PRIVILEGE DEBUGGING IN THE SOLARIS™ 10 OPERATING SYSTEM

Glenn Brunette, Security Program Office, Client Solutions
Darren Moffat, Solaris Security Technologies Group,
Operating Platforms Group (OPG)

Sun BluePrints™ OnLine — February 2006

819-5507-10
Revision 0.9, 1/18/06
Edition: February 2006

Please
Recycle

TABLE OF CONTENTS

Privilege Debugging in the Solaris™ 10 Operating System 1
Introduction 1
What is Process Rights Management? 2
What is Privilege Debugging? 3
Using DTrace for Privilege Debugging 4
Using the privdebug Program 5
Privilege Debugging Using an Example Apache2 Service 7
Privilege Debugging the Apache2 Service—Global Zone Method 8
Privilege Debugging the Apache2 Service—Local Zone Method 17
Conclusion 21
References 21
Sun BluePrints 21
Sun Product Documentation and Manual Pages 21
Other Documentation, Articles and Web Sites 22
About the Authors 22
Glenn Brunette 22
Darren Moffat 23
Acknowledgements 23
Ordering Sun Documents 23
Accessing Sun Documentation Online 23

Privilege Debugging in the Solaris™ 10 Operating System

This Sun BluePrints™ OnLine article describes how, in the Solaris™ 10 Operating System (Solaris 10 OS), to profile applications and services in order to determine which Solaris 10 privileges they attempt to use. With this information, organizations can then restrict those applications and services so that they are granted only the absolutely necessary privileges that they need to fulfill their intended purpose.

This article contains the following sections:

- Introduction
- What is Process Rights Management?
- What is Privilege Debugging?
- Using DTrace for Privilege Debugging
- Using the privdebug Program
- Privilege Debugging Using an Example Apache2 Service
- Conclusion
- References
- About the Authors
- Acknowledgements

Introduction

The traditional UNIX® privilege model is based on the concept of a super-user. In this model, the system associates all of its privileged operations with the `root` account or—more precisely—the user identifier (UID) 0. All other UIDs are considered unprivileged by the operating system. This “all or nothing” approach to privilege delegation means that any application that must perform a privileged operation, such as a binding to a reserved network port (for example, one whose port number is less than 1024), must be started as `root`.

Starting applications in this manner, however, is inherently risky because it means that the application will have privilege to do anything on the system. Administrators are forced to trust the applications to use only the privileges that they need and only in the ways that are expected. Consequently, disaster could ensue should the application not manage its use of privilege safely, or should the application be misconfigured or exploited in some way.

The Solaris 10 OS introduced many new security features and capabilities. One of its most significant security enhancements is the *Process Rights Management* capability. Process Rights Management replaces the dependence on a special UID (for example, UID 0), using instead a number of discrete and well-defined privileges that can be individually delegated or revoked as needed. While support for the traditional privilege model is still maintained for backward compatibility, this new approach offers administrators and software developers the ability to exercise fine-grained control over the delegation and use of privilege in the Solaris OS.

In IT security, the well-known “least privilege” principle states that: *“Every program and every user of the system should operate using the least set of privileges necessary to complete the job.”*[1]

In the Solaris 10 OS, administrators can now configure their applications and services for least privilege through the careful delegation of individual privileges. Programs and services that traditionally needed to be started as `root` can now simply be assigned only the privileges they need, and started as an otherwise unprivileged user. This way, they have only the powers they require to perform their specific privileged tasks. Optionally, applications can also be developed to be “privilege aware,” which gives them even greater control over which privileges are enabled, disabled, or used. When you deploy applications and services on Solaris 10 OS, the question arises: How do you know what privileges an application needs?

This article describes how to quickly and easily determine which privileges might be needed by a program, service, or process. With this information, you can adjust how that program, service, or process is started so that only the absolutely necessary privileges are actually granted. This can be accomplished whether you are invoking programs from a command line or starting services using the Solaris 10 Service Management Facility (SMF). In turn, these actions can improve the overall security of the system by ensuring that applications and services are running with just the right privileges—and no more.

What is Process Rights Management?

Solaris Process Rights Management is a capability in the Solaris 10 OS that allows the traditional super-user authority to be divided into a discrete set of privileges, each with a specific purpose. A privilege or set of privileges can be granted to a process to enable it to accomplish tasks that normally would have required administrative or super-user privilege. For example, the `file_dac_read` privilege is used to provide an otherwise unprivileged process with the ability to read any file on the system, regardless of its ownership or permissions. For an entire listing of privileges, see `privileges(5)` or the output of the `ppriv -vl` command.

Three special privilege keywords are discussed in this article: `all`, `zone`, and `basic`.

- As the name suggests, `all` refers to the complete set of privileges on the system. The `all` keyword is appropriate only when used within the Solaris 10 Global Zone.
- When working within a Solaris 10 Container, the `zone` keyword is used to denote all of the privileges available for use in a local zone. Today, Solaris™ Containers run with inherently less privilege than the Global Zone (although this might change in a future release of the Solaris OS). Several of the privileges found in the `all` set are not available in a Solaris Container, hence the reason for a separate keyword.
- The `basic` keyword is used to refer to a set of privileges that all processes—privileged and unprivileged—are accustomed to having. These `basic` privileges are, by default, assigned to every process, although they can be taken away if they are not needed.

Given that there are nearly 50 distinct privileges available in the Solaris 10 OS, it is critical to quickly and easily determine which privileges are relevant for any given program, service, or process. Subsequent

1.“”The Protection of Information in Computer Systems,” Jerome Saltzer and Michael Schroeder. April 17. 1975. http://web.mit.edu/Saltzer/www/publications/protection/

sections describe some of the methods available for enumerating the privileges used by a process or service.

What is Privilege Debugging?

Strictly speaking, the term *privilege debugging* is used to describe the process of enumerating those privileges needed by a service or process in order to be able to run and perform its tasks successfully. With this list of privileges, a privilege profile can be developed for use with Role-based Access Control (RBAC) execution profiles or SMF service profiles, allowing administrators to define the privileges with which a service or program should be run.

