

Restricting file access in OpenSolaris™ by dropping basic privileges

Johannes Nicolai

August 28, 2006

Abstract

During Google Summer of Code 2006, students got paid for developing software for a bigger Open Source organization. *Expanding the basic privilege set* was one project proposal of the OpenSolaris¹ community I applied and was accepted for. This article provides the background knowledge to understand my contributions to the OpenSolaris kernel. After remembering the least privilege principle and its implementation in traditional Unices and Solaris 10, I describe the new introduced basic privileges in detail and present some thoughts about future extensions. Later on, examples of how to use these privileges in order to better control file access are provided. Finally, the article gives an overview about the concrete code changes in the OpenSolaris kernel and how to adapt file system drivers to use them.

Contents

1 Introduction: Google Summer of Code, OpenSolaris and intentions of this article	2
2 The least privilege access principle: Minimizing the impact of exploits	3
3 Traditional Unix Behaviour: All or nothing	5
4 Solaris 10 privileges: Coming closer to the least privilege access principle	8
5 Proposed file access behaviour of OpenSolaris: Google summer of code contributions to the basic privilege set	13
6 The future: What functionality to expect next in the basic privilege set	21

¹OpenSolaris is a trademark of Sun Microsystems.

A	How to use the new basic privileges: Confining ssh-agent	23
B	Overview of the new introduced kernel functions: secpolicy_vnode_gen_* and secpolicy_vnode_nanon_*	28
C	Porting guide: How to use the new kernel functions in a file system driver with the UFS implementation as example	31
D	Sources: Where to find further information	37

1 Introduction: Google Summer of Code, OpenSolaris and intentions of this article

In summer 2006, Google announced its successful Summer of Code program for the second time. Google Summer of Code (<http://code.google.com/soc/>) gives students the opportunity to work for a bigger Open Source project in the summer and get paid for it. Students are welcome to apply for project proposals of participating Open Source projects that will chose the best applications and give the applicants the chance of working on problems that are likely to be solved within three months. During that period, accepted students are mentored by a person affiliated with the Open Source project (also called mentoring organization).

The OpenSolaris community was given the opportunity by Google to be supported by two Summer of Code students and therefore provided project proposals. I applied for the task *Expanding the basic privilege set* and was accepted by Google and the OpenSolaris community. My project was perfectly mentored by Darren Moffat, senior programmer at Sun Microsystems and one leader of the OpenSolaris security community. I like to thank Darren at this place because he established my contacts in the OpenSolaris community, gave a lot of hints how to cope with kernel related tools and technologies and always answered my numerous questions.

My project introduced new 'basic' privileges that enable better control over how processes may access resources. At the beginning I focused on restricting inbound and outbound network access but early found out with the help of the community that this may be done much better with an application based network filter tool². So I concentrated my work on controlling file access of processes.

Currently, it is not possible in OpenSolaris to restrict a process to only access world readable/writable/executable files. This behavior is wished for processes that require access to global libraries and configuration files, but require none of the following:

1. Reliance on their associated user ID.

²See <http://www.opensolaris.org/jive/thread.jspa?threadID=9766&tstart=75> for a discussion on the network-discuss mailing list

2. Reliance on their associated group membership.
3. Working with files that have limited access (cannot be accessed by everybody).

There are even situations a process does not need to access any file after starting up (like ssh-agent, see appendix A) or do not need read/write/execute permissions at all. So a possibility to switch file access completely or partially off (e. g. read only processes) is wished as well.

To support all these scenarios, I introduced nine new basic privileges in the OpenSolaris kernel after exchanging thoughts with the OpenSolaris community³ and ported some file systems to use them. All changes may be reviewed on <http://myhpi.de/~nicolai/webrev/>

This article intends to do the following

- Explain why restricting the power of processes is that important (section 2).
- Give an overview about the traditional way how to restrict processes (section 3).
- Introduce the main ideas of the privilege model in OpenSolaris (section 4).
- Present the new introduced basic privileges to restrict file access in detail (section 5).
- Outline expected changes of the basic privilege set in the future (section 6).
- Demonstrate on an example how to use the new basic privileges in a program (appendix A).
- Discuss the new functions in the kernel that deal with the new privileges (appendix B).
- Show on the UFS file system driver how to change an existing file system driver to use the new kernel functions (appendix C).
- Provide sources where to find further information (appendix D).

