PRIVILEGE BRACKETING IN THE SOLARIS™ 10 OPERATING SYSTEM

Glenn Brunette, Security Program Office, Client Solutions

Sun BluePrints™ OnLine — April 2006

819-6320-10
Revision 1.0, 3/29/06
Edition: April 2006

Please
Recycle

TABLE OF CONTENTS

Privilege Bracketing in the Solaris™ 10 Operating System

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] This Sun BluePrints™ OnLine article describes how to use the Process Rights Management feature of the Solaris™ 10 Operating System (Solaris 10 OS) to implement this principle for any given software program.

Process Rights Management allows software developers to write privilege-aware programs that run with only the privileges they need, dropping those that are not needed or are no longer required. Further, using a programming technique called *privilege bracketing*, a developer can control exactly when a privilege or set of privileges is active or in effect.

Software developers can use the privilege bracketing technique to ensure that a program is running with privilege only when that privilege is required. This is accomplished by placing privileged software operations between code that effectively enables and disables specific privileges. Using the methods described in this article, software developers will be able to develop privileged programs that are more secure and resilient to flaws because the use of privilege within the code can be more tightly controlled.

This article contains the following sections:

Note – This Sun BluePrints OnLine article relies on the description of Process Rights Management and related concepts that were provided in the Sun BluePrints article titled *Privilege Debugging in the Solaris 10 Operating System* (Brunette, Glenn and Moffat, Darren, February 2006), which is available at http://www.sun.com/blueprints/0206/819-5507.pdf.

---

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

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.

Process Rights Management

Introduced in the Solaris 10 OS, Process Rights Management replaces the dependence on a special UID (for example, UID 0), using instead nearly 50 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.

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, enabling 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 local file on the system, regardless of its ownership or permissions. Similarly, the `net_privaddr` privilege is used to allow a process to bind and listen on a privileged network port (that is, one whose port number is between 1 and 1023). For an entire listing of privileges, see `privileges(5)` or the output of the `ppriv -vl` command.

Privilege Keywords

There are three special privilege keywords: `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. By default, these `basic` privileges are assigned to every process, although they can be taken away if they are not needed.

Privilege Sets

A privilege set is simply a mechanism for grouping together a collection of privileges. Every process has the following four distinct privilege sets assigned that collectively determine which privileges are currently in effect for the process, as well as which are available to the process and its children.

Privilege Set	Description
`Permitted` (or `P`)	Contains all of the privileges that the process is permitted to use. This represents the maximum or complete set of privileges for a given process.
`Effective` (or `E`)	Contains those privileges from the `Permitted` set that are currently in effect.
`Inheritable` (or `I`)	Contains those privileges that will be passed on to child processes upon the use of the `exec(2)` family of system calls.
`Limit` (or `L`)	Upper bound of the privileges that a process and its children can obtain. Changes to this set take effect upon the next `exec(2)` call.

To illustrate the concept of privilege sets, consider the following example, in which the privilege sets associated with the `rpcbind` process are displayed using the `ppriv(1)` command.

```
# ppriv -S `pgrep rpcbind`
260:    /usr/sbin/rpcbind
flags = PRIV_AWARE
        E: net_privaddr,proc_fork,sys_nfs
        I: none
        P: net_privaddr,proc_fork,sys_nfs
        L: none
```

In this example, the `rpcbind` process is running with three privileges (`net_privaddr`, `proc_fork`, and `sys_nfs`). These privileges are the only ones that the `rpcbind` process is able to use because its `Permitted` privilege set is equal to its `Effective` set. Note that the `rpcbind` process is not permitted to use the `proc_exec` privilege, which means that it is not able to use the `exec(2)` family of system calls. Because the `Permitted` and `Limit` sets take effect upon the use of an `exec(2)` call, there is no reason for those sets—in this example, anyway—to contain any privileges for this process. For more information on these privilege sets, their relationship to one another, and their use, refer to the `privileges(5)` manual page.

Note – See “References” on page 12 for posts, articles, and resources that discuss the new Process Rights Management capability in the Solaris 10 OS. Most of this material focuses on describing what this new privilege model is, why it is useful, and how it can be applied using mechanisms such as the Solaris Role-based Access Control (RBAC) facility, the Service Management Facility (SMF), and individual tools such as `ppriv(1)`.