The Solaris 10 OS provides a rudimentary privilege debugging capability in the `ppriv(1)` command. The `ppriv` command can be configured to attach to an existing process, or start a new one, with privilege debugging enabled using its `-D` (privilege debugging) option.

```
usage: ppriv [-v] [-S] [-D|-N] [-s spec] { pid | core } ...
    ppriv -e [-D|-N] [-s spec] cmd [args ...]
    ppriv -l [-v] [privilege ...]

    (report, set or list process privileges)
```

However, the `ppriv` notion of privilege debugging is limited to debugging privilege check failures. When a necessary privilege is not available, the `ppriv` command can be configured to tell the user which privilege was needed when the program failed. Consider the following example.

```
$ ppriv -e -D cat /etc/shadow
cat[6286]: missing privilege "file_dac_read" (euid = 100, syscall =
225) needed at ufs_iaccess+0xd2
cat: cannot open /etc/shadow
```

In this example, an unprivileged user (EUID `100`) was not able to open the `/etc/shadow` file for reading. The attempt to perform this action generated a missing privilege message, explaining that the `file_dac_read` privilege was not granted to the user, and therefore the operation could not proceed. In addition to printing the error message to the user's terminal session, the following information is logged via SYSLOG.

```
Nov 14 09:35:50 blackhole genunix: [ID 702911 kern.notice] cat[6286]:
missing privilege "file_dac_read" (euid = 100, syscall = 225) needed at
ufs_iaccess+0xd2
```

Further, the Solaris 10 OS can be globally configured to use this form of privilege debugging through the use of the `priv_debug` parameter defined in `/etc/system`. If this parameter is set to `1`, then privilege debugging is enabled for the entire system.

As with many `/etc/system` parameters, the current setting can be queried using the `mdb(1)` command.

```
# echo "priv_debug/X" | mdb -k
priv_debug:
priv_debug:   0
```

In this case, the `priv_debug` setting was disabled (`0`). Although this approach is useful for debugging single error messages, it can be cumbersome when needing to iteratively exercise all of the possible failure cases for a given service before a listing of privileges can be derived. Ideally, a process should be able to run in a training or evaluation mode and in a controlled environment in order to have access to any privilege on the system. While in this state, the use of all of the privileges by a process (or group of processes) could then be recorded in order to enumerate all the privileges that had been used.

To be successful even in this model, however, the process or service must be fully exercised to ensure that every code path in which privileges might be used has been traversed. Failure to fully exercise the service in this way might mean that some privileges needed by the service have not been identified. Further, be sure to exercise not only the successful operation of the software, but also error conditions. Error conditions often involve infrequently-used areas of the source code and could require privileges.

Using DTrace for Privilege Debugging

To assist in determining which privileges are used by a program (successfully or not), use the Dynamic Tracing Tool (DTrace) in the Solaris 10 OS. In the Solaris 10 OS 03/2005 release, there are two privilege-specific DTrace probes that are useful for privilege debugging: `priv-ok` and `priv-err`.

```
# dtrace -l | grep priv-
 4523    sdt      genunix         priv_policy_only priv-ok
 4524    sdt      genunix       priv_policy_choice priv-ok
 4525    sdt      genunix              priv_policy priv-ok
 4526    sdt      genunix        priv_policy_only priv-err
 4527    sdt      genunix      priv_policy_choice priv-err
 4528    sdt      genunix         priv_policy_err priv-err
```

Warning – The specific DTrace probes used in this article are not declared by Sun to be a stable interface (see `attributes(5)` for more information). As a result, these particular interfaces might change or might be removed in future releases, updates, or patches to the Solaris OS.

These probes are used to detect the successful (`priv-ok`) and unsuccessful (`priv-err`) use of privilege. With this information and a little DTrace scripting, it is possible to implement a privilege debugging mechanism that does not rely solely on failure cases for detection.

In the following example, DTrace is used to simply print an entry whenever the `priv-ok` or `priv-err` probes are activated. This provides a quick and easy way to verify that this approach will work.

```
# dtrace -n 'sdt:::priv-*'
dtrace: description 'sdt:::priv-*' matched 6 probes
CPU   ID                    FUNCTION:NAME
 0  4525          priv_policy:priv-ok
 0  4523     priv_policy_only:priv-ok
 0  4523     priv_policy_only:priv-ok
 0  4525          priv_policy:priv-ok
 0  4525          priv_policy:priv-ok
 0  4525          priv_policy:priv-ok
 0  4525          priv_policy:priv-ok
 0  4528     priv_policy_err:priv-err
 0  4523     priv_policy_only:priv-ok
```

Before continuing, however, it is useful to know what other kinds of information can be gathered. Adjusting the DTrace invocation in the following way can undercover more helpful information.

```
# dtrace -n 'sdt:::priv-* { printf("%d %d %s\n", arg0, pid, execname); }'
dtrace: description 'sdt:::priv-* ' matched 6 probes
CPU   ID                        FUNCTION:NAME
 0  4525       priv_policy:priv-ok 27 7062 sh
 0  4525       priv_policy:priv-ok 26 7083 sh
 0  4525       priv_policy:priv-ok 26 7083 ps
 0  4525       priv_policy:priv-ok 28 7083 ps
```

Just as in the previous example, DTrace was configured to display a message whenever the `priv-ok` or `priv-err` probe was used. In this case, the output was changed so that it included the number of the privilege being checked, the process identifier attempting to use the privilege, and the actual command name (associated with the process ID). The example output shows that a shell used privileges 27 and 26, and then the `ps` command used privileges 26 and 28.

What does this mean? Normally, users and processes interact with privilege names in the Solaris 10 OS, not privilege numbers. However, because this command collects information from within the kernel, which operates using privilege numbers, then numbers appear in the output. To map privilege numbers to names, inspect the `/usr/include/sys/priv_const.h` file.

```
#define PRIV_PROC_EXEC      26
#define PRIV_PROC_FORK      27
#define PRIV_PROC_INFO      28
```

In this case, the shell was likely spawning the `ps` command, a combination of a `fork(2)` and `exec(2)`. The `ps` command used the `proc_info` privilege to verify that it had permission to show processes other than those owned by the user issuing the `ps` command.

Caution – This particular approach might not work in all cases, because third-party kernel modules can register additional privileges with the kernel. Such updates would not change the Sun-provided `/usr/include/sys/priv_const.h` file. It is also possible for an administrator to add privileges using the `/etc/security/device_policy` file.

