW IN    Anywhere (v6)
```

Figure 27: UFW Rules for OpenShift VM

With this setup it was now safe to proceed with the experiment. Docker engine was installed on the machine following the official Docker documentation https://docs.docker.com/engine/install/ubuntu/. Docker engine was installed instead of Docker desktop because Docker desktop creates a VM to work off of and as of current uses linux-kit builds for their kernels. This is an issue because the operation of this experiment requires the ipip module to be enabled which requires a specific version of that module for the kernel and as of current linux-kit builds are behind in adding kernel modules to download. This was the second issue with using just containers on the OpenShift platform as they also built using linux-kit kernels for the host machine.

This however is not an issue with Docker engine as it uses the kernel of the host machine and all created containers share the same host kernel in this case '6.8.0-57-generic'. As the initial attempts to setup the testbed involved containers Docker files were created to create the images for these containers and the files where pushed to a GitHub repository as well as the images themselves pushed to Docker hub for later retrieval.

The next section requires a little knowledge on how Docker handles kernels and networking. When a Docker container is created without a specified network it creates it on a the default bridge network. If the container is then started in privileged mode it can create network interfaces on the container. If that container is then restarted it will remove any newly created network interfaces. As for kernels in default operation the host kernel is differed any kernel actions necessary and the kernel modules are hidden to the container. To enable the container to be able to access and modify kernel modules a flag has to be set during container creation or running '-v /lib/modules:/lib/modules'. This flag will copy the kernel modules of the host machine into the container so they can be accessed and enabled/disabled using modprobe from the kmod package.

With this knowledge it is now known that to once the containers are created they need to be ran continuously to using a command so they can be exec'd into to create the necessary configurations and then the HiPerConTracer program can be ran on it which require the network configuration beforehand to work in the intended way. Firstly two bridge networks were setup using the bridge network driver these being control-net and test-net. Control-net would be the modified HiPerConTracer service without the change to the checksum Tweak so in theory it should take different paths to the host for each "session" which according to the research conducted seems to be the round or burst of the packets sent in this case being 20 packets per minute. This should make it so collisions between the two actors payloads should be fewer, then if the Checksum Tweak is not set ensuring consistent routing through the network to the intended host.The setup is similar to the proposed attack diagram shown earlier.

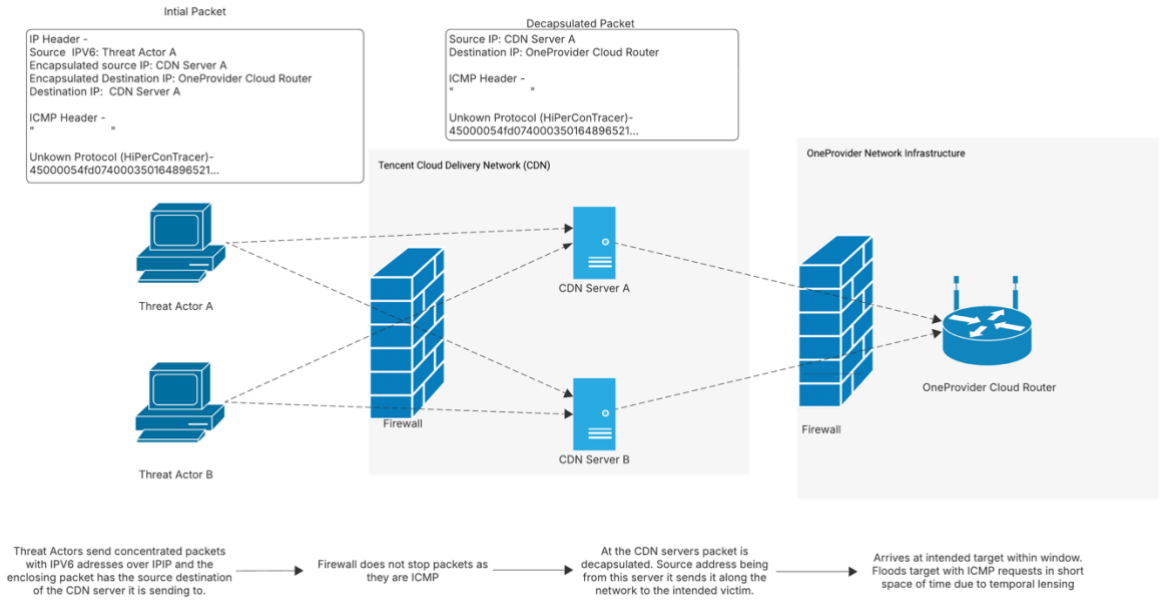

Figure 28: Diagram Explaining the Theorised Attack Vector & Method of Attack

This will have two actors and cdn-server's (containers) attacking a single destination container. The network hosts and configuration for both created networks is as follows.