Making Programs Privilege Aware

Consider how Process Rights Management can be valuable from the viewpoint of a software developer. A developer could modify a program to become privilege aware so that it runs with only the privileges that it needs, and activates specific privileges only when it needs them. Privilege bracketing ensures that the program is running with privilege only when privilege is needed. A developer "brackets" the code by enabling the privilege just before it is needed, and disabling the privilege immediately after the privileged functions complete, thereby providing clear boundaries within which privileges can be used.

This section shows how to change a program so that it:

- Drops any privileges that it will never need.
- Disables any privileges not immediately needed.
- Enables the remaining privileges only when it needs them (disabling them when they are no longer required).
- Relinquishes the use of privileges when they are no longer needed at all by the rest of the program.

Note – To make use of Solaris privileges, you do not need to modify the source code of programs. Programs can be executed with an additional or a reduced set of privileges using RBAC, SMF, or `ppriv`. Developing programs to be privilege aware, however, gives developers more fine-grained control over when privileges are enabled, used, and relinquished.

Using the ping Program for this Example

To illustrate the concept of making programs privilege aware, this section uses the OpenSolaris version of the `/usr/sbin/ping` program, a `set-id` program, which was chosen for the following reasons:

- The `ping(1M)` source code is simple and straightforward to read and understand.
- All of the changes needed to make it privilege aware were contained in one file.

In this article, the term `set-id` refers to files that are either `set-uid` or `set-gid`. The `ping` program is `set-uid` (to "`root`"), which means that when the `ping` program is run (by any user), the actual `ping` code is executed with the privileges of the `root` user. Without privilege bracketing, the `ping` code could potentially run with any privilege available to `root`—which is all of them.

Note – There is nothing special about `ping` with respect to privileges and privilege bracketing. The same techniques described in this section could be applied to other programs, whether they are `set-id` or not. In the case of a non-`set-id` program, privilege bracketing can be used to drop privileges granted via the "basic" set, which are granted to every process (by default) on startup. For example, suppose a program ran without privilege, but a developer did not want to allow that program to be able to execute other programs. To do this, the developer could simply drop the `proc_exec` privilege.

In February 2003, before the `ping` command was made privilege aware, `ping` was a `set-uid` root program that controlled its use of privilege using the `seteuid(2)` function. When the program started as `root`, it quickly set its effective UID to the UID of the calling user to run with less privilege. When it came time to execute a privileged operation, the code issued another `seteuid` call that reset its EUID to `root` so that the privileged operation could succeed.

With the introduction of Process Rights Management in the Solaris 10 OS, this model was no longer needed. Rather than executing code as EUID 0, specific privileges are used to define the types of privileged operations that are permitted. This is a major advance because, in the Solaris 10 OS, privilege-aware programs will run with only the privileges that they need, exactly when they need them.

Note – Line numbers in code samples represent a snapshot of the code as of the writing of this article. They are meant to provide a relative reference only. Line numbers will likely change over time when modifications are made to the OpenSolaris source code.

Privilege Bracketing Using Private Header Files and Functions

This section examines the OpenSolaris `ping.c` source code and walks through the steps of making the `ping` program privilege aware using private header files and functions—the term “private” referring to the stability of the interfaces (for example, functions) used by Sun to solve this problem. In this particular case, the functions used were private to the Solaris ON (Operating System and Networking) software consolidation and were not intended for use by software developers working on code outside of that consolidation. For an example of how to more generally solve this same problem using public interfaces, see “Privilege Bracketing Using Public Header Files and Functions” on page 8. For information about interface stability classifications, see `attributes(5)`.

Specifying the Header File for Private Interfaces

To make a program privilege aware, begin by including a new header that declares functions and defines constants used by the privilege manipulation functions that will be used later in this section.

```
72    #include <priv_utils.h>
```

The `priv_utils.h` file is posted on the OpenSolaris Web site at http://cvs.opensolaris.org/source/xref/usr/src/head/priv_utils.h.

