

Enforcing a Two Man Rule Using Solaris 10 RBAC

by: Glenn Brunette, 05/11/2005

<http://www.securitydocs.com/library/3278>

The "two man rule" (also sometimes called the "four eyes rule") has its origins in military protocol although for quite some time it has been welcomed into the stockpile of IT security controls used by organizations around the world. The "two man rule" specifies that there must be two individuals that must act in concert in order to perform some action. Further, each individual should have comparable knowledge and skill in order to detect attempts of subversion initiated by the other. One can certainly see why this is essential for tasks such as safeguarding a country's nuclear arsenal. One would not want all of that authority to rest in the hands of one individual.

The "two man rule" is also useful for IT security. Whether you are talking about physical or logical access controls, the "two man rule" has been applied to help secure IT assets for quite some time. Whether you want to protect access to key data sets or restrict who may perform sensitive or high impact operations on a system, using the "two man rule" may be an option. That said, in many circumstances, more traditional IT security controls are likely appropriate. Using the "two man rule" is most often reserved for restricting the most sensitive IT security operations performed within an organization. Whether and where you could apply the "two man rule" to your organization will depend on your policy, architecture, processes and requirements.

Let's assume that you are interested in applying the "two man rule" on a given system. For our example, we will focus on implementing the "two man rule" on the Solaris 10 operating system. It should be noted that the same techniques apply to both Solaris 8 and 9 since all three versions of the operating system included the Solaris Role-based Access Control ("RBAC") facility. We will make use of this facility to implement the "two man rule" as described above.

The content and examples below will include references to both *normal* users and *roles*. In the Solaris OS, roles are similar to normal user accounts with two important differences. First, a role cannot be accessed directly over the network or from the console. You can only use the *su(1M)* command (or *smc(1M)*) to assume a role. In either case, a user must first authenticate to the system as himself before attempting to access a role. Secondly, a role can only be assumed by authorized users. That is, before a given user can assume a role, an administrator must assign that role to the user otherwise attempts to access the role will fail. Both of these restrictions are important for preserving accountability and will factor quite heavily in our implementation of the two man rule in Solaris 10.

So let's begin. For our example, we will implement the "two man rule" for our hypothetical IT security administrators, *joe* and *john*. From here, there are several ways that we can go:

- Configure both *joe* and *john* with normal accounts on the system, each able to assume an individual Solaris role. Neither *joe* nor *john* will know the password for the role that they are permitted to assume. For example, *joe* is permitted to assume the role *roleA* but is unable to do so since only *john* knows the password for that role. This option creates two roles (one for each user) each able to perform a restricted operation. This approach has the benefit of either user being able to assume the role (i.e., it does not require that one user be the initiator of the action). Further by permitting both users to be on the system they have more opportunity to monitor the actions of the other (e.g., Solaris auditing, syslog, etc.).
- Configure only one user on the system with permission to assume one role. In this example, one user would know the initial password while the other user would know the second (i.e., role) password. This case is similar to the first although it only requires one user and one role account on the system. It suffers from the disadvantage of not permitting the second user to log directly into the system thereby potentially missing critical monitoring opportunities. Further, it requires that *joe* must be first logged into the system before *john* can assume *roleA*.
- Configure both *joe* and *john* as in example one but only give them access to the same shared role. In this case, neither user would have the password to authenticate to that role. In order to obtain the authenticator for that role, both users would be required to work together to obtain the password from an external source such as a physical safe. This option has the benefit of greater protection since access to the safe could be placed in an area that requires additional physical controls and provides greater opportunities for monitoring. Further, the authenticator could be monitored and

changed by a third party who has no relationship to either *joe* or *john*. This approach also preserves the benefits of the first approach where both *joe* and *john* can easily monitor the activities of the other.

For our example, I will choose the first option. This option will allow me to showcase the capabilities of Solaris RBAC to implement the "two man rule" without depending on external factors such as those described in the third option above. The actual choice that you make will depend on your policy and requirements. Additional security controls (physical and logical) beyond those described in this article may also be required depending on the level of assurance that you need for your situation.

First, let's verify our users - *joe* and *john*:

```
# getent passwd joe john
joe:x:105:1::/export/home/joe:/bin/pfsh
john:x:106:1::/export/home/john:/bin/pfsh
```

You will note that these users have been configured with profile shells. This is only done to simplify some of the commands that follow. If you did want to use a profile shell, you could prefix your commands with *pfexec(1)* when you wanted them to be evaluated by Solaris RBAC to run with privilege.