Control-net:

- Subnet: 172.18.0.0/16
- Gateway: 172.18.0.1
- victim-1: 172.18.0.6
- cdn-server-1: 172.18.0.5
- cdn-server-2: 172.18.0.4
- actor-1: 172.18.0.3
- actor-2: 172.18.0.2

Test-net:

- Subnet: 172.19.0.0/16
- Gateway: 172.19.0.1
- cdn-server-3: 172.19.0.4
- cdn-server-4: 172.19.0.5
- victim-2: 172.19.0.6
- actor-3: 172.19.0.3
- actor-4: 172.19.0.2

The two setups are identical in configuration with the only difference being the modification to the HiPerConTracer service. To allow the actor containers to setup the IPIP tunnel they were all started using the following command, `docker run --privileged -v /lib/modules:/lib/modules --network=control-net -itd --name=vactor-1 actor tail -f /dev/null` with the name of the container and network it is one being changed if necessary. The container is started with tail -f /dev/null to keep it continuously running while configuring the network interfaces.

Looking into one of the actor containers and having a look at the network interfaces it can be seen that the interface ipip0 is setup in a similar fashion to previously discussed in the tunnelling section.

```
# ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if12: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ba:43:18:93:74:16 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 172.18.0.3/16 brd 172.18.255.255 scope global eth0
       valid_lft forever preferred_lft forever
3: tunl0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
6: ipip0@NONE: <POINTOPOINT,NOARP,UP,LOWER_UP> mtu 1480 qdisc noqueue state UNKNOWN group default qlen 1000
    link/ipip 172.18.0.3 peer 172.18.0.5
    inet 172.18.0.5/16 scope global ipip0
       valid_lft forever preferred_lft forever
    inet6 fe80::5efe:ac12:3/64 scope link
       valid_lft forever preferred_lft forever
#
```

Figure 29: Configures IPIP tunnel on Actor container

The main difference is that the private IP of the tunnel is set the same as the peer or remote IP on the network. This is done to spoof the IP address of the cdn-server container in this case cdn-server-1. This effect of this will be seen when the HiPerConTracer service is ran. This is necessary as the HiPerConTracer service requires the ability to create a UDP socket on the source destination s specified in the command's output and to spoof the IP address the private IP needs to be set to the remote address so it creates the UDP socket on the container which it maps from the tunnels address.

With all this configured and the modified HiPerConTracer service built and in the folder for the service if the program is ran using the following command `./src/hipercontracer --source 172.18.0.5 --destination 172.18.0.6 --ping --pinginterval 60000 --pingburst 20 --pingpacketsize 84` it will start the service. To explain the fields on the command the –source is the remote IP which because of the tunnel will be mapped to the tunnel interface on the container and the –destination is the remote IP of the victim-1. Then –ping chooses the to send pings and the interval is set to 60000 milliseconds or 1 minute and the ping burst to 20 per burst and finally the packet size is set to 84 to match the ones seen in the capture. When this command is ran the output will be of a timeout on the actor container end but the packets will be sent to the destination. The timeout is in fact for the response packets which the service expects to receive to complete the timing which because of the spoofed IP address are sent to the actual container of the spoofed IP address this being cdn-server-1.

The image below shows this in operation with three terminals in order of left to right of actor-1, cdn-server-1 and victim-1. Cdn-server-1 and victim-1 are running tshark to capture network traffic from eth0 with the filer of -Y icmp for only ICMP traffic.