Using the privdebug Program

Building on the foundation provided by DTrace, `privdebug` is a small Perl program that can help with privilege debugging operations. Within the `privdebug` command is a small DTrace program that performs the core of the enumeration work. The Perl wrapper is used to perform the privilege number to name mapping and to provide more flexibility with respect to pre- and post-processing. Use it to enumerate the privileges used by programs and services.

Note – The `privdebug` command is currently unsupported but freely available. The source code for the `privdebug` program can be found at the following URL:
`http://www.opensolaris.org/os/community/security/files/`

The `privdebug` command has the following syntax:

```
# ./privdebug -h
privdebug [-f] [-v] [-H] [-o out]
   -n <EXECNAME>    Debug a specific program name
   -p <PID>       Debug a specific process ID
   -z <ZONENAME>    Debug a specific zone name
privdebug [-f] [-v] [-H] [-o out]
   -e <COMMAND>     Execute and debug a specific command
privdebug --help | -h
```

The `privdebug` command has two primary modes:

- It can attach to an existing process running somewhere on the system using the following options: `-n`, `-p`, and `-z`. These options can be used in any combination to be as specific as needed.
- It can execute the specific program to be debugged using the `-e` option.

Upon successful startup, `privdebug` will display a header line, which includes the following fields:

Field	Description
`STAT`	Contains the status of the privilege operation. Indicates whether a given privilege was used (`USED`) or whether the privilege check failed because the privilege was needed (`NEED`).
`TIMESTAMP`	Simple timestamp indicating when the privilege check was made. Time is expressed as the number of elapsed seconds since January 1, 1970.
`PPID`	Parent process ID associated with the process that caused the privilege check.
`PID`	Process PID of the process that caused the privilege check.
`PRIV`	Name of the privilege that was being checked.
`CMD`	Name of the command that caused the privilege check.

The rest of this section provides several sample invocations and associated output.

In the following example, `privdebug` is used to trace the `in.telnetd` process (and all of its children):

```
# ./privdebug -n in.telnetd -f -v
STAT TIMESTAMP        PPID  PID   PRIV            CMD
USED 1231183251124451 238   7115  sys_audit       in.telnetd
USED 1231183251139719 238   7115  sys_audit       in.telnetd
USED 1231183251612259 238   7115  proc_fork       in.telnetd
USED 1231183251974167 7115  7116  proc_exec       in.telnetd
USED 1231183472328575 238   7115  proc_fork       in.telnetd
USED 1231183472556716 7115  7117  proc_exec       in.telnetd
USED 1231183478414533 238   7115  proc_fork       in.telnetd
USED 1231183482742793 7115  7118  file_dac_write  in.telnetd
USED 1231183504062754 7115  7118  proc_exec       in.telnetd
[...]
```

In the following example, `privdebug` is attached to PID 7062 (`/bin/bash`). In this case, the output shows privileges associated with the shell, as well as commands run from the shell:

```
# ./privdebug -p 7062 -f -v
STAT TIMESTAMP        PPID  PID   PRIV            CMD
USED 1231312074567753 6673  7062  proc_fork       sh
USED 1231312076416810 7062  7154  proc_exec       sh
USED 1231323896848942 6673  7062  proc_fork       sh
USED 1231323898693786 7062  7155  proc_exec       sh
USED 1231323924895005 7062  7155  proc_fork       zlogin
USED 1231323927290452 7155  7156  proc_fork       zlogin
USED 1231323958729003 7155  7156  sys_devices     zlogin
USED 1231323959964745 7155  7156  file_dac_write  zlogin
USED 1231323960139726 7155  7156  proc_exec       zlogin
```

In the final example, the output traces commands run directly from `privdebug`:

```
# ./privdebug -f -v -e ps
STAT TIMESTAMP        PPID  PID   PRIV           CMD
USED 1232521405311329 7278  7279  proc_exec      ps
USED 1232521494214885 7278  7279  proc_info      ps
USED 1232521494225841 7278  7279  proc_info      ps
```

Using these basic constructs, you can enumerate privileges used or needed by processes and services in the Solaris 10 OS.

Privilege Debugging Using an Example Apache2 Service

This section applies privilege debugging tools in a real world scenario using the Apache2 Web service that is available in the SUNWapch2r and SUNWapch2u packages in the Solaris 10 OS. This service is identified by the SMF Fault Management Resource Identifier (FMRI) as:

`svc:/network/http:apache2`

which will be abbreviated in this section as `apache2`. In addition, the abbreviated name `httpd` will be used to refer to the actual Apache2 processes that are started by the system.

This section describes two approaches used to examine the process of privilege debugging:

- Privilege Debugging the Apache2 Service—Global Zone Method
- Privilege Debugging the Apache2 Service—Local Zone Method

To illustrate the basic concepts, privilege debugging will be performed first within a Solaris 10 Global Zone, to keep the example simple and to leverage all of the privileges on the system. Next, privilege debugging will be performed in a Solaris Container called `web_svc`. Examining privileges within the context of a Solaris Container affords some protection given that, by default:

- You will grant all privileges to the program or service.
- You might not want this service or program to impact others running in other Solaris Containers or in the Global Zone.

For the second example, the command-line prompts will be prefixed with either `global` (for the Global Zone) or `web_svc` (for the Solaris Container) so that it is obvious where the commands are issued.

Note – The details of how to configure and install a Solaris Container are outside of the scope of this document. The Solaris Container used in this example is based on the default Sun template. It was changed only to define the Solaris Container root directory (that is the `zonepath`) and network interfaces.

Privilege Debugging the Apache2 Service—Global Zone Method

This section examines the process of privilege debugging within the context of a Solaris 10 Global Zone. All of the commands and output recorded in this section have been entered from a terminal session attached to the Global Zone.

The process in this section involves completing the following steps:

- Step 1. Verify the Current State of the Apache2 Service
- Step 2. Install the ApachePD Privilege Debugging Rights Profile
- Step 3. Assign the ApachePD Rights Profile to the gmb Account
- Step 4. Assign the Necessary DTrace Privileges to the gmb Account
- Step 5. Start a Privilege Debugging Session
- Step 6. Start the Apache2 Service
- Step 7. Exercise the Apache2 Service
- Step 8. Stop the Apache2 Service
- Step 9. Construct an SMF Method Context for the Apache2 Service
- Step 10. Validate an SMF Method Context for the Apache2 Service

By following these steps, you will be able to enumerate the privileges used by the Apache2 service and then construct an SMF method context so that the service can be automatically started and stopped with the set of privileges identified.