2 The least privilege access principle: Minimizing the impact of exploits

In almost every well known operating system, processes started by a certain user run under the security context of this user. They may access the user's

³See <http://www.opensolaris.org/jive/thread.jspa?threadID=10577&tstart=45> for a discussion on the security-discuss mailing list

files, initiate network connections, consume CPU resources and communicate with other local processes with the user's identity. Two processes started by the same user have exactly the same power concerning the actions they are allowed to perform regardless of their task they should actually fulfill. A calculator program, for example, normally has no need to communicate over network or delete the user's files like a mail client or browser that was started by the same user would have, but if it tries, it will succeed.

Normally, the fact that a process is able to perform actions it does not need to perform in order to fulfill its task should not be a problem, because it does not perform them. This becomes a problem in the moment, a process is attacked by an intruder that forces it to perform actions that were not intended by the programmer. In the worst case, the intruder is able to let the process execute arbitrary code under its security context. While this sounds unbelievable for the first time, you hear about such a possibility, there are many standard attacks (also called exploits) available in the web that may be downloaded by inexperienced attackers and applied to a vulnerable process. The availability of the vulnerable process is then made to do everything the intruder wants it to do⁴. Typically, the exploit forces the attacked process to start a shell with the rights of the attacked process for the intruder. An exploited calculator process that runs with the rights of an attacked user will never start a shell normally but of course it is allowed to do so because the user running the calculator is.

This problem leads us to the least privilege principle: A process should only have the power to perform actions it also needs to fulfill its tasks. A calculator for example only needs to read inputs, use the processor, write its output and perhaps access some globally readable libraries and configuration files. It should not have the rights to execute other processes, initiate network access or delete any files. If a process is able to exactly tell its needs to the operating system, it is very unlikely that a standard exploit for this process may cause any harm.

Unfortunately, most operating systems did not support the least privilege principle in the past and only partially support it nowadays. Most actions that may be performed by a process on a resource⁵ like accessing⁶ it or modifying its meta information⁷ are still hard coded. Actions may be exploited in any case.

The next sections illustrate the development from the traditional Unix *all or nothing* approach over the privilege model introduced in Solaris 10 to the new basic privileges I added to this model up to potential changes in the future. To show the increasing impact of the least privilege principle on operating system design, I created a figure style that uses the so called *behaviour brick/action notation*.

A behaviour brick is symbolized by a rectangle and has a name associated with it that is written in the first compartment of this rectangle. If the frame

⁴Take a look in *Hacking : the art of exploitation* by Jon Erickson to learn how exploits technically work.

⁵Resources are e. g. ordinary files, message queues, sockets and processes.

⁶Accessing basically means performing reading, writing, executing, entering and removing actions.

⁷Meta information are e. g. access rights or modification/creation/access time stamps

of the brick is dotted, it may be removed by the process, if it is solid, it cannot be removed. The more solid bricks exist, the less the least privilege principle is implemented in the operating system.

An action is an operation that should be performed by a process. It is symbolized by an arrow with its name written above it. Action arrows are placed above one or more brick columns. For every brick column, an action arrow is placed above, only the highest present brick is taken under consideration, lower bricks if present or not will be ignored. If for all brick columns in question, the conditions written in the second compartment of the highest available bricks in these columns are true for the process, the action is granted. In all other cases it is denied. If no brick is present in a brick column in question, the action is not granted, too.

Bricks are arranged in rows and columns. As already explained, bricks in a column always grant or deny the same actions whereas lower bricks are ignored if higher bricks are present. Bricks in one row typically grant different actions but belong to the same *privilege level*. In a default configuration of a process, if one brick in a row is present, all bricks in the row are present. The meaning and sense of this notation will get clearer in the next sections when these figures are actually used.

3 Traditional Unix Behaviour: All or nothing

The security model of traditional Unix operating systems is based on only one big distinction. Either a process is running under the context of the super user ($\text{euid} == 0$) and may perform every imaginable operation or it is running under the context of an ordinary user ($\text{euid} != 0$) and is only allowed to perform actions on objects that explicitly grant access rights for the user, its associated groups or for everybody⁸. Figure 1 illustrates the traditional Unix file access behaviour. There is only one layer of solid behaviour bricks, so it is not possible for a process to specify that it

- does not need read/write/execute the contents of files at all, even if it has the euid , egid or group memberships to do so.
- does not need to read/write/execute the contents of owned files but only of globally readable/writable/executable files.
- does need read access to every file but no write access at all (or vice versa).

