ANEMONA Code Examples

Henrique Denes Hilgenberg Fernandes
Brazilian Institute of Information in Science and Technology
Dept. of Operation, Network and Security of Information
Brasília DF Brazil
denes@ibict.br

Elias Procópio Duarte Júnior
Federal University of Paraná
Dept. of Informatics
Curitiba PR Brazil
elias@ufpr.br

Martin Alejandro Musicante
Federal University of Rio Grande do Norte
Dept. of Informatics
Natal RN Brazil
mam@dimap.ufrn.br

Technical Report # RT001/2007, Federal University of Paraná, Dept. Informatics, http://www.inf.ufpr.br/info/techrep/

Abstract

In this report we present three case studies in which ANEMONA [1] is used in actual conditions of network management. These case studies show ANEMONA programs that notify an administrator whenever a suspicious behavior is detected (an attack) or when the number of transmitted and received IP packages indicates link overload.

1 Attack Detection

This case study uses ANEMONA programs to detect a probable Denial of Service (DoS) attack, which causes the unavailability of services in the attacked hosts. ICMP Flooding and TCP Syn Flooding [2] Attacks were tried, both based on flood. We successfully detected both attacks through ANEMONA scripts.

1.1 ICMP Flooding Attack Detection

This attack consists in sending a huge lot of ICMP protocol echo requests [3] to a given host. The attacked computer will try to respond all of these requests, consuming system resources and causing the unavailability of some services.

The management object icmp.InEchos contains the number of received ICMP echo requests and its type is Counter32. With this object, we designed an ANEMONA application, which informs an administrator that an attack is happening.

In order to build this application, we had been monitoring the above mentioned object's behavior in several situations. Based on these data, we chose critical values which were used in the current program. To try it, we used a Linux computer with NET-SNMP agent, Event and Expression MIB prototypes and an ANEMONA translator. We also used a computer with Linux and another one with Windows NT. The mentioned computers were connected to an Ethernet Network through 10 Mbps network adapters.

Since the object icmp.icmpInEchos is a counter, it needed to be sampled as delta. These samples, all of them delta with 6 seconds of interval, were achieved through the following set of tools: Event and Expression MIBs and ANEMONA translator.

While the network was normally working, with no ICMP echo requests, the icmp.icmpInEchos value was kept on zero. When the first host begun to ping the server, the delta value of icmp.icmpInEchos became 6 and, when there were two hosts pinging the server, the same value became 12.

Based on the statements above and ICMP protocol definitions, we can say that each ping command performed will generate an ICMP request per second.

If you type a ping command with the parameter '-f', the destination host will be flooded with requests as fast as they are responded. Using 'ping –f', we collected, in intervals of 1 minute, the delta values shown in the table of figure 1.

Elapsed time of attack	icmp.icmpInEchos' delta (6 s interval)
0 min.	3089
1 min.	4034
2 min.	4107
3 min.	4113
4 min.	4126
5 min.	4130
6 min.	4119
7 min.	4126

Figure 1: ICMP Flooding attack

Based on the results above, we chose 1000 as critical value to icmp.icmpInEchos delta, using an interval of 6 seconds between samples, value which should stands for 166 simultaneous pings. We also noticed that during this attack the delta values of icmp.icmpInEchos were always greater than 1000.

The following ANEMONA program implements this application:

```
watch: victim.cce.ufpr.br using private
icmp.icmpInEchos.0 is Counter32: delta
begin
        when (icmp.icmpInEchos.0 > 1000)
        do
                notify manger.cce.ufpr.br private icmp.icmpInEchos.0
        end
end
```

In this program, the host victim.cce.ufpr.br is monitored with the object icmp.icmpInEchos from its MIB sampled as delta. When this object's value will be grater than 1000, a notification with icmp.icmpInEchos.0 will be sent to manager.cce.ufpr.br[1].

The ANEMONA translator generated the set of actions shown straightforward, where we can notice 14 SNMP commands needed to program this application plus additional operations to found empty entries in MIB tables.

```
snmpset victim.cce.ufpr.br private 98.1.1.2.1.2.1 o icmp.icmpInEchos.0
snmpset victim.cce.ufpr.br private 98.1.1.2.1.3.1 i deltaValue
snmpset victim.cce.ufpr.br private 98.1.1.1.1.2.1 s "$1 > 1000"
snmpset victim.cce.ufpr.br private 98.1.1.1.1.3.1 i unsigned32
snmpset victim.cce.ufpr.br private 99.1.1.1.1.3.1 i boolean
snmpset victim.cce.ufpr.br private 99.1.1.1.1.4.1 i absoluteValue
snmpset victim.cce.ufpr.br private 99.1.1.1.1.5.1 o 98.1.2.1.1.3.1
snmpset victim.cce.ufpr.br private 99.1.1.1.1.8.1 i 1
snmpset victim.cce.ufpr.br private 99.1.1.1.1.9.1 i unequal
snmpset victim.cce.ufpr.br private 99.1.1.1.1.10.1 i 0
snmpset victim.cce.ufpr.br private 99.1.2.1.1.3.1 i notification
snmpset victim.cce.ufpr.br private 99.1.2.1.1.6.1 o icmp.icmpInEchos.0
snmpset victim.cce.ufpr.br private 99.1.2.1.1.7.1 i true
snmpset victim.cce.ufpr.br private 99.1.1.1.1.6.1 i true
```