Step 1. Verify the Current State of the Apache2 Service

Begin by ensuring that the Apache2 service is in a known good state to eliminate the possibility that service configuration problems could cause failures in subsequent steps. Issue the following commands to verify that the Apache2 service can be successfully started and stopped.

```
# svcs apache2
STATE       STIME   FMRI
disabled    9:59:13 svc:/network/http:apache2

# svcadm enable apache2
# svcs apache2
STATE      STIME   FMRI
online     9:59:25 svc:/network/http:apache2

# svcadm disable apache2
# svcs apache2
STATE       STIME   FMRI
disabled    9:59:31 svc:/network/http:apache2
```

Note – Before proceeding further, depending on your environment, consider testing the service more strenuously. You must have a high degree of assurance that the service is functioning properly and as expected. If the Apache2 service is not operating correctly in any way, fix the service before proceeding to Step 2.

Step 2. Install the ApachePD Privilege Debugging Rights Profile

Next, create a privilege debugging rights profile for the Apache2 service. This profile (called ApachePD) will be used to ensure that the service is started with a specific UID, GID, and set of privileges (in this case, `all`). To accomplish this task, modify the `/etc/security/prof_attr` and `/etc/security/exec_attr` files, as shown in the following examples.

```
# grep "^ApachePD" /etc/security/prof_attr
ApachePD:::Apache Service:
# grep "^ApachePD" /etc/security/exec_attr
ApachePD:solaris:cmd:::/lib/svc/method/http-apache2:
uid=webservd;gid=webservd;privs=all
```

In order to ensure that all of the needed privileges are enumerated, be sure to specify the UID and GID that you intend to use when the program or service is actually deployed. In general, the Solaris kernel will first check whether a process can perform the action without the use of privilege, even when the process has additional privileges. As such, privilege checks might not be performed (and therefore not enumerated) if the user attempting the operation can accomplish it without the use of privilege.

The `http-apache2` command, defined in the execution profile that was just created, is configured to run with `all` privileges in order to ensure that any privilege checks that are performed can succeed, allowing the program to continue uninterrupted. Depending on your environment, consider using fewer privileges if you are certain that they will not be needed. This approach can be used to contain or restrict a program even while its use of privileges is examined. In the worst case, if a needed privilege is not available, then the program will likely terminate and the process of testing will need to start over.

Step 3. Assign the ApachePD Rights Profile to the gmb Account

Once the Solaris 10 RBAC rights profile is ready, it can be assigned to the user who will perform the evaluation. In this example, the `gmb` account will be used to perform privilege debugging-related tasks. In order to use the ApachePD rights profile defined in the previous step, assign that rights profile to the `gmb` account using the `usermod(1M)` command. Exercise caution in order to avoid inadvertently losing any existing rights profiles in the process. The `usermod(1M)` manual page states the following information.

```
[...]
-P profile One or more comma-separated rights profiles defined in
           prof_attr(4). This replaces any existing profile setting.
           If no profile list is specified, the existing setting is
           removed.
[...]
```

This means that the `-P` option is destructive and will simply replace any existing profiles with those supplied on the command line. Therefore, you must add the new ApachePD rights profile while also

preserving any others that already exist. Use the `profiles(1)` command to determine whether the `gmb` account has any existing profiles.

```
# profiles gmb
Basic Solaris User
All
```

Be careful when using the `profiles` command because its output includes profiles defined in `user_attr(4)` and profiles that are globally defined in `/etc/security/policy.conf`. In this case, the `Basic Solaris User` and `All` profiles are globally defined and will not be overwritten by the `usermod` command that follows.

Use the following command to assign the ApachePD rights profile to the `gmb` user.

```
# usermod -P “ApachePD” gmb
```

Verify that this change has been properly applied using the `profiles(1)` command.

```
# profiles gmb
ApachePD
Basic Solaris User
All
```

Alternatively, the contents of `/etc/user_attr` could have been visually inspected.

```
# grep "^gmb:" /etc/user_attr
gmb::::type=normal;profiles=ApachePD
```

Note – Using the `profiles` command is recommended over manually inspecting the `user_attr` file because it shows what the consumers of the information see. It is also useful for detecting typographical errors that are not as obvious when looking at the raw data files.

At this point, the `gmb` user is now able to use the ApachePD rights profile to start or stop the Web server.

Step 4. Assign the Necessary DTrace Privileges to the gmb Account

The next step involves using the `gmb` account to start and stop the Web server, and also to perform the DTrace operations necessary to enumerate the privileges used by the Apache2 service.

Note – In some environments, the use of DTrace might be reserved solely for specific administrative users. In such cases, another user can be used in place of the `gmb` user. For the purpose of this example, however, it is more important that some user be able to use DTrace in order to run the `privdebug` command.

To allow the gmb user to use DTrace, the following privileges must be granted.

```
# ppriv -vl dtrace_kernel dtrace_user dtrace_proc
dtrace_kernel
   Allows DTrace kernel-level tracing.
dtrace_user
   Allows DTrace user-level tracing. Allows use of the syscall and
   profile DTrace providers to examine processes to which the user
   has permissions.
dtrace_proc
   Allows DTrace process-level tracing. Allows process-level tracing
   probes to be placed and enabled in processes to which the user has
   permissions.
```

In the following example, the three DTrace privileges are granted to the gmb user.

```
# usermod -K defaultpriv=basic,dtrace_kernel,dtrace_user,dtrace_proc gmb
```

Just as with the usermod(1M) case above, the -K option will also overwrite the existing values for the defaultpriv parameter. Therefore, it is important to reinforce the need for the basic set of privileges. Otherwise, they will no longer be available to the gmb account. Once again, visually inspect the /etc/user_attr to verify that the appropriate privileges have been assigned.

```
# grep "^gmb:" /etc/user_attr
gmb::::type=normal;defaultpriv=basic,dtrace_kernel,dtrace_user,
dtrace_proc;profiles=ApachePD
```

Alternatively, privileges can be checked when the gmb user logs into the system.

```
gmb$ id
uid=101(gmb) gid=1(other)
gmb$ ppriv -S $$
6567:  -sh
flags = <none>
    E: basic,dtrace_kernel,dtrace_proc,dtrace_user
    I: basic,dtrace_kernel,dtrace_proc,dtrace_user
    P: basic,dtrace_kernel,dtrace_proc,dtrace_user
    L: all
```

Step 5. Start a Privilege Debugging Session

Your environment is ready to start the privilege debugging session. Complete this step in a separate terminal or window so that it can be left running independently from where the Apache2 service is started or managed. This helps avoid any confusion between the output of the service (if any) and the output of the privdebug command.