Initializing the Program's Privileges

Next, in the `main` function, initialize the program’s privileges, dropping all of the privileges that are not needed, as shown in the following code.

```
247         /*
248          * This program needs the net_icmpaccess privileges.  We'll fail
249          * on the socket call and report the error there when we have
250          * insufficient privileges.
251          */
252         (void) __init_suid_priv(PU_CLEARLIMITSET, PRIV_NET_ICMPACCESS,
253             (char *)NULL);
```

Remember that `ping` is still `set-uid root`, which means that, when it is started, it will have the privileges that have been assigned to `root` (which is, by default, `all`). This is necessary in order to preserve backward compatibility.

The purpose of the `__init_suid_priv` function, as described in `priv_utils.h` code, is to do the following (actual source code comments are provided):

```
48 /*
49  * Should be run at the start of a set-uid root program;
50  * if the effective uid == 0 and the real uid != 0,
51  * the specified privileges X are assigned as follows:
52  *
53  * P = I + X + B (B added insofar allowable from L)
54  * E = I
55  * (i.e., the requested privileges are dormant, not active)
56  * Then resets all uids to the invoking uid; no-op if euid == uid == 0.
57  *
58  * flags: PU_LIMITPRIVS, PU_CLEARLIMITSET, PU_CLEARINHERITABLE
59  *
60  * Caches the required privileges for use by __priv_bracket().
61  *
62  */
```

The `__init_suid_priv` function, as used in the `ping.c` code, does the following tasks:

- Resets the real and effective UID of the `ping` process to that of the calling user so that it is no longer running as `root`.
- Clears the limit set of the `ping` process, which means that any children spawned by this process will themselves have no privileges.
- Adds the `net_icmpaccess` privilege to the permitted set of the `ping` process so that it can be enabled and used when necessary. Consequently, the four privilege sets (`Effective`, `Inheritable`, `Permitted`, and `Limit`) of the `ping` process will look like the following example.

```
# ppriv -S `pgrep ping`
14527:  ping localhost
flags = PRIV_AWARE
        E: basic
        I: basic
        P: basic,net_icmpaccess
        L: none
```

For more information on each of these privilege sets, refer to “Privilege Sets” on page 3. For additional background information, refer to the Sun Process Rights Management Tutorial at http://iforce.sun.com/protected/solaris10/adoptionkit/tech/least/tutorial.html.

According to `ppriv`, the `net_icmpaccess` privilege is used to allow a process to send and receive ICMP packets.

```
$ ppriv -lv net_icmpaccess
net_icmpaccess
        Allows a process to send and receive ICMP packets.
```

With the changes made to `ping` thus far, rather than having the potential to access all of `root`'s power, a `ping` process will now run as the calling user's UID with a single (non-basic) privilege that allows the sending or receiving of ICMP packets.

In addition, `priv_utils.h` offers more instruction as to how to proceed.

```
65 /*
66  * After calling __init_suid_priv we can __priv_bracket(PRIV_ON) and
67  * __priv_bracket(PRIV_OFF) and __priv_relinquish to get rid of the
68  * privileges forever.
69  */
```

Implementing Privilege Bracketing Around Privileged Operations

Recall that, before `ping` was made privilege aware, it used `seteuid` to control when its privileges were in effect. This was necessary for the program to run in a privileged capacity only when it needed to execute privileged operations. In the new model, the `ping` process leverages privilege bracketing to control when a privilege (listed in the permitted privilege set of the process) is made effective.

To see privilege bracketing in action, take a look at the following code, which appears in the `setup_socket` function in `ping.c`.

```
1196         /* now we need the net_icmpaccess privilege */
1197         (void) __priv_bracket(PRIV_ON);
1198
1199         recv_sock = socket(family, SOCK_RAW,
1200             (family == AF_INET) ? IPPROTO_ICMP : IPPROTO_ICMPV6);
1201
1202         if (recv_sock < 0) {
1203                 Fprintf(stderr, "%s: socket %s\n", progname, strerror(errno));
1204                 exit(EXIT_FAILURE);
1205         }
1206
1207         /* revert to non-privileged user after opening sockets */
1208         (void) __priv_bracket(PRIV_OFF);
```