Often a process needs to perform one special action that is normally only granted to the super user. For example, ping needs to send ICMP packets

⁸Actually, every process is associated with an effective user id (euid), a real user id and a saved user id. The effective user id is the id that is used if checking for permissions. Typically, all three ids are equal. A process may swap its ids or chose a totally new one (only if $\text{euid} == 0$) to permanently or temporarily change its power. Of course, an exploit is also able to do that, so running with an $\text{euid} != 0$ but having saved the super user id offers only limited protection.

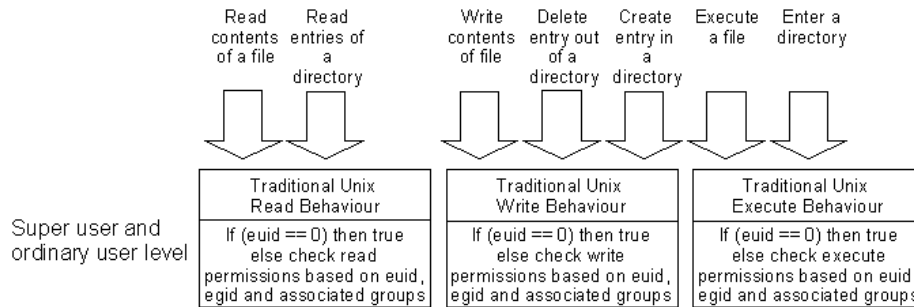


Figure 1: Traditional Unix File Access Behaviour

over a raw socket, a web server likes to bind on port 80 and a ftp daemon likes to chroot to a protected environment and may store and read files with different ownership there. As you can see from figure 2, these actions require the processes to run with the effective user id set to zero (the super user). Because the behaviour bricks cannot be removed by the process, switching to super user context enables the process to perform ALL special actions and not just the actions it really needs to perform its task – implementing the least privilege principle is not possible⁹.

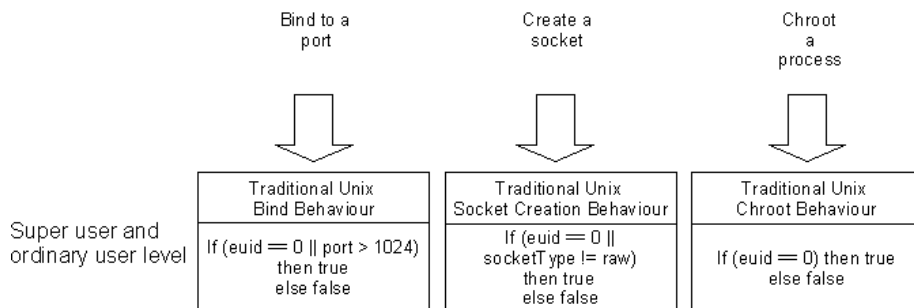


Figure 2: Traditional Unix File Network and Chroot Behaviour

The second problem one may recognize in figure 2 is that permissions granted to an ordinary user cannot be dropped. It is not possible for any process to declare to the operating system that it will never ever initiate or accept any network traffic.

To give some further examples about the problems with the traditional Unix *all or nothing* approach, I like to show some additional actions that may only

⁹Actually, there are workarounds that try to solve these problems with set uid binaries that were started with the effective user id set to the owner of the binary, perform the special operation and then change back their effective uid to a binary-specific uid or to the uid of the user that started the process. This always requires applying a suid bit to a super user owned binary by the super user itself and only helps if these special actions should be performed only once at process startup and there is no way to exploit the process during that time.

be performed by the super user or cannot be denied at all. Figure 3 illustrates that it is not possible to

- Enable a process running under the context of an ordinary user to send a signal to a process it does not own.
- Deny a process to send signals at all or only allow sending signals to processes in the same session.
- Deny a process to fork itself¹⁰.
- Deny a process to create a hardlink at all or only allow creating hardlinks to owned files.