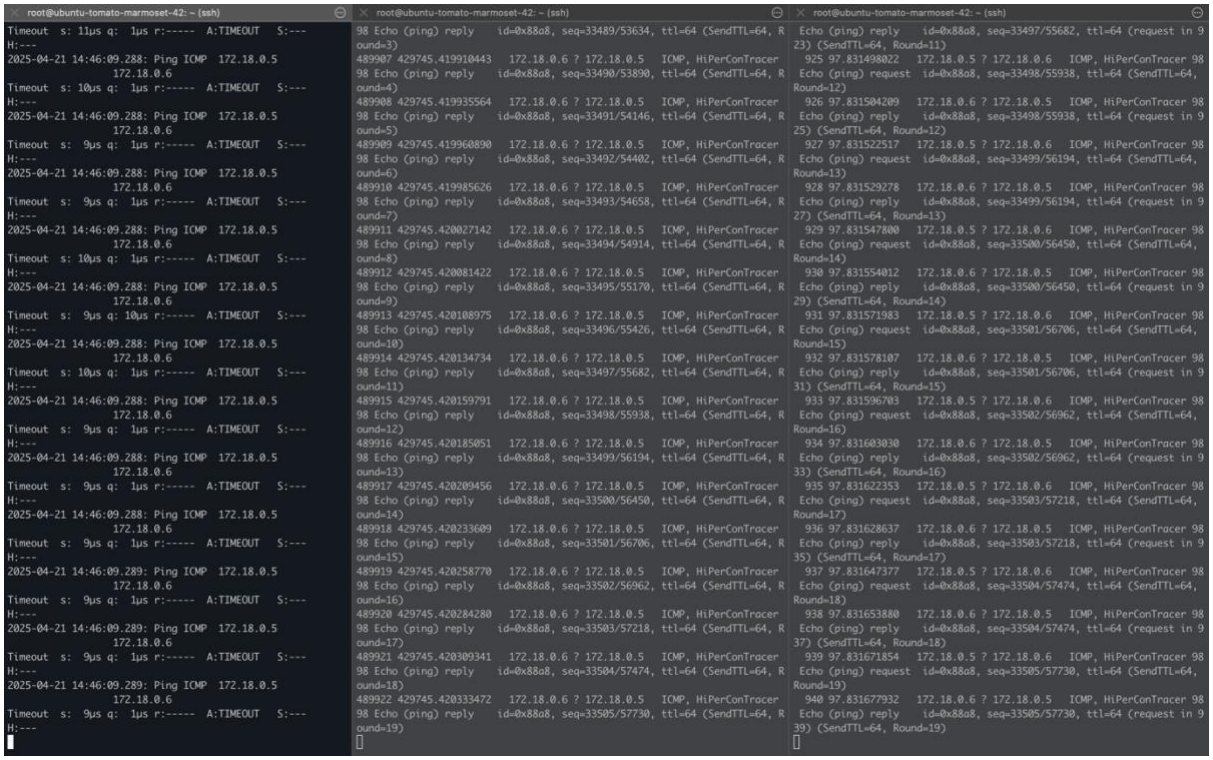

Figure 30: Test Run of Attack. Containers from left to right, Actor-1, CDN-Server-1, Victim-1

As can be seen from the image the HiPerConTracer service sends the packet burst to victim-1 which receives the request and then due to the spoofed IP address incorrectly sends the reply packet to actual container of that address instead of back to the initial sender. This causes the HiPerConTracer service to timeout on the packet sending as it did not receive a reply from the victim container. **Note:** *If the remote IP of the tunnel is set to the private IP is set to the remote IP it will create a loop for the victim container consistently sending it to itself which does throttle the network. This was discovered when trying to setup the tunnel configuration and was of interest to note.*

The victim containers on both testbeds where tunning a netmonitor.sh bash script on them to collect the information on the TX or transmission of data and the RX or receiving of data and then the total which these values being in packets per second. This script was taken from https://gist.github.com/talawahtech/de78601f1201d9586ac19fff420024b8 and modified to push these via MQTT to be received by the host machines broker, which is why port 1883 was opened on the firewall rules.

This data was then parsed using the aforementioned mqtt_to_influx.py script to add to an InfluxDB on the host machine and then to be used a data source for a Graphana dashboard which can be seen below and can the dashboard can be accessed by this link http://87.44.27.48:3000/ and login in with user 'guest' and password 'guest'.