As you can see, the `__priv_bracket` function is used around the privileged operation—in this case, the `socket(2)` call—to control whether the instructions are executed with privilege. This is one form of privilege bracketing that enables and disables all of the privileges cached by the `__init_suid_priv` function invoked earlier in the program (and described in “Initializing the Program's Privileges” on page 5). There are other privilege manipulation functions available to allow more fine-grained control if needed. To learn more, refer to the Solaris 10 product documentation—”Developing Privileged Applications” in the *Solaris Security for Developer’s Guide* at:

http://docs.sun.com/app/docs/doc/816-4863/6mb20lvf9?a=view

Relinquishing Privileges when No Longer Needed

The final task is to relinquish privileges when you are certain that they are no longer needed.

```
602         __priv_relinquish();
```

The `__priv_relinquish` function is called after the `setup_socket` function has completed in `main`. Because the program will no longer need the `net_raw_icmpaccess` privilege (the only non-basic

privilege available to the process), it can now be safely dropped by calling the `__priv_relinquish` function.

Note – Once a privilege is relinquished, it will no longer be available to the process. If a privilege might be needed later in the program, then simply disable it (removing it from the `Effective` privilege set of the process) and use functions described later in this article (`priv_delset` and `setppriv`) instead. Disabling the privilege effectively takes away that privilege but does not relinquish it entirely so that, if the privilege is needed later, it can simply be added back to the `Effective` set (using `priv_addset` and `setppriv`).

Privilege Bracketing Using Public Header Files and Functions

The privilege functions and header file described in “Privilege Bracketing Using Private Header Files and Functions” on page 5 are private to Solaris and, more specifically, to the ON [OS and Networking] consolidation. Therefore, the approach described above will work just fine if you are modifying OpenSolaris `set-id` programs such as `atq`, `atrm`, `traceroute`, `lpstat`, and the like. For details on Solaris consolidations, see http://blogs.sun.com/roller/page/kupfer?entry=the_hitchhiker_s_guide_to.

What if you are developing programs for another consolidation or something that is entirely external to OpenSolaris? Can you still implement privilege bracketing? Absolutely!

In addition to the ON private header files and functions discussed above, there is also an available set of public header files and functions. For the sake of comparison, this section describes how to adapt the privilege-aware version of `ping.c` (as described in “Privilege Bracketing Using Private Header Files and Functions” on page 5) to use the public versions of the privilege manipulation header files and functions. Doing so accomplishes the same result—namely, making the program privilege aware and implementing bracketing around the privileged operations.

Note – By way of convention, this section describes only the changes that need be implemented to convert the `ping.c` program modified earlier to use the new public header files and functions.

Specifying the Header File for Public Interfaces

The first thing to do is change the header file. To use the public interfaces, be sure to include `priv.h` and *not* `priv_utils.h`.

```
72c72
< #include <priv_utils.h>
---
> #include <priv.h>
```

Defining a Convenience Function to Initialize the Program's Privileges

Next, go to the section of the code that configured the privilege sets at the start of the program. Recall that this was done in order to drop any privileges that were never needed, and disable those that were left but not needed right now. This was originally accomplished using the `__init_suid_priv` function, which

provided a convenient wrapper for that functionality. Unfortunately, that function is private, so rather than a single line, a little more work is required.

To make the code easier to follow for this article, a new function (`setup_privs`) was created to handle the initial privilege operations. The `setup_privs` function handles the majority of the work originally done by the `__init_suid_priv` function.

```
225a226
> static priv_set_t *setup_privs(void);
227a229,298
>  * setup_privs()
>  */
> priv_set_t *
> setup_privs(void)
> {
>     priv_set_t *pPrivSet = NULL;
>     priv_set_t *lPrivSet = NULL;
>
>     /*
>      * Start with the 'basic' privilege set and then remove any
>      * of the 'basic' privileges that will not be needed by this
>      * process.  The 'net_icmpaccess' privilege will be added
>      * since we know that we will need it for the permitted set.
>      */
>
>     if ((pPrivSet = priv_str_to_set("basic", ",", NULL)) == NULL) {
>             perror("priv_str_to_set");
>             return (NULL);
>     }
>
>     /*
>      * Let's clear all of the privileges we know we will not
>      * need from the 'basic' set.
>      */
>
>     (void) priv_delset(pPrivSet, PRIV_FILE_LINK_ANY);
>     (void) priv_delset(pPrivSet, PRIV_PROC_EXEC);
>     (void) priv_delset(pPrivSet, PRIV_PROC_FORK);
>     (void) priv_delset(pPrivSet, PRIV_PROC_INFO);
>     (void) priv_delset(pPrivSet, PRIV_PROC_SESSION);
>
>     /* Next add the known required privilege, 'net_icmpaccess' */
>
>     (void) priv_addset(pPrivSet, PRIV_NET_ICMPACCESS);
>
>     /* Set the permitted privilege set. */
>
>     if (setppriv(PRIV_SET, PRIV_PERMITTED, pPrivSet) != 0) {
>             perror("setppriv(PRIV_SET, PRIV_PERMITTED)");
>             return (NULL);
>     }
>
>     /* Ensure that the 'net_icmpaccess' privilege is off by default. */
>
>     if (priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_NET_ICMPACCESS,
>             NULL) != 0) {
>             perror("priv_set(PRIV_OFF, PRIV_EFFECTIVE)");
>             return (NULL);
```

```
>     }
>
>     /* Clear the limit set. */
>
>     if ((lPrivSet = priv_allocset()) == NULL) {
>             perror("priv_allocset");
>             return (NULL);
>     }
>
>     priv_emptyset(lPrivSet);
>
>     if (setppriv(PRIV_SET, PRIV_LIMIT, lPrivSet) != 0) {
>             perror("setppriv(PRIV_SET, PRIV_LIMIT)");
>             return (NULL);
>     }
>
>     priv_freeset(lPrivSet);
>
>     return (pPrivSet);
> }
>
> /*
```