The following example uses the following options.

- -n to attach to the apachectl command as it is started
- -f (follow) to follow any children spawned by the apachectl command
- -v (verbose) to obtain more context about the privileges that are used or attempted

```
gmb$ ./privdebug -f -v -n apachectl
STAT TIMESTAMP   PPID  PID  PRIV         CMD
```

Step 6. Start the Apache2 Service

With the `privdebug` tool now running, start the Apache2 service to discover which privileges it (or any of its children) attempts to use. As stated previously, run the following commands in a second terminal session or window in order to avoid confusing this output with that of the `privdebug` command.

The Apache2 service can be started with the following command (as the `gmb` user).

```
gmb$ pfexec /lib/svc/method/http-apache2 start
```

The `pfexec` command prefix is needed because, by default, the `gmb` user is not using a profile shell (such as `/bin/pfsh`). If the `gmb` user had been configured to use a profile shell, the `pfexec` prefix would not be needed. Using either the `pfexec` command prefix or a profile shell enables the `gmb` user to leverage the ApachePD rights profile to start the Apache2 service as the `webservd` user and group and with all of the privileges on the system.

Next, verify that the Apache2 service has been started using the `ps(1)` command.

```
gmb$ ps -aef | grep httpd
webservd 6619 6618  0 11:27:00 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 6621 6618  0 11:27:00 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 6618    1  0 11:26:59 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 6620 6618  0 11:27:00 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 6622 6618  0 11:27:00 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 6623 6618  0 11:27:00 ? 0:00 /usr/apache2/bin/httpd -k start
```

If the `gmb` user had also been given the `proc_owner` privilege or the `Process Management` rights profile, then the `gmb` account could also inspect the `httpd` processes to verify whether they were in fact started with the privileges defined in the ApachePD rights profile (`all`). Lacking this privilege, the following command must be run as `root` or as a user with either the `proc_owner` privilege or the `Process Management` rights profile.

```
# ppriv -S 6618
6618:  /usr/apache2/bin/httpd -k start
flags = <none>
    E: all
    I: all
    P: all
    L: all
```

The Apache2 service has now been successfully started as the `webservd` user and group, and with `all` of the privileges on the system.

Next, determine which privileges the `apachtctl` or `httpd` processes used while the Apache2 service was starting by inspecting the output of the `privdebug` command.

```
STAT TIMESTAMP         PPID  PID   PRIV           CMD
USED 1213891387930291 6603  6612  proc_fork      apachectl
USED 1213891393258070 6603  6612  proc_fork      apachectl
USED 1213891394717823 6612  6617  proc_exec      apachectl
USED 1213891459762639 6612  6617  net_privaddr   httpd
USED 1213891647640244 6612  6617  proc_fork      httpd
USED 1213892608914595 1     6618  proc_fork      httpd
USED 1213892614884001 1     6618  proc_fork      httpd
USED 1213892619276719 1     6618  proc_fork      httpd
USED 1213892623679694 1     6618  proc_fork      httpd
USED 1213892627919397 1     6618  proc_fork      httpd
```

Based on this output, three specific privileges were used: `proc_fork`, `proc_exec`, and `net_privaddr`. Note that this is exactly the privilege profile that was recommended in the Sun BluePrints article titled *Limiting Service Privileges in the Solaris 10 Operating System* (see http://www.sun.com/blueprints/0505/819-2680.html), which was written without the benefit of the `privdebug` command.

The following example shows a brief description of the privileges used.

```
# ppriv -vl proc_fork proc_exec net_privaddr
proc_fork
   Allows a process to call fork1()/forkall()/vfork()
proc_exec
   Allows a process to call execve().
net_privaddr
   Allows a process to bind to a privileged port number. The privilege
   port numbers are 1-1023 (the traditional UNIX privileged ports) as
   well as those ports marked as "udp/tcp_extra_priv_ports" with the
   exception of the ports reserved for use by NFS.
```

Step 7. Exercise the Apache2 Service

At this point, it is absolutely critical to properly exercise the Apache2 service. You want to make sure that all of the paths through the code have been followed at least once in order to verify that you have enumerated the correct set of privileges. In this specific case, you should also make sure that any programs or services called by the Apache2 service (such as CGI scripts) are also exercised so that their use of privilege can also be determined.

In fact, as a general rule, consider repeating the process of starting the service if, at times, different command-line or configuration file options will be used. Be sure to exercise both operational and administrative functions. Essentially, run the service through a typical quality assurance process in order to verify that it is working correctly and has exercised the required functionality. While exercising the service, you will be able to collect a list of privileges that can be used to further bound the service.

The following example simply verifies connecting to the Web server.

```
$ telnet localhost 80
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 14 Nov 2005 17:01:47 GMT
Server: Apache/2.0.52 (Unix) DAV/2
Content-Location: index.html.en
Vary: negotiate,accept-language,accept-charset
TCN: choice
Last-Modified: Fri, 04 May 2001 00:01:18 GMT
ETag: "22391-5b0-40446f80;223a7-961-8562af00"
Accept-Ranges: bytes
Content-Length: 1456
Connection: close
Content-Type: text/html; charset=ISO-8859-1
Content-Language: en
Expires: Mon, 14 Nov 2005 17:01:47 GMT

Connection to localhost closed by foreign host.
```

Note – Use a more comprehensive set of tests when enumerating privileges for a service that is intended for a production environment.

The process of exercising the code should normally continue until sufficient confidence demonstrates that all of the relevant code paths have been followed. During this step, be sure to review the output of the `privdebug` command for new privileges that might be used by the service.