During the normal operation of this host, with no attacks being performed, this program was translated and, after that, attacks were produced typing 'ping –f' in another host. After an average of 4.3 seconds, the host manager.cce.ufpr.br got a notification.

1.2 TCP Syn Flooding Attack detection

A TCP Syn Flooding Attack causes a failure in the engine responsible to establish connections in TCP protocol. To establish a connection, it is necessary to perform a three-way-handshake [8], which begins when a client sends to the server a TCP segment with a flag Syn set in its header. Normally the server returns a segment with the flags Syn and Ack set to the client's address found in IP header. By the end, the client sends an Ack to the server and a connection is established.

A TCP Syn Flooding Attack floods a given server with TCP segments which contain Syn flags set in their headers, but with their source addresses doctored in their IP headers. These source addresses correspond to an unreachable host. So, when a server gets these segments, it tries to return segments with flags Syn and Ack set to the source addresses written in the IP headers from the former segments. The segments returned by the server will not be responded until a timeout occur, which does not allow the TCP to complete the three-way-handshake.

[1] During the tests, manager.cce.ufpr.br was just a canonical name to victim.cce.ufpr.br.

During an attack, the attacker host sends several requests to one or more TCP ports in the attacked server. The number of requests shall be enough to flood request queues, causing unavailability in services running in the attacked ports.

To detect an attack like this, the object tcp.tcpAttemptFails counts how many times the TCP state's machine switches from states SYN-SENT or SYN-RCVD to state CLOSED, plus how many times it switches from SYN-RCVD to LISTEN.

A working server, waiting for connections, stays in LISTEN until it gets a Syn request. When it gets this request, which will have its source address doctored to an unreachable one during an attack, the server will send to the source address a segment with flags Syn and Ack set and will switch to SYN-RCVD, where it will wait for an ack. Since there will not this confirmation, the server will stay on SYN-RCVD until it reaches a timeout, when switches to CLOSED, incrementing tcp.tcpAttemptFails.

In this case study, we used the program Neptune [2], which produces spoofing, generating doctored segments, with source and destination addresses and port given by the user.

The equipment used was a computer with Linux, NET-SNMP agent, Event and Expression MIBs, ANEMONA translator and Apache, WU-FTP and telnetd servers plus another computer with Linux, telnet and Neptune. Both were connected to an Ethernet Network through 10 Mbps interfaces.

As the object tcp.tcpAttemptFails is a counter, it was sampled as delta, with 6 seconds interval, using Expression MIB and ANEMONA translator.

With the net working normally, we connected FTP, HTTP and telnet services using the monitored host as client and also as server. The value of tcp.tcpAttempt Fails.0 was kept in zero. We concluded that these errors are not frequent.

We ran the first attack against the service at port 23 (telnet), using 5000 segments and, using 6 seconds intervals, we collected the values from figure 2. After 30 seconds, the delta value of tcp.tcpAttemptFails was stable in 300. Ending this attack, the absolute value of tcp.tcpAttemptFails.0 was 4887.

Elapsed time of attack	tcp.tcpAttemptFails' delta (6 s interval)
0 s	0
6 s	0
12 s	170
18 s	300
24 s	301
30 s	300

Figure 2: TCP Syn Flooding Attack

Based on these test results, we chose 25 as critical value to tcp.tcpAtemptFails delta, using 6 seconds of interval between samples, which stands for four failures per second.