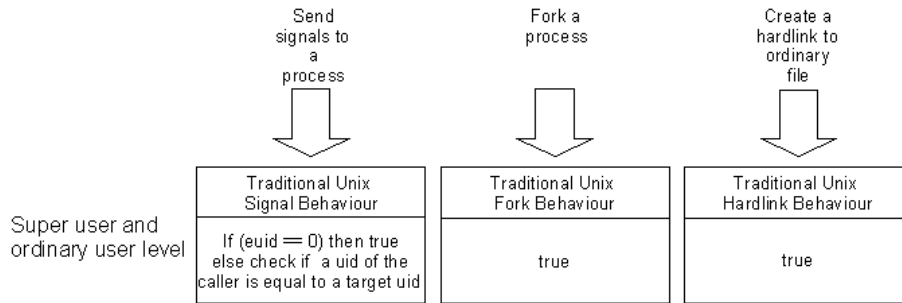


Figure 3: Traditional Unix Signal, Fork and Hardlink Behaviour

While the list of limitations is not nearly complete, I like to expand it with four last actions shown in figure 4. Because the traditional Unix security model only provides one layer of non-removable behaviour bricks it is (also) not possible to

- deny a process to execute a file if it has sufficient access privileges.
- deny a process to query the state of other processes at all or only allow it to query the state of processes running under the same user id.
- deny a process to change access and modification times of owned files.
- enable a process running under the context of an ordinary user to change access and modification times of non-owned files.
- deny a process to change permissions of owned files.
- enable a process running under the context of an ordinary user to change permissions of non-owned files.

To implement the least privilege principle, the traditional Unix security model has to be changed as follows:

¹⁰You may use setrlimit to do that but this is a hack.

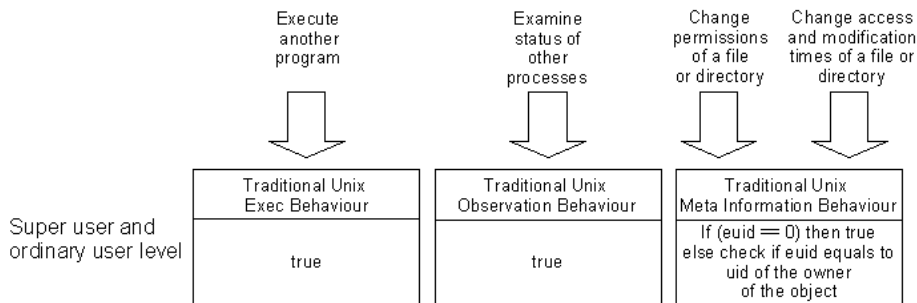


Figure 4: Traditional Unix Exec Observation and Meta Information Behaviour

- Less solid behaviour bricks: If a process is able to remove behaviour bricks it does not need to fulfill its task, it cannot be exploited to use them.
- More than just one layer: If behaviour bricks are arranged on multiple layers, the process has fine grained control over what behaviour it really needs and what bricks it can drop without losing too much power.

The next section illustrates how the Solaris 10 privileges model followed that direction and what limitations are left still open.

4 Solaris 10 privileges: Coming closer to the least privilege access principle

Up to Solaris 10, the Solaris operating system was based on the concepts of the traditional Unix security model. With Solaris 10, a privilege based security model was introduced that is in some aspects comparable to the POSIX 1003.1e standard draft and its implementations (e. g. the Linux¹¹ capability model) but has some nice features like basic privileges that are not defined there.

You may imagine privileges as removable behaviour bricks. Instead of permission checks relying on the effective user id, all checks are now based on the privileges present (or not present) in the effective privilege set¹². So if a process does not need a privilege, it may simply drop it from the effective privilege set (but may add it later again). If it also drops the privilege from its permitted

¹¹Linux is a registered trademark of Linus Torvalds.

¹²Actually, a process is associated with an effective, a permitted, an inheritable and a limit privilege set. Checks are based on the effective privilege set that may only contain privileges also present in the permitted privilege set that in turn may only be shrunk but not expanded during process runtime. If a process executes a program, it will inherit the permissions that are both present in the inheritable and the limit set. Legacy programs that do not know anything about privileges (they are not privilege aware) will continue to work as expected because they are treated as if they had the appropriate privileges in their sets. This is not an in-depth explanation of the Solaris privilege model, see appendix D or privileges(5) for further details.

privilege set, the privilege is also removed from its effective set and there is no possibility for the process to get it back.