Step 8. Stop the Apache2 Service

Just as there are privileges associated with starting a service, there might also be privileges that are used when a service is stopped or restarted. In the following example, the `gmb` user stops the Apache2 service using the same command (found in the ApachePD rights profile) that was used to start the service. The difference here is that the `stop` argument (rather than the `start` argument) is specified in the command line.

```
gmb$ pfexec /lib/svc/method/http-apache2 stop
```

Verify that the Apache2 service has been successfully stopped using the following command.

```
gmb$ pgrep httpd
gmb$
```

After confirming that the Apache2 service has been stopped, inspect the output of the `privdebug` command to determine whether any new privileges were recorded.

```
STAT TIMESTAMP         PPID  PID   PRIV             CMD
USED 1216522920650160 6635  6696  proc_fork        apachectl
USED 1216522926222381 6635  6696  proc_fork        apachectl
USED 1216522927697807 6696  6698  proc_exec        apachectl
USED 1216522988410641 6696  6698  proc_session     httpd
USED 1216522988438860 6696  6698  proc_session     httpd
```

In this example, the new instance of `apachectl` was started, which in turn executed the `httpd` process to run, which required the `proc_session` privilege. Essentially, to kill the Apache2 service, the `httpd` command is used with the `-k stop` command line argument. To verify this, refer to the source code of the shell script `/usr/apache2/bin/apachectl`.

The `proc_fork` and `proc_exec` privileges were needed to execute the `httpd` process. The `proc_session` privilege was needed so that the `httpd` process could send a signal to processes outside of its own session (the running set of `httpd` processes). For more information on `proc_session`, see the following command output.

```
# ppriv -vl proc_session
proc_session
   Allows a process to send signals or trace processes outside
   its session.
```

Step 9. Construct an SMF Method Context for the Apache2 Service

This step is necessary only if you are installing a service to be managed by SMF. Once the service is successfully exercised, you can construct an SMF method context from the enumerated privileges, which can then be used to automatically start or stop the service with the privileges that it needs.

To summarize the information obtained thus far in this section, the following privileges are needed by the Apache2 service.

- `proc_fork`, `proc_exec`, and `net_privaddr` (when started)
- `proc_fork`, `proc_exec`, and `proc_session` (when stopped)

To construct the `start/privileges` and `stop/privileges` properties for the Apache2 service SMF instance, first understand which privileges are considered `basic`.

```
# ppriv -l basic
file_link_any
proc_exec
proc_fork
proc_info
proc_session
```

Whenever using the `start` or `stop privilege` property, it is important to begin with the `basic` privilege. This allows for changes in the `basic` privilege set without needing to update the privilege profile. From this foundation, remove any privileges found in the `basic` set that are not needed (for example, those that do not appear in the enumerated list of used privileges generated by `privdebug`). Finally, add

any other non-`basic` privileges that were enumerated during this privilege debugging run. This process provides the following.

- `start/privileges =`
  `basic,!file_link_any,!proc_info,!proc_session,net_privaddr`
- `stop/privileges =`
  `basic,!file_link_any,!proc_info`

Of course, just as in the ApachePD rights profile, you must also properly set the `user` and `group` properties to `webservd`. Note that there are other non-privilege related steps that you must perform before the Apache2 service will run as a non-`root` user. For more information, refer to the Sun BluePrints article titled *Limiting Service Privileges in the Solaris 10 Operating System* at: http://www.sun.com/blueprints/0505/819-2680.html

Step 10. Validate an SMF Method Context for the Apache2 Service

This final step confirms the value of the efforts described in this section. Attempt to configure SMF to start the service using the set of privileges identified previously to see what this would look like in action. Using the above information, the Apache2 SMF instance was modified as follows.

```
# svcprop -p stop -p start apache2
start/exec astring /lib/svc/method/http-apache2\ start
start/timeout_seconds count 60
start/type astring method
start/user astring webservd
start/group astring webservd
start/privileges astring basic,!file_link_any,!proc_info,!proc_session,net_privaddr
start/limit_privileges astring :default
start/project astring :default
start/resource_pool astring :default
start/supp_groups astring :default
start/use_profile boolean false
start/working_directory astring :default
stop/exec astring /lib/svc/method/http-apache2\ stop
stop/timeout_seconds count 60
stop/type astring method
stop/user astring webservd
stop/group astring webservd
stop/privileges astring basic,!file_link_any,!proc_info
stop/limit_privileges astring :default
stop/project astring :default
stop/resource_pool astring :default
stop/supp_groups astring :default
stop/use_profile boolean false
stop/working_directory astring :default
```

The entire `start` and `stop` property groups are shown to help see exactly what was used for this example. The actual lines changed are formatted in **bold**.

Next, update the SMF repository with the changes that were just made by using the following command.

```
# svcadm refresh apache2
```

With these settings in place, verify that they work as expected. Just as before, start off with the service disabled.

```
# svcs apache2
STATE     STIME  FMRI
disabled    12:37:53 svc:/network/http:apache2
```

Next, enable the service and verify that it is actually running.

```
# svcadm -v enable -s apache2
svc:/network/http:apache2 enabled.
# svcs apache2
STATE     STIME  FMRI
online     12:38:11 svc:/network/http:apache2
```

With the service running, verify that it is running with the correct user, group, and privileges.

```
# getent passwd webservd
webservd:x:80:80:WebServer Reserved UID:/:
# getent group webservd
webservd::80:
# pcred 6816 (process ID of httpd process started by SMF)
6816:  e/r/suid=80 e/r/sgid=80
# ppriv -S 6816
6816:  /usr/apache2/bin/httpd -k start
flags = <none>
    E: net_privaddr,proc_exec,proc_fork
    I: net_privaddr,proc_exec,proc_fork
    P: net_privaddr,proc_exec,proc_fork
    L: all
```

Everything looks good thus far. The Apache2 service was started as user and group `webservd` with only the privileges that were specified previously.

Finally, disable the Apache2 service and then verify that the service was truly disabled.

```
# svcadm -v disable -s apache2
svc:/network/http:apache2 disabled.
# svcs apache2
STATE     STIME  FMRI
disabled    12:38:19 svc:/network/http:apache2
# pgrep httpd
#
```

At this point, everything is working as expected. The privileges used by the Apache2 service were successfully enumerated, and those privileges were used to start and stop the service using SMF.