We can verify that these accounts do not have any special privileges (beyond those available to any regular Solaris user) by running the following commands: # roles joe john joe : No roles john : No roles

```
# profiles -l joe john
```

```
All:
  *
```

```
All:
  *
```

```
# auths joe john
joe : solaris.device.cdrw, solaris.profmgr.read,
      solaris.jobs.users,
      solaris.mail.mailq, solaris.admin.usermgr.read,
solaris.admin.logsvc.read,
      solaris.admin.fsmgr.read,
      solaris.admin.serialmgr.read,
      solaris.admin.diskmgr.read,
solaris.admin.procmgr.user, solaris.compsys.read,
      solaris.admin.printer.read,
      solaris.admin.prodreg.read,
solaris.admin.dcmgr.read,
      solaris.snmp.read,
      solaris.project.read,
      solaris.admin.patchmgr.read,
solaris.network.hosts.read,
      solaris.admin.volmgr.read
john : solaris.device.cdrw, solaris.profmgr.read,
      solaris.jobs.users,
      solaris.mail.mailq, solaris.admin.usermgr.read,
solaris.admin.logsvc.read,
      solaris.admin.fsmgr.read,
      solaris.admin.serialmgr.read,
```

```
solaris.admin.diskmgr.read,  
solaris.admin.procmgr.user, solaris.compsys.read,  
solaris.admin.printer.read,  
solaris.admin.prodreg.read,  
solaris.admin.dcmgr.read, solaris.snmp.read,  
solaris.project.read,  
solaris.admin.patchmgr.read,  
solaris.network.hosts.read,  
solaris.admin.volmgr.read
```

As you can see, neither *joe* nor *john* have been granted access to any roles, rights profiles or authorizations (beyond those available to any normal Solaris user). You will notice several authorizations that are available to them. They have been granted by default using the *AUTHS_GRANTED* and *PROFS_GRANTED* parameters in the */etc/security/policy.conf* file.

To each of these users we will grant the rights profile *Audit Review*. This will allow both *joe* and *john* to review Solaris audit records so that each can monitor the activities of the other. This step is not necessary for the "two man rule" if you have an external third party who is able to monitor the activities of the two users.

```
# usermod -P "Audit Review" joe  
# usermod -P "Audit Review" john
```

Let's verify that this change has taken effect:

```
# profiles -l joe john  
  
Audit Review:  
  /usr/sbin/praudit      euid=0  
  /usr/sbin/auditreduce  euid=0  
  /usr/sbin/auditstat    euid=0  
All:  
  *  
  
Audit Review:  
  /usr/sbin/praudit      euid=0  
  /usr/sbin/auditreduce  euid=0  
  /usr/sbin/auditstat    euid=0  
All:  
  *
```

Now that we have verified the accounts have been configured as expected, we will create two roles: *roleA* and *roleB*. *roleA* will be assigned to *joe*, and *roleB* will be assigned to *john*. To these roles, our restricted operations will be assigned.

```
# roleadd -d /export/home/roleA -m roleA  
64 blocks  
  
# passwd roleA  
New Password:  
Re-enter new Password:  
passwd: password successfully changed for roleA  
  
# roleadd -d /export/home/roleB -m roleB  
64 blocks
```

```
# passwd roleB
New Password:
Re-enter new Password:
passwd: password successfully changed for roleB

# usermod -R roleA joe
# usermod -R roleB john
```

Now, let's verify that the roles have been created and assigned as we expected:

```
# getent passwd roleA roleB
roleA:x:107:1::/export/home/roleA:/bin/pfsh
roleB:x:108:1::/export/home/roleB:/bin/pfsh

# roles joe john
joe : roleA
john : roleB
```

Great. Our last step is to assign the restricted operations to the roles so that *joe* and *john* can perform them after they have jointly assumed either *roleA* or *roleB*. For our example, we will assign the *Crypto Management* rights profile to both *roleA* and *roleB*. This will allow either of those roles to execute cryptographic administration functions such as those provided by *cryptoadm(1M)*:

```
# rolemod -P "Crypto Management" roleA
# rolemod -P "Crypto Management" roleB
```