The following ANEMONA program implements this application[2]:

[2] During the tests, manager.cce.ufpr.br was just a canonical name to victim.cce.ufpr.br.

```
watch: victim.cce.ufpr.br using private
tcp.tcpAttemptFails.0 is Counter32: delta
begin
        when tcp.tcpAttemptFails.0 > 25
        do
                notify manager.cce.ufpr.br private tcp.tcpAttempFails.0
        end
end
```

In this program, victim.cce.ufpr.br is monitored, having tcp.tcpAttemptFails from its MIB sampled as delta. When delta value from this object will be greater than 25, a notification with tcp.tcpAttemptFails.0 will be sent to manager.cce.ufpr.br. Actions generated by this program translation are shown straightforward:

```
snmpset victim.cce.ufpr.br private 98.1.1.2.1.2.1 o tcp.tcpAttemptFails.0
snmpset victim.cce.ufpr.br private 98.1.1.2.1.3.1 i deltaValue
snmpset victim.cce.ufpr.br private 98.1.1.1.1.2.1 s "$1 > 25"
snmpset victim.cce.ufpr.br private 98.1.1.1.1.3.1 i unsigned32
snmpset victim.cce.ufpr.br private 99.1.1.1.1.3.1 i boolean
snmpset victim.cce.ufpr.br private 99.1.1.1.1.4.1 i absoluteValue
snmpset victim.cce.ufpr.br private 99.1.1.1.1.5.1 o 98.1.2.1.1.3.1
snmpset victim.cce.ufpr.br private 99.1.1.1.1.8.1 i 1
snmpset victim.cce.ufpr.br private 99.1.1.1.1.9.1 i unequal
snmpset victim.cce.ufpr.br private 99.1.1.1.1.10.1 i 0
snmpset victim.cce.ufpr.br private 99.1.2.1.1.3.1 i notification
snmpset victim.cce.ufpr.br private 99.1.2.1.1.6.1 o tcp.tcpAttemptFails.0
snmpset victim.cce.ufpr.br private 99.1.2.1.1.7.1 i true
snmpset victim.cce.ufpr.br private 99.1.1.1.1.6.1 i true
```

During normal computer operation, i. e., with no attack happening, this program was translated and we produced an attack similar to the former, from another host. In a representative result, after 7 seconds, manager.cce.ufpr.br got a notification.

This attack was detected with the time above but, even if it was aborted as soon as noticed, the service in port 23 would be interrupted. It happens due TCP state's machine only switches from SYN-RCVD to CLOSED after a timeout, when this service is already interrupted.

2 Link Overload Detection

This experience produced a notification when an overload was noticed over a link. In order to simulate a link overload, the link utilization was measured based on the number of IP datagrams, using a flow generator program, especially designed to it, whose source code is available in [1].

This program sends a huge number of small size UDP packages in order to cause link overload. We chose UDP because, different from TCP, this protocol has no algorithms to control the data flow.

The object ip.ipInReceives counts the number of IP datagrams received and ip.ipOutRequests counts the number of IP datagrams sent. The total of datagrams in a link is given by the sum of these two objects' values.

We used a computer with Linux, NET-SNMP agent, Event and Expression MIBs, ANEMONA translator and UDP flow generator's server and client plus another computer with Linux and UDP flow generator's server and client, both connected through a 10 Mbps interfaces.

Since objects ip.ipInReceives and ip.ipOutRequests are counters, they were sampled as delta, using 6 seconds intervals. The following ANEMONA program samples the mentioned objects as delta, add them and assign its result to an entry in Expression MIB results table, denoted by *utilization*, whose address will be reported to the user after this program translation.

```
watch: ahost.cce.ufpr.br using private
ip.ipInReceives.0 is Counter32: delta
ip.ipOutRequests.0 is Counter32: delta
begin
        bind utilization to (ip.ipInReceives.0 + ip.ipOutRequests.0);
end
```

The actions produced by this program are shown below:

```
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.2.1 o ip.ipInReceives.0
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.3.1 i deltaValue
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.2.2 o ip.ipOutRequests.0
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.3.2 i deltaValue
snmpset ahost.cce.ufpr.br private 98.1.1.1.1.2.1 s "$1 + $2"
snmpset ahost.cce.ufpr.br private 98.1.1.1.1.3.1 i counter32
```

After these commands, the entry address corresponding in the Expression MIB results table is associated to macro *utilization* and will be 98.1.2.1.1.2.1. This address is printed in the screen by ANEMONA translator.

The data achieved with the program above were collected in two distinct situations: with no network resources utilization and with heavy network utilization.

Without using applications which require network resources, the representative values of the total of network datagrams, given by the summation of ip.ipInReceives and ip.ipOutRequests deltas, registered in intervals of 1 minute were 80, 88 and 84, respectively.

To ensure heavy utilization of network resources, we established FTP connections to the server inside the monitored host. Although the heavy link utilization during a FTP session is related with datagram's size, this application ensures a continuous traffic in the server-client direction and is relevant to this experience.

In the beginning, we established a FTP session, after that, two, three and four sessions, respectively. To each new session, we collected the number of datagrams in this network three times, which is the sum of ip.ipInReceives and ip.ipOutRequests deltas, according to figure 3.