Figure 5 shows the Solaris 10 file access behaviour. The names of the non-solid behaviour bricks correspond with the names of the Solaris 10 privileges that decide whether these behaviour bricks are present for the process. In comparison to the traditional Unix approach, behaviour bricks are now arranged in two layers whereas the bricks on super user level (realized through Solaris privileges) may be removed by the process or may not be present at process startup. Typically, a process started by the super user has associated all defined privileges with it, whereas a process started by an ordinary user may only use behaviour bricks on the ordinary user level. The main difference to the traditional approach is that now super user processes may decide to permanently drop its power to write every file, while they do not lose their power to read every file or vice versa. It is still not possible for a process to specify that it

- does not need read/write/execute the contents of files at all, even if it has the euid, egid or group memberships to do so.
- does not need to read/write/execute the contents of owned files but only of globally readable/writable/executable files.

This is because the behaviour bricks on the ordinary user level are not (yet) removable. To make these bricks removable was one of my task during Google Summer of Code (see section 5).

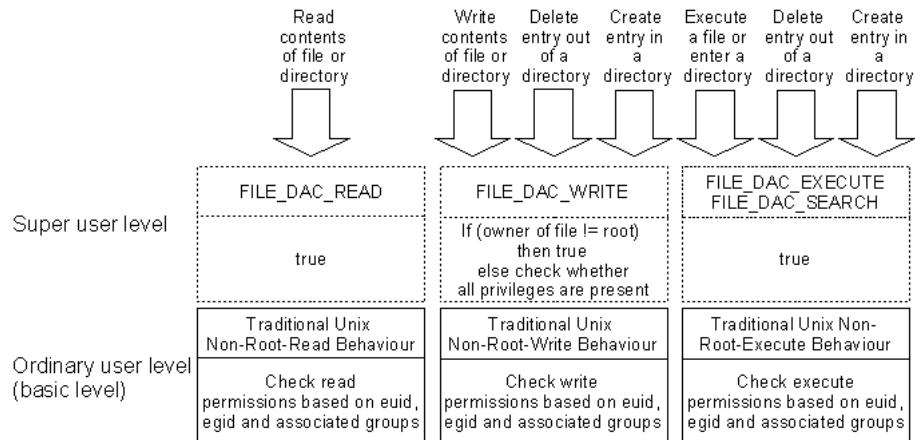


Figure 5: Solaris 10 File Access Behaviour

Of course, the Solaris 10 privilege model did not only concentrate on privileges related to unrestricted file access. The special actions needed by ping, a web server or a ftp daemon illustrated in the previous chapter are now granted (or denied) by removable behaviour bricks on the super user level as shown in

figure 6. A process that needs to bind to a port under 1024, create raw sockets or chroot to a restricted environment still has to be started by the super user or with the rights of the super user¹³ but may immediately drop all non-needed privileges (removable behaviour bricks) so that it only has the power of an ordinary process plus the special power it really needs.

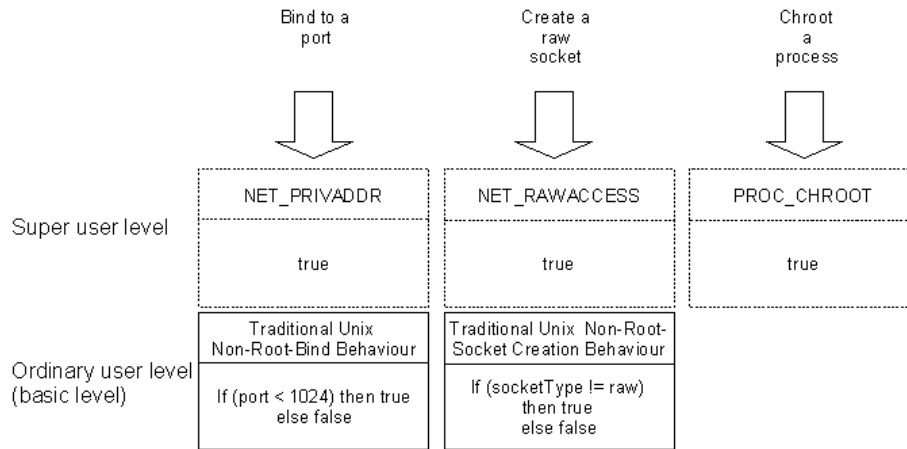


Figure 6: Solaris 10 Network and Chroot Behaviour

The current implementation already comes very close to the least privilege principle, but it is still not possible for a process to declare to the operating system that it will never ever initiate or accept any network traffic because the behaviour bricks on the ordinary user level are not (yet) removable for this kind of actions.