Figure 31: Graphana Dashboard after 6 hours of Running Testbed

This was the output of the script after 6 hours of running and despite some peaks which are due to timing issues the Test network is behind by at least a packet a second in terms of packets/second for all three topics which could be due to the fact that the routing of the packets meant that collisions where more likely with them being sent the route every time. The decision to use this script for measuring the network was due to it's relative quick completion as Ookla's speedtest-cli script https://www.speedtest.net/apps/cli was considered but due to the time needed for completion it would finish outside of the window needed to see the effect of the collision. This would be the same as any other bandwidth script that tests internet speed as they would not run instantly. The chosen script however just read values from the kernel itself so had no issue with running on time which is why it was chosen.

As both server where set up identically and the only difference was the change in the modified HiPerConTracer script it seems that it does have an effect on the packets routing and is more likely to cause collisions that lead to decreased network performance.

Discussion

5.1 Hypothesis:

In relation to the hypothesis "***The attack was a denial-of-service attack using the HiPerConTracer program and IPIP tunnelling to send packets using spoofed IP addresses to the victim in a concentrated manor to conduct a temporal lensing attack."*** It can be seen from the results from the previous section that the there was a noticeable difference on average between the control and test testbeds results. If we consider than this performance scales with the amount of actor servers, then it could be possible to prove the hypothesis and if more manipulation of protocols for the traffic to ensure they arrive within the same timeframe as can be seen from the minimal collisions that did occur it would have a greater effect on the network and should greatly increase the attacks effect. In relation to the actual attack conducted it does appear to be a denial-of-service attack using tunnelling for the spoofed IP addresses and as the previous research has shown it is possible using timing to conduct a temporal lensing attack on the network. This shows that the hypothesis is plausible but should not be taken as confirmed until further testing.

5.2 What Could Have Been Done Differently:

In terms of what could be changed in terms of the investigations carried out during the course of this investigation Is two major changes which would be to scale up the attack vector by adding more containers and in terms of the output of the results a tool that was constantly monitoring network bandwidth would work better for showing and comparing results instead of reading it directly from the kernel as GreyNoise's report on the effect of the attack was it caused network disruption and if the target of the attack is the router as theorized the this information would be more pertinent to that. The results would still be pushed via mqtt and then displayed on the dashboard. The other thing that would be changed is just in relation to configuration would be to setup the running of the containers using a docker compose script so it could be easily migrated if necessary but was unnecessary considering the scale of the testbed.

5.3 Further Steps:

As alluded previously one of the future steps to take regrading this investigation would be to scale up the testbeds to the same or half of the number used in the attack to see if the effect of temporal lensing scales with the number of actors and try to manipulate the timing of the traffic to concentrate its arrival at the victim within a shorter timeframe to supposedly amplify the attack's effectiveness according to the previously quoted research paper. Another avenue to investigate considering it is now known is the possibility of the TCP traffic being from the HiPerConTracer service as looking at development branches for the program then was work done on adding both a TCP and UDP features to the program. Finally with the knowledge that has been accrued on the operation of and potential attack vector is there any of the network traffic in the capture that was sent to the host from the other ping payloads that was sent within the same timeframe as one of the packet bursts and what would be the compounded effect of those payloads on that network.

5.4 Conclusion:

Though this research has narrowed down the possible attack vectors of the attack conducted and has formed a hypothesis that further testing has supported this does not mean the hypothesis is true. The capture given was only a sample of a single attack using this method and as GreyNoise have said there has been multiple of these conducted since August 2024. This has just been a snippet that they have provided for researchers to investigate and possibly further packet captures containing this attack might lead to discovering the true purpose of this attack.

However, from research of this packet capture and trying to recreate the traffic seen in it has helped gain further knowledge of one of the programs used in the creation of these packets, Snort had uncovered that the packets where spoofed IP addresses which was previously suspected and uncovered due to the 'Don't fragment' flag on the IP header that they were probably tunnelled using 6in4 or IPIP or another tunnelling protocol. From conducting research on the formed hypothesis, it was possible to recreate a small-scale version of the hypothesised attack and observe the effects of this attack. One thing that is known is that the investigation into these Noise Storms is not finished and needs further research until its purpose can be definitively defined.