Why go through the exercise of starting with the basic set of privileges and removing them all? Why not just start out with no privileges and simply add those that are needed? The answer lies in the fact that the basic privilege set in Solaris is not intended to be static—over time, additional non-administrative privileges might be added to future Solaris OS versions. If the program was started with no privileges, then it might fail because in some future Solaris version it might need a privilege for an operation that previously did not require one. In essence, this is a way of future-proofing code.

Initializing the Program's Privileges

Now that a function has been defined to help mimic most of the behavior of `__init_suid_priv`, consider how this function might be used in the following example:

```
243a315
>     priv_set_t *privSet = NULL;
252,253d323
<     (void) __init_suid_priv(PU_CLEARLIMITSET, PRIV_NET_ICMPACCESS,
<         (char *)NULL);
254a325,337
>     if ((privSet = setup_privs()) == NULL) {
>             exit(EXIT_FAILURE);
>     }
>
>     /*
>      * Reset the real and effective UIDs for this process.
>      */
>
>     if (setreuid(getuid(), getuid()) != 0) {
>             perror("setreuid");
>             exit(EXIT_FAILURE);
>     }
>
```

Note how the `__init_suid_priv` call has been replaced by calls to both `setup_privs` and `setreuid(2)` functions. These changes are straightforward because all of the work was done in the

`setup_privs` function. The code captures the privilege set parameter (`privSet`) because it will be needed later in the code when it comes time to relinquish privileges.

Relinquishing Privileges when No Longer Needed

Continue linearly down the code to see where other replacements are needed. At this point, all of the hard work is over. The rest of the changes needed to complete the conversion from private to public privilege manipulation functions are trivial. The following example shows the next replacement.

```
602c685,687
<     __priv_relinquish();
---
>     /*
>      * Clear the permitted set of the 'net_icmpaccess' privilege.
>      */
603a689,696
>     (void) priv_delset(privSet, PRIV_NET_ICMPACCESS);
>
>     if (setppriv(PRIV_SET, PRIV_PERMITTED, privSet) != 0) {
>             perror("setppriv(PRIV_PERMITTED)");
>             exit(EXIT_FAILURE);
>     }
>     priv_freeset(privSet);
>
```

In this code, the `__priv_relinquish` function is replaced with a call to `setppriv(2)`. For this change to work, however, the `net_icmpaccess` privilege must be removed from `privSet` using the `priv_delset` function. Remember that `privSet` (returned from the `setup_privs` function used earlier in the code) contains the privileges from the basic privilege set that had not already been dropped, as well as the `net_icmpaccess` privilege. By removing the `net_icmpaccess` privilege in the above code example, `setppriv` will effectively drop the last privilege use by the program, thereby relinquishing all of the program's privileges. This is because all of the basic privileges typically granted to processes had been dropped in the `setup_privs` function at the start of the program. The only privilege that had remained was `net_icmpaccess`. Now that the privilege has been dropped, the program will run with no privileges.