Up to this point, we have only seen removable behaviour bricks on the super user level. A super user process may remove them to lose power it does not need but ordinary processes do not benefit at all because for them, the behaviour bricks at super user level are usually not present and the remaining bricks on ordinary user level cannot be removed (there is no Solaris 10 privilege defined (yet)). Other operating systems like Linux that have a comparable security model¹⁴ only define removable behaviour bricks at super user level. The Solaris 10 privilege model went a step further and also defined five basic privileges – removable behaviour bricks at ordinary user level – that are also present in the privileges sets for ordinary processes. So an ordinary process may remove some

¹³If a processes' binary is owned by the super user and the suid bit is set, the process will be started with the effective user id set to zero (super user). This is treated as if all defined privileges are present in all privilege sets. After having dropped all unnecessary privileges from the sets, the process will typically change its effective user id back to its real user id while its privileges sets will remain the same so it can still benefit from its additional power.

¹⁴In Linux, removable behaviour bricks are called capabilities. Linux capabilities are inspired by the POSIX 1003.1e standard draft and are also arranged in various sets. Most of the super user privileges of Solaris are also defined as capabilities in Linux. See the Linux man page for capabilities for further details.

or all of these basic privileges from its effective or permitted privilege set and restrict its power below the normal level. The set of all defined basic privileges is called basic privilege set. For an ordinary process the effective and permitted sets are equal to the basic privilege set¹⁵.

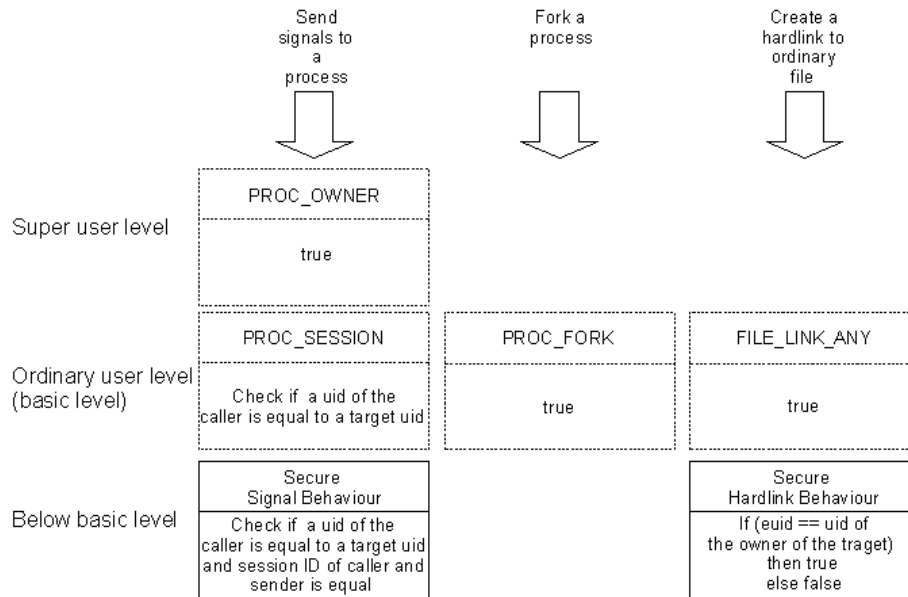


Figure 7: Solaris 10 Signal, Fork and Hardlink Behaviour

In figure 7 you can see three basic and one super user privilege. It is possible in Solaris 10 to

- enable a process running under the context of an ordinary user to send a signal to a process it does not own.
- deny a process to send signals to processes in other sessions.
- deny a process to fork itself.
- deny a process to create a hardlink to a file owned by a different user.

It is still not possible to

- deny a process to send signals at all,
- deny a process to create a hardlink at all,

because the behaviour bricks shown on the *below basic* level cannot be removed (yet).

¹⁵The inheritable set is also equal to the basic privilege set whereas the limit set still contains all defined privileges.