To see what privileged commands are now available to these roles, use the following command:

```
# profiles -l roleA roleB

Crypto Management:
  /usr/sbin/cryptoadm      euid=0
  /usr/sfw/bin/CA.pl       euid=0
  /usr/sfw/bin/openssl     euid=0
All:
  *

Crypto Management:
  /usr/sbin/cryptoadm      euid=0
  /usr/sfw/bin/CA.pl       euid=0
  /usr/sfw/bin/openssl     euid=0
All:
  *
```

Great! We are all set. At this point, we have two users (*joe* and *john*). *joe* is permitted to assume *roleA*, and *john* is permitted to assume *roleB*. *joe* does not know the password for *roleA* (although *john* does), and similarly *john* does not know the password for *roleB* (although *joe* does). Both of the users have been assigned the *Audit Review* rights profile allowing them to monitor each other's activities. Both of the roles have been assigned the *Crypto Management* rights profile allowing them to perform cryptographic management operations on the system.

Let's verify that this all actually works... For my example, all of the users and roles were created within a zone called "blue".

The first thing that I will do to test is log into the zone as *joe*:

```
# zlogin -l joe blue
[Connected to zone 'blue' pts/2]
Last login: Tue Apr 12 07:20:19 on pts/2
joe$ id -a
uid=105(joe) gid=1(other) groups=1(other)
joe$ auths
solaris.audit.read,
    solaris.device.cdrw,solaris.profmgr.read,
    solaris.jobs.users,solaris.mail.mailq,
solaris.admin.usermgr.read,
    solaris.admin.logsvc.read,
    solaris.admin.fsmgr.read,
    solaris.admin.serialmgr.read,
solaris.admin.diskmgr.read,
    solaris.admin.procmgr.user,solaris.compsys.read,
    solaris.admin.printer.read,
solaris.admin.prodreg.read,
    solaris.admin.dcmgr.read,
    solaris.snmp.read,
    solaris.project.read,
solaris.admin.patchmgr.read,
    solaris.network.hosts.read,
    solaris.admin.volmgr.read
joe$ profiles -l

    Audit Review:
        /usr/sbin/praudit      euid=0
        /usr/sbin/auditreduce  euid=0
        /usr/sbin/auditstat    euid=0
    All:
        *
joe$ roles
roleA
```

First, since *joe* does know the password for *roleB*, let's see if we can assume that role:

```
joe$ su roleB
Password:
Roles can only be assumed by authorized users
su: Sorry
```

Even with a valid password, *joe* is not permitted to assume *roleB*. Next, let's try to guess the password for *roleA*:

```
joe$ su roleA
Password:
su: Sorry
joe$ su roleA
Password:
su: Sorry
joe$ su roleA
Password:
```

```
su: Sorry
```

No luck. In order to prevent brute force password attempts, you could also configure account lockout so that an account will be administratively locked after *n* consecutive failed authentication attempts, where *n* is the value defined by the *RETRIES* parameter in */etc/default/login*. Next, let's see what user *john* would see in the audit trails after *joe* executed the above commands. Since *john* has been granted the *Audit Review* rights profile, he is able to look at the Solaris audit records. If you wanted to further limit what *john* could see in the audit trail, you could develop a small wrapper script to call *praudit(1M)* and *auditreduce(1M)* to filter out unwanted records. So, let's pick up our example with *john* logging into the "blue" zone:

```
# zlogin -l john blue
[Connected to zone 'blue' pts/2]
john$ id -a
uid=106(john) gid=1(other) groups=1(other)
john$ auths
solaris.audit.read,
    solaris.device.cdrw,solaris.profmgr.read,
    solaris.jobs.users,solaris.mail.mailq,
solaris.admin.usermgr.read,
    solaris.admin.logsvc.read,
    solaris.admin.fsmgr.read,
    solaris.admin.serialmgr.read,
solaris.admin.diskmgr.read,
    solaris.admin.procmgr.user,solaris.compsys.read,
    solaris.admin.printer.read,
solaris.admin.prodreg.read,
    solaris.admin.dcmgr.read,
    solaris.snmp.read,
    solaris.project.read,
solaris.admin.patchmgr.read,
    solaris.network.hosts.read,
    solaris.admin.volmgr.read
john$ profiles -l

    Audit Review:
        /usr/sbin/praudit      euid=0
        /usr/sbin/auditreduce  euid=0
        /usr/sbin/auditstat    euid=0
    All:
        *
john$ roles
roleB
```