Privilege Debugging the Apache2 Service—Local Zone Method

Having completed the process in a Solaris 10 Global Zone, this section proceeds to apply this knowledge to evaluating services running in a Solaris Container (called `web_svc`).

It must be noted however that while the service can be run from within a Solaris Container, the `privdebug` command must still be run from within the Global Zone because the DTrace facility is only available to the Global Zone. This is not a problem because DTrace can see and inspect processes running in Solaris

Containers. To add a touch of clarity to this section, as noted above, command prompts will be prefixed with the location from which the commands were issued, either `global` or `web_svc` (for the Global Zone and Solaris Container respectively).

The process in this section involves completing the following steps.

- Step 1. Verify the Current State of the Apache2 Service
- Step 2. Install the ApachePD Privilege Debugging Rights Profile
- Step 3. Assign the ApachePD Rights Profile to the gmb Account
- Step 4. Assign the Necessary DTrace Privileges to the gmb Account
- Step 5. Start a Privilege Debugging Session
- Step 6. Start the Apache2 Service
- Step 7. Exercise the Apache2 Service
- Step 8. Stop the Apache2 Service
- Step 9. Construct an SMF Method Context for the Apache2 Service
- Step 10. Validate an SMF Method Context for the Apache2 Service

Step 1. Verify the Current State of the Apache2 Service

This step is performed in the `web_svc` Solaris Container using the same commands that were specified in the Global Zone example (see “Step 1. Verify the Current State of the Apache2 Service” on page 8). There is no expected change in command usage or output.

Step 2. Install the ApachePD Privilege Debugging Rights Profile

This step is performed in the `web_svc` Solaris Container using the same process as in the Global Zone example (see “Step 2. Install the ApachePD Privilege Debugging Rights Profile” on page 9)—with one important difference. In place of using the keyword `all` in the privilege declaration, the keyword `zone` must be used. This is necessary because Solaris Containers do not have permission to use many of the privileges defined by the `all` keyword.

When installed, the ApachePD rights profile should look like the following example.

```
root@web_svc# grep "^ApachePD" /etc/security/prof_attr
ApachePD:::Apache Service:
root@web_svc# grep "^ApachePD" /etc/security/exec_attr
ApachePD:solaris:cmd:::/lib/svc/method/http-apache2:
uid=webservd;gid=webservd;privs=zone
```

Step 3. Assign the ApachePD Rights Profile to the gmb Account

This step is performed in the `web_svc` Solaris Container using the same commands that were specified in the Global Zone example (see “Step 3. Assign the ApachePD Rights Profile to the gmb Account” on page 9). There is no expected change in command usage or output. Remember that, for this step, the `gmb` account must exist in the Solaris Container (in addition to being in the Global Zone).

When installed, the `gmb` account in the `web_svc` Solaris Container should have the following ApachePD rights profile.

```
root@web_svc# profiles gmb
ApachePD
Basic Solaris User
All
```

Step 4. Assign the Necessary DTrace Privileges to the gmb Account

This step is performed in the Global Zone using the same commands that were specified in the Global Zone example (see “Step 4. Assign the Necessary DTrace Privileges to the gmb Account” on page 10). There is no expected change in command usage or output. Remember that, for this step, the `gmb` account must exist in the Global Zone (in addition to being in the Solaris Container).

When installed, the `gmb` account in the Global Zone should have the following DTrace privileges assigned.

```
root@global# grep "^gmb:" /etc/user_attr
gmb::::type=normal;defaultpriv=basic,dtrace_kernel,dtrace_user,
dtrace_proc
```

Step 5. Start a Privilege Debugging Session

This step is performed in the Global Zone using the same command that was specified in the Global Zone example above (see “Step 5. Start a Privilege Debugging Session” on page 11). Because this example seeks to monitor the Apache service being run inside of a Solaris Container, a new `-z` command line option is used to instruct the `privdebug` command to look only at events happening within the specified zone. There is no expected change in command output.

To initiate the privilege debugging session, use the following command:

```
gmb@global$ ./privdebug -f -v -n httpd -z web_svc
STAT TIMESTAMP   PPID  PID  PRIV         CMD
```

Step 6. Start the Apache2 Service

This step is performed in the `web_svc` Solaris Container using the same process as in the Global Zone example (see “Step 6. Start the Apache2 Service” on page 12)—with one important difference. When the Apache2 service is started, instead of running with all privileges, it will run with privileges defined by the `zone` keyword. Just as in the previous example, the service is started using the following command.

```
gmb@web_svc$ pfexec /lib/svc/method/http-apache2 start
```

When running, the Apache2 service should look like the following example.

```
root@web_svc# ps -aef | grep httpd
webservd 1195 1193  0 10:48:37 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 1197 1193  0 10:48:37 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 1198 1193  0 10:48:37 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 1193  672  0 10:48:36 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 1196 1193  0 10:48:37 ? 0:00 /usr/apache2/bin/httpd -k start
webservd 1194 1193  0 10:48:37 ? 0:00 /usr/apache2/bin/httpd -k start
root@web_svc# pcred 1193
1193:  e/r/suid=80 e/r/sgid=80
 groups: 1
root@web_svc# ppriv -S 1193
1193:  /usr/apache2/bin/httpd -k start
flags = <none>
    E: zone
    I: zone
    P: zone
    L: zone
```

In this case, no changes to the privilege profile are expected. Just as in the Global Zone example, the required privileges include `proc_exec`, `proc_fork`, and `net_privaddr`.

Step 7. Exercise the Apache2 Service

This step is performed in the `web_svc` Solaris Container using the same commands that were specified in the Global Zone example (see “Step 7. Exercise the Apache2 Service” on page 13). There is no expected change in command usage or output.

Step 8. Stop the Apache2 Service

This step is performed in the `web_svc` Solaris Container using the same process as in the Global Zone example (see “Step 8. Stop the Apache2 Service” on page 14)—with one important difference. When the Apache2 service is started, instead of running with all privileges, it will be running with privileges defined by the `zone` keyword. Just as in the previous example, the service is stopped using the following command.

```
gmb@web_svc$ pfexec /lib/svc/method/http-apache2 stop
```

In this case, no changes to the privilege profile are expected. Just as in the Global Zone example, the required privileges include `proc_exec`, `proc_fork`, and `proc_session`.

