

1. Use the *snmptranslate* tool, whose man page is installed on GAUL, to find or retrieve the following. Use *script* to capture a session in which you find each answer.
[12 marks total]

- a. The complete numeric OID of the variable myWWWOkCount from your MIB.

One possible way is to use the command:

```
snmptranslate -On -IR myWWWOkCount
```

The output would look something like:

```
.1.3.6.1.4.1.8072.10387.1.1
```

- b. The full textual OID of the variable myWWWNotFoundCount from your MIB.

One possible way is to use the command:

```
snmptranslate -Onf -IR myWWWNotFoundCount
```

The output would look something like:

```
.iso.org.dod.internet.private.enterprises.netSnmp.myWWWMIB.myWWWMIBObjects.myWWWNotFoundCount
```

- c. The full details of the variable myWWWForbiddenCount from your MIB.

One possible way is to use the command:

```
snmptranslate -Td -IR myWWWForbiddenCount
```

The output would look something like:

```
MY-WWW-MIB::myWWWForbiddenCount
myWWWForbiddenCount OBJECT-TYPE
-- FROM MY-WWW-MIB
SYNTAX Integer32
MAX-ACCESS read-write
STATUS current
DESCRIPTION "This is an object that counts forbidden access queries."
DEFVAL { 0 }
 ::= { iso(1) org(3) dod(6) internet(1) private(4) enterprises(1) netSnmp(8072)
myWWWMIB(10387) myWWWMIBObjects(1) 3 }
```

- d. A "graphical" tree of your complete MIB.

One possible way is to use the command:

```
snmptranslate -Tp -IR myWWWMIIB
```

The output would look something like:

```
+--myWWWMIIB(10387)
|
+--myWWWMIIBObjects(1)
| |
| +- -RW- Integer32 myWWWOkCount(1)
| +- -RW- Integer32 myWWWNotFoundCount(2)
| +- -RW- Integer32 myWWWForbiddenCount(3)
| +- -RW- Integer32 myWWWMethodNotImplementedCount(4)
| +- -RW- Integer32 myWWWVersionNotSupportedCount(5)
| +- -RW- Integer32 myWWWShutdownCount(6)
|
+--myWWWMIIBConformance(2)
```

2. Suppose that we would like to get the number of successful web server responses in each hour of the day using only the implementation and coding done above. In other words, we would like *snmpget* to report the number of OK responses that occur in each hour of the day. How could we use *snmpset* to help us do this? Is this guaranteed to always give us an accurate count? Explain. [8 marks]

Essentially, we would use *snmpset* to reset the *myWWWOkCount* variable to 0, so that each hour starts with a fresh counter. This way, each hour would accumulate only the requests received in the hour. So, every hour, we would first do an *snmpget* to retrieve the number of requests for the hour, followed by an *snmpset* command to reset the variable back to 0 for the next hour of results. This is not always guaranteed to give an accurate count of all requests, however. If a request arrives at the server between our *snmpget* and *snmpset* requests to read and reset the counter, that request will not be counted. While the chances of this are small, it is still possible.

3. Instead of using simplistic SNMP security mechanisms (community strings), suppose we were to use SNMPv3 security for encryption and authentication in our agent. (So, in other words, you can assume we have secure communications between SNMP managers and the SNMP agent.) Compare the security issues of using shared memory versus Internet sockets for communication between the SNMP agent and your web server. Does one make the management of the web server less secure than the other? How would you be able to do insecure and secure communication with either mechanism? [10 marks]

In using shared memory, we are requiring the agent and web server to be on the same machine; no remote access to the shared memory region is possible. This makes shared memory attractive for avoiding external attacks ... someone must breach an account on the system first, before they can get anywhere close to the shared memory region. Even then, they need to do a couple of things. One is that they need to know

the key used to get access to the shared region. This is not overly difficult, as `ipcs` will usually give this away, if you don't have access to the source code of the program to tell you how the key was generated. Having the key alone is not enough, however, as shared memory regions are protected by the same permission scheme used to protect Unix files. With the sample code provided, only the owner of the process creating the shared memory region will have access to it, and the rest of the world cannot access it. It is possible to make shared memory regions insecure, however, by giving them group or world access privileges. Generally speaking, though, shared memory regions can be quite secure if used properly.

Internet sockets, on the other hand, are a different story. As soon as you create an Internet socket, any remote host in the world can connect to it by default, unless you have a firewall protecting you. (And, even then, anyone inside the perimeter of the firewall still has access.) Worse yet, no special privileges are required to send a message to a socket, so by default, there is absolutely nothing to secure either the counters or the shutdown mechanism from attacks. Security can still be provided however, by filtering out packets or connection attempts from external hosts ... which then makes local access a requirement, as was the case with shared memory. However, as mentioned earlier, there is no mechanism to protect you once someone gets access to the local host, as user identity is not tied to socket messages. To secure things properly, you would need to employ some form of encryption, authentication and message integrity measures that would not be necessary for shared memory. In short, you can have security when using sockets, but it is not available by default, and requires a lot of work and complexity to get right, when compared to shared memory.