Next, *john* uses the *praudit* and *auditreduce* commands to look at any attempts to *su* taken by *joe*:

```
john$ auditreduce -m AUE_su -r joe | praudit -s
file,2005-04-12 07:25:06.000 -04:00,
header,97,2,AUE_su,,10.8.31.9,2005-04-12 07:28:30.220 -04:00
subject,gmb,root,other,joe,other,1834,3097759606,
12114 22 129.150.66.247
text,bad auth. for user roleB
return,failure,2
header,97,2,AUE_su,,10.8.31.9,2005-04-12 07:28:40.043 -04:00
subject,gmb,root,other,joe,other,1835,3097759606,
```

```

12114 22 129.150.66.247
text,bad auth. for user roleA
return,failure,2
header,97,2,AUE_su,,10.8.31.9,2005-04-12 07:28:49.940 -04:00
subject,gmb,root,other,joe,other,1836,3097759606,
12114 22 129.150.66.247
text,bad auth. for user roleA
return,failure,2
header,97,2,AUE_su,,10.8.31.9,2005-04-12 07:28:59.683 -04:00
subject,gmb,root,other,joe,other,1837,3097759606,
12114 22 129.150.66.247
text,bad auth. for user roleA
return,failure,2
file,2005-04-12 07:28:59.000 -04:00,

```

>From this audit trail, you can see that user *joe* attempted to assume both *roleA* and *roleB*. In fact, *joe* attempted to assume *roleB* three times with each attempt ending in failure. For more information on the audit logs, their format and options for configuring them, see *audit(1M)*. For the purposes of this example, Solaris auditing was configured with the following policies:

```

# auditconfig -getpolicy
audit policies = argv,cnt,perzone

```

Further, the users and roles defined in this example were audited as follows:

```

# cat /etc/security/audit_user
#
# Copyright (c) 1988 by Sun Microsystems, Inc.
#
# ident "@(#)audit_user.txt      1.6      00/07/17 SMI"
#
#
# User Level Audit User File
#
# File Format
#
#      username:always:never
#
root:lo:no
joe:lo,ad,ex:
john:lo,ad,ex:
roleA:lo,ad,ex:
roleB:lo,ad,ex:

```

These settings effectively audited login and logout events, administrative actions and command execution (*exec(2)* calls). You should configure Solaris auditing to comply with your local auditing requirements and policies. The configuration presented serves only as an example to illustrate the type of information that can be collected.

By now, you can see that *joe* cannot access the *Crypto Management* rights profile on his own. Let's see what would happen if *joe* attempted to simply perform a restricted cryptographic operation on his own:

```

joe$ cryptoadm stop
cryptoadm: failed to stop cryptographic framework daemon - Not owner.

```

Still no luck. In order to perform those restricted operations, he will need *john* to help him assume *roleA*. In the final example below, we illustrate that working in concert, *joe* and *john* together can successfully assume *roleA*.

```
joe$ su roleA
Password:
roleA$ id -a
uid=107(roleA) gid=1(other) groups=1(other)
roleA$ profiles -l

    Crypto Management:
        /usr/sbin/cryptoadm      euid=0
        /usr/sfw/bin/CA.pl       euid=0
        /usr/sfw/bin/openssl     euid=0
    All:
        *
```

With these privileges, *joe* and *john* can perform operations such as starting and stopping the Solaris cryptographic framework:

```
roleA$ echo "foo" | digest -a md5
d3b07384d113edec49eaa6238ad5ff00
roleA$ cryptoadm stop
roleA$ echo "foo" | digest -a md5
digest: failed to initialize PKCS #11 framework: CKR_GENERAL_ERROR
roleA$ cryptoadm start
roleA$ echo "foo" | digest -a md5
d3b07384d113edec49eaa6238ad5ff00
```

Clearly, you can see that the "two man rule" is enforced since neither *joe* nor *john* can perform any cryptographic management operations on the system without knowledge and support of the other. Further, should either user attempt to access their assigned role, the other user can be alerted through the presence of a Solaris audit record (that neither user can modify). Very cool.

Finally, I would like to thank both Collin Sampson and Scott Rotondo for their careful review and feedback provided on this article.

As always, I am interested in your feedback and thoughts. Please let me know if you have any questions. Take care!