	Elapsed Time		
Number of FTP sessions	1 minute	2 minutes	3 minutes
1 Session	2055	2926	3003
2 Sessions	3052	3244	2762
3 Sessions	3686	3318	3837
4 Sessions	3796	3569	3230

Figure 3: Traffic in FTP sessions

After that, we monitored the number of network datagrams while the UDP flow generator runs. First of all, we monitored the flow in only one direction, i. e., with a server running in the monitored host and a client in the other computer on the net. Then we monitored the flow in two directions, with servers and clients running in each one of the computers used. Figure 4 shows representative values of the number of datagrams on the net, using 6 seconds interval, collecting from 6 in 6 seconds, with the flow generator running in one direction and two directions.

Elapsed Time	1 direction	2 directions
0 seconds	90	86
6 seconds	16091	96463
12 seconds	55483	57489
18 seconds	81360	62784
24 seconds	65859	125463
30 seconds	78524	124944

Figure 4: Number of IP datagrams

Based on data above, we chose 8000 as critical value for the delta of the number of datagrams on network. The program below samples ip.ipInReceives.0 and ip.ipOutRequests.0 as delta, calculates its sum, assigning this result to an entry in the Expression MIB results table, denoted by *utilization*. When the value assigned to *utilization* will be greater than the critical value, in this case 8000, a notification will be sent to the specified host with the object instance assigned to *utilization*.

```
watch: ahost.cce.ufpr.br using private
ip.ipInReceives.0 is Counter32: delta
ip.ipOutRequests.0 is Counter32: delta
begin
        bind utilization to (ip.ipInReceives.0 + ip.ipOutRequests.0);
        when utilization > 8000
        do
                notify manager.cce.ufpr.br private utilization
        end
end
```

The actions produced by this program translation are shown straightforward. Thus, the reader is able to compare the effort necessary to produce commands to manual programming with the use of ANEMONA programs.

```
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.2.1 o ip.ipInReceives.0
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.3.1 i deltaValue
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.2.2 o ip.ipOutRequests.0
snmpset ahost.cce.ufpr.br private 98.1.1.2.1.3.2 i deltaValue
```

```
snmpset ahost.cce.ufpr.br private 98.1.1.1.1.2.1 s "$1 + $2"
snmpset ahost.cce.ufpr.br private 98.1.1.1.1.3.1 i counter32
snmpset ahost.cce.ufpr.br private 99.1.1.1.1.3.1 i boolean
snmpset ahost.cce.ufpr.br private 99.1.1.1.1.4.1 i absoluteValue
snmpset ahost.cce.ufpr.br private 99.1.1.1.1.5.1 o 98.1.2.1.1.2.1
snmpset ahost.cce.ufpr.br private 99.1.1.1.1.8.1 i 1
snmpset ahost.cce.ufpr.br private 99.1.1.1.1.9.1 i unequal
snmpset ahost.cce.ufpr.br private 99.1.1.1.1.10.1 i 0
snmpset ahost.cce.ufpr.br private 99.1.2.1.1.3.1 i notification
snmpset ahost.cce.ufpr.br private 99.1.2.1.1.6.1 o 98.1.2.1.1.2.1
snmpset ahost.cce.ufpr.br private 99.1.2.1.1.7.1 i true
snmpset ahost.cce.ufpr.br private 99.1.1.1.1.6.1 i true
```

With a computer working under normal operation conditions, i. e., without running UDP flow generator, this program was translated and, after that, we start the flow generator server in this computer, and we ran its client on another computer attached on the network. In a representative experience, after 19 seconds a notification was received by manager.cce.ufpr.br[3].

3 Conclusion

This technical report documented the case studies done to monitor a given system, in order to detect Denial of Service attacks and also to detect a link overload.

During these tests, we confirmed the usefulness of ANEMONA to program Event and Expression MIBs, letting the user free from think about and type a lot of complex commands from NET-SNMP agent, as well as control MIB tables. Instead of this, the user has a high-level interface, given by ANEMONA language.

About this tool performance, we can say that actions generated by it are exactly the same than those would be generated by manual MIBs configuration, but with a smaller incidence of mistakes and faster execution, which increases the network administrator's productivity.

4 References

[1] FERNANDES, H. *ANEMONA: Uma Linguagem de Configuração para Aplicações de Monitoração de Redes*. Dissertação (Mestrado em Informática) – Departamento de Informática, Universidade Federal do Paraná, 2001.
[2] "Project Neptune". *Phrack Magazine. Volume Seven. Issue Forty-Eight.* July, 1996.
[3] COMER, D. *Internetworking with TCP/IP,* volume 1. New Jersey: Prentice-Hall, 1997.

[3] During the tests, manager.cce.ufpr.br was just a canonical name to ahost.cce.ufpr.br.