References

GreyNoise Intelligence, GitHub (September 2024) *"2024-09-noise-storms"*, Available *at:* https://github.com/GreyNoise-Intelligence/2024-09-noise-storms (First Accessed on: 20th September 2024)

GreyNoise Intelligence (March 2023) *"GreyNoise Noise Storms"*, Available at: https://observablehq.com/@greynoise/noise-storms (First Accessed on: 4th January 2025)

Schuetz, D (September 2024) *"Ping Storms at GreyNoise", Available* at: https://darthnull.org/noisestorms/ (First Accessed on: 28th September 2024)

Dreibholz, T (April 2025) *"HiPerConTracer: High-Performance Connectivity Tracer",* Available at: https://www.nntb.no/~dreibh/hipercontracer/ (First Accessed on: 28th September 2024)

Dreibh, GitHub (April 2025) *"hipercontracer",* Available at: https://github.com/dreibh/hipercontracer (First Accessed on: 28th September 2024)

Dreibh, GitHub (April 2025) *"The HiPerConTracer Reverse Tunnel Tool",* Available at: https://github.com/dreibh/hipercontracer?tab=readme-ov-file#the-hipercontracer-reverse-tunnel-tool (First Accessed on: 8th January 2025)

Morell, C (October 2024) *"A Detour In Time: Meta-Analysis of GreyNoise Ping Storms" ,* Available at: https://medium.com/@colin.l.morrell/a-detour-in-time-meta-analysis-of-greynoise-ping-storms-6b9262d0daea (First Accessed on: 2nd December 2024)

PcapPlusPlus (2024) *"Welcome to PcapPlusPlus!",* Available at: https://pcapplusplus.github.io/ (First Accessed on: 20th October 2024)

PcapPlusPlus (2024) *"API Reference",* Available at: https://pcapplusplus.github.io/docs/api (First Accessed on: 20th October 2024)

Schuetz, D (October 2024) *"Update on August Ping Storms", Available* at: https://darthnull.org/noise-storm-update/ (First Accessed on: 28th September 2024)

RedHat, Developers (May 2024) *"An introduction to Linux virtual interfaces: Tunnels",* Available at: https://developers.redhat.com/blog/2019/05/17/an-introduction-to-linux-virtual-interfaces-tunnels#ipip_tunnel (First Accessed on: 2nd February 2025)

Vanhoef , M , Beitis, A (2025) *"Haunted by Legacy: Discovering and Exploiting Vulnerable Tunnelling Hosts",* Available at: https://papers.mathyvanhoef.com/usenix2025-tunnels.pdf (First Accessed on: 3rd March 2025)

SourceForge, (2024) *"HexInject: News",* Available at: https://hexinject.sourceforge.net/ (First Accessed on: 22nd October 2024)

Docker Documentation (2024) *"Install Docker Engine on Ubuntu",* Available at: https://docs.docker.com/engine/install/ubuntu/ (First Accessed on: 3rd March 2025)

GitHub Gist (2025) *"talawahtech/netmonitor.sh",* Available at: https://gist.github.com/talawahtech/de78601f1201d9586ac19fff420024b8 (First Accessed on: 20th March 2025)

Ookla (2024) *"SPEEDTEST® CLI",* Available at: https://www.speedtest.net/apps/cli (First Accessed on: 8th April 2025)

Bibliography

Vanhofm, GitHub (March 2025) *"tunneltester",* Available at: *https://github.com/vanhoefm/tunneltester?tab=readme-ov-file* (First Accessed on: 25th April 2025)

Suedbroecker', T (2024) *"Open the door for root users in Red Hat OpenShift (example Deployment) ¶",* Available at: *https://suedbroecker.net/2021/12/09/open-the-door-for-root-users-in-red-hat-openshift¶/* (First Accessed on: 23rd March 2024)

Snort Documentation (2024) *"Snort 3 Installation",* Available at: *https://docs.snort.org/start/installation* (First Accessed on: 20th October 2024)

RedHat Documentation (2024) *"OpenShift Container Platform 4.18",* Available at: *https://docs.redhat.com/en/documentation/openshift_container_platform/4.18* (First Accessed on: 23rd March 2025)