Note – Future versions of the Solaris OS might add new non-administrative privileges to the basic privilege set. If this is done, the `setup_privs` code above will need to be reviewed and possibly modified in order to ensure that all unused privileges have been dropped. Otherwise, the `ping` program would run with those new privileges in effect.

Implementing Privilege Bracketing Around Privileged Operations

The next section of replacement code is where the bracketing of the `net_icmpaccess` privilege is enforced. In this case, the call to `__priv_bracket` is replaced with a call to the `priv_set(3C)` function. The `priv_set` function is called with the `PRIV_ON` parameter, which enables the `net_icmpaccess` privilege in the effective privilege set of the process.

```
1197c1290,1293
<     (void) __priv_bracket(PRIV_ON);
---
>     if (priv_set(PRIV_ON, PRIV_EFFECTIVE, PRIV_NET_ICMPACCESS, NULL) != 0) {
>             perror("priv_set(PRIV_ON, PRIV_EFFECTIVE)");
>             exit(EXIT_FAILURE);
>     }
```

Similarly, the companion instance of `__priv_bracket` is replaced with another call to `priv_set` (once the privileged operations are complete) to remove the `net_icmpaccess` privilege from the effective privilege set of the process, therefore completing the bracketing of privilege.

```
1208c1304,1308
<     (void) __priv_bracket(PRIV_OFF);
---
>     if (priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_NET_ICMPACCESS,
>             NULL) != 0) {
>             perror("priv_set(PRIV_OFF, PRIV_EFFECTIVE)");
>             exit(EXIT_FAILURE);
>     }
```

That is all there is to it. As you can tell, there is a bit more work in the initial setup of privilege sets for a process. However, once complete, the use of the public privilege manipulation functions is straightforward.

Conclusion

With the information contained in this article (supplemented by the Solaris 10 product documentation, the OpenSolaris community, and the “References” on page 12), you should have more than enough information to get started converting your own programs to become privilege aware. Keep in mind that this capability is useful for all types of processes—not just `set-id` programs. Even “unprivileged” programs can be made to be privilege aware, limiting which basic privileges they can use, and when they can use them.

References

- Casper Dik's WebLog – Solaris Privileges

  http://blogs.sun.com/casper/20040722

- Sun BigAdmin Xpert Session – Process Rights Management in the Solaris 10 OS

  http://www.sun.com/bigadmin/xperts/sessions/16_prm/

- Sun Process Rights Management Tutorial

  http://iforce.sun.com/protected/solaris10/adoptionkit/tech/least/tutorial.html

- OpenSolaris `ping.c` Source Code

  http://cvs.opensolaris.org/source/xref/usr/src/cmd/cmd-inet/usr.sbin/ping/ping.c

- OpenSolaris `priv_utils.h` Source Code

  http://cvs.opensolaris.org/source/xref/usr/src/head/priv_utils.h

- OpenSolaris `priv.h` Source Code
  http://cvs.opensolaris.org/source/xref/on/usr/src/head/priv.h
- Solaris 10 `attributes(5)` Manual Page
  http://docs.sun.com/app/docs/doc/816-5175/6mbba7evc?a=view
- Solaris 10 `exec(2)` Manual Page
  http://docs.sun.com/app/docs/doc/816-5167/6mbb2jafk?a=view
- Solaris 10 `ping` Manual Page
  http://docs.sun.com/app/docs/doc/816-5166/6mbb1kqbe?a=view
- Solaris 10 `ppriv(1)` Manual Page
  http://docs.sun.com/app/docs/doc/816-5165/6mbb0m9p2?a=view
- Solaris 10 `privileges(5)` Manual Page
  http://docs.sun.com/app/docs/doc/816-5175/6mbba7f3a?a=view
- Solaris 10 `seteuid(2)` Manual Page
  http://docs.sun.com/app/docs/doc/816-5167/6mbb2jakd?a=view
- Solaris 10 `setppriv(2)` Manual Page
  http://docs.sun.com/app/docs/doc/816-5167/6mbb2jakk?a=view

About the Author

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).

Acknowledgements

The author would like to thank the following people for their inspiration, technical feedback, and overall support in the development of this article: Casper Dik, Darren Moffat, Joep Vesseur, and Mark Thacker.

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`