Step 9. Construct an SMF Method Context for the Apache2 Service

This step is performed in the `web_svc` Solaris Container using the same commands that were specified in the Global Zone example (see “Step 9. Construct an SMF Method Context for the Apache2 Service” on page 15). There is no expected change in command usage or output.

Step 10. Validate an SMF Method Context for the Apache2 Service

This step is performed in the `web_svc` Solaris Container using the same commands that were specified in the Global Zone example (see “Step 10. Validate an SMF Method Context for the Apache2 Service” on page 16). There is no expected change in command usage or output.

Conclusion

This article has discussed the concept of privilege debugging and shown how such a process could be implemented in the Solaris 10 OS using the `privdebug` command, which is built upon Sun's Dynamic Tracing (DTrace) facility. This process can be applied to both the Solaris 10 Global Zone as well as to any Solaris Containers on the system.

Using the privilege debugging approach defined in this article, you can more readily understand and enumerate the privileges used by programs and services. With this knowledge, you can further restrict those programs and services to have only the privileges that they need—and no more. Implementing the principle of least privilege in this manner can help better protect your systems and limit the adverse impact should any of those programs or services be vulnerable or be exploited. Remember that this method works only if you can sufficiently exercise the required functionality of all the processes involved in providing the service.

Alternatively, the list of privileges can be discovered by analysis of the source code, looking for all places where system calls are made, such as those `libc` and related functions that are described in Section 2 of the Solaris manual pages (see http://docs.sun.com/app/docs/doc/816-5167).

It is not always possible, however, nor is it always obvious which privileges might be needed when the system call is made. Further, it is important to understand that automated enumeration and analysis of privileges used by an application is not fool-proof. It is recommended that you always question the use of privilege attempted by your applications before putting the system into production. Poorly written applications, for example, can attempt to use privileges that they do not need or should not have. A combination of both source code (where possible) and runtime analysis, sanity checked by a human expert, is often the best approach.

References

Sun BluePrints

- Limiting Service Privileges in the Solaris 10 Operating System

  http://www.sun.com/blueprints/0505/819-2680.pdf

- Restricting Service Administration in the Solaris 10 Operating System

  http://www.sun.com/blueprints/0605/819-2887.pdf

Sun Product Documentation and Manual Pages

- Solaris 10 Privileges Overview

  http://docs.sun.com/app/docs/doc/816-4557/6maosrjfj?a=view

- `attributes(5)`

  http://docs.sun.com/app/docs/doc/816-5175/6mbba7evc?q=attributes%285%29&a=view

- `exec(2)`
  http://docs.sun.com/app/docs/doc/816-5167/6mbb2jafk?a=view
- `fork(2)`
  http://docs.sun.com/app/docs/doc/816-5167/6mbb2jag7?a=view
- `mdb(1)`
  http://docs.sun.com/app/docs/doc/817-0689/6mgfkpctk?a=view
- `pfexec(1)`
  http://docs.sun.com/app/docs/doc/816-5165/6mbb0m9o5?a=view
- `ppriv(1)`
  http://docs.sun.com/app/docs/doc/816-5165/6mbb0m9p2?a=view
- `privileges(5)`
  http://docs.sun.com/app/docs/doc/816-5175/6mbba7f3a?a=view
- `profiles(1)`
  http://docs.sun.com/app/docs/doc/816-0210/6m6nb7mi9?a=view
- `ps(1)`
  http://docs.sun.com/app/docs/doc/816-5165/6mbb0m9pj?q=ps&a=view
- `usermod(1)`
  http://docs.sun.com/app/docs/doc/816-5166/6mbb1kqjs?a=view

Other Documentation, Articles and Web Sites

- `privdebug` command
  `http://www.opensolaris.org/os/community/security/files/`
- Sun Security Portal
  http://www.sun.com/security/
- The Least Privilege Model in the Solaris 10 OS
  http://www.sun.com/bigadmin/features/articles/least_privilege.html

About the Authors

Glenn Brunette

Glenn Brunette is a Sun Distinguished Engineer with nearly 15 years' experience in information security. Glenn currently works in Sun's Client Solutions CTO as the Director and Chief Architect of the CSO Security Office. In this role, Glenn is responsible for global security strategy and architecture, security-

focused collaboration and knowledge sharing, as well as improving the quality and security of products and services delivered to Sun's customers.

Glenn is the driving force behind Sun's Systemic Security approach and is also an OpenSolaris Security Community Leader, the co-founder of the Solaris Security Toolkit software, and a frequent author, contributor, and speaker at both Sun and industry events. Externally, Glenn has served as the Vice-Chair of the Enterprise Grid Alliance Grid Security Working Group and Working Group Champion for the National Cyber Security Partnership's Technical Standards and Common Criteria Task Force. Finally, Glenn is an active contributor to the Center for Internet Security's Unix Benchmark team. Glenn is a Certified Information Systems Security Professional (CISSP) and has been trained in the National Security Agency's INFOSEC Assessment Methodology (IAM).

Darren Moffat
Darren Moffat is a Senior Staff Engineer in the Solaris Security Technologies Group (part of Solaris Software). Darren has been involved with the design and development of security technologies that form the core of the Solaris OS. Darren started with Sun UK in December, 1996, working in Enterprise Services (then SunService) doing Trusted Solaris and general OS security support. He joined sustaining engineering in Solaris Software for the NFS/Naming products, then moved to California to join the development teams working on Solaris security. Before joining Sun, Darren worked as an analyst/ programmer for the UK Ministry of Defence. He is a graduate of the Computing Science Department of Glasgow University in Scotland.

Acknowledgements

The authors would like to thank Bart Blanquart, Casper Dik, and Scott Rotondo for their inspiration, technical feedback, and overall support in the development of this article.

Ordering Sun Documents

The SunDocs℠ program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals through this program.

Accessing Sun Documentation Online

The `docs.sun.com` Web site enables you to access Sun technical documentation online. You can browse the `docs.sun.com` archive or search for a specific book title or subject at `http://docs.sun.com/`.

To reference Sun BluePrints OnLine articles, visit the Sun BluePrints OnLine Web site at:
`http://www.sun.com/blueprints/online.html`