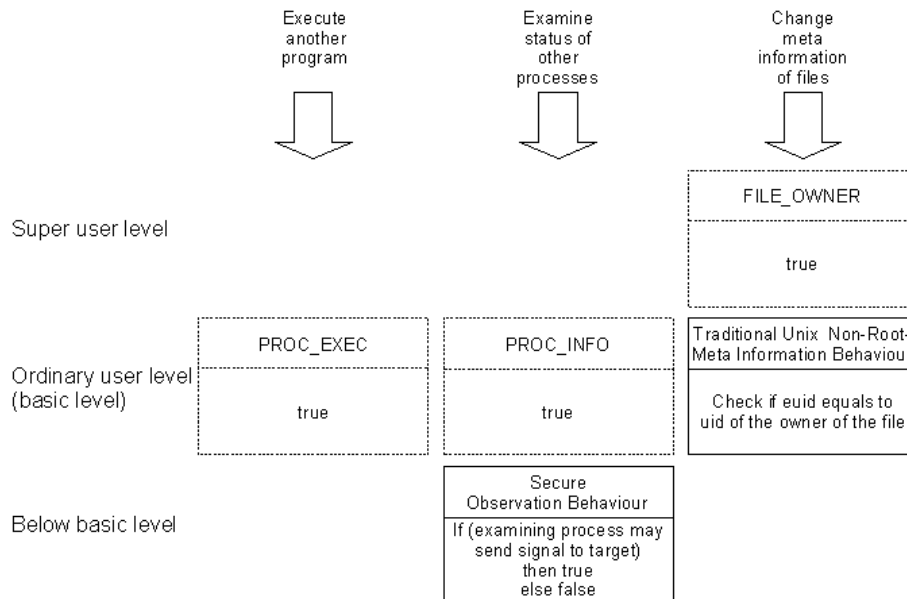


Figure 8: Solaris 10 Exec, Observation and Meta Information Behaviour

In the previous chapter, I mentioned the limitations of the traditional Unix security model concerning process state observation, alteration of meta information and execution of processes. As figure 8 shows, Solaris 10 made some progress to solve these problems with some new basic and super user privileges that may be dropped if not needed. It is now possible to

- deny a process to execute a file if it has sufficient access privileges.
- deny a process to query the state of other processes running under a different user id.
- enable a process running under the context of an ordinary user to change access and modification times of non-owned files.
- enable a process running under the context of an ordinary user to change permissions of non-owned files.

but still not possible to

- deny a process to query the state of other processes at all.
- deny a process to change access and modification times of owned files.
- deny a process to change permissions of owned files.

As you may have noticed from the given examples, there are only few basic privileges in Solaris 10. Most behaviour bricks on the basic and below basic

level are still not removable which in turn leads to the remaining limitations. To fully support the least privilege access principle, these solid bricks have to be substituted by removable ones by introducing new basic privileges. This exactly was my job during Google Summer of Code and is explained in detail in the next section.

5 Proposed file access behaviour of OpenSolaris: Google summer of code contributions to the basic privilege set

As mentioned in the previous section, dropping the currently defined basic privileges only restricts some actions related with forking, signaling and observing processes as well as hardlinking to non-owned files and executing binaries. There are no removable behaviour bricks in Solaris 10 to switch file access completely or partially off or to restrict the capability of a process to change meta information of its owned files.

In most cases, switching read/write/execute file access completely off is a too radical approach. The processes may still want to load globally readable libraries and configuration files and may need to change globally writable files or directories. This brought me to the idea of introducing a third layer of behaviour bricks – the *anonymous* level. Anonymous processes should be processes that choose to (partially) give up their identity, that means they cannot benefit from the fact that they are running with a special effective uid and are a member of certain groups¹⁶. If a process started by user Bob decides to be anonymous, it may not access files, Bob can access but everybody is not allowed to access. If Alice shares her files with a group Bob is member in, and his process is anonymous, it cannot access Alice's files but still read or write globally accessible files. A process may also decide to give up its power to write its owned files but still need the ability to read them. In this case, a process only becomes partially anonymous.

If you remember figure 5, it was my job to make the behaviour bricks on ordinary user level removable to give processes the possibility to become anonymous and to introduce removable behaviour bricks on the anonymous level to give processes the possibility to switch off read/write/execute file access at all. In figure 9 you see the results of my work.

The semantics of the new introduced basic privileges are now individually explained:

- *FILE_NANON_READ*:

Without having set *FILE_NANON_READ*, a process is only able to perform read operations on globally readable files or directories because

¹⁶Anonymous processes are not anonymous in a way that they are hidden in the process table, they only lose the power to benefits from the object access rights associated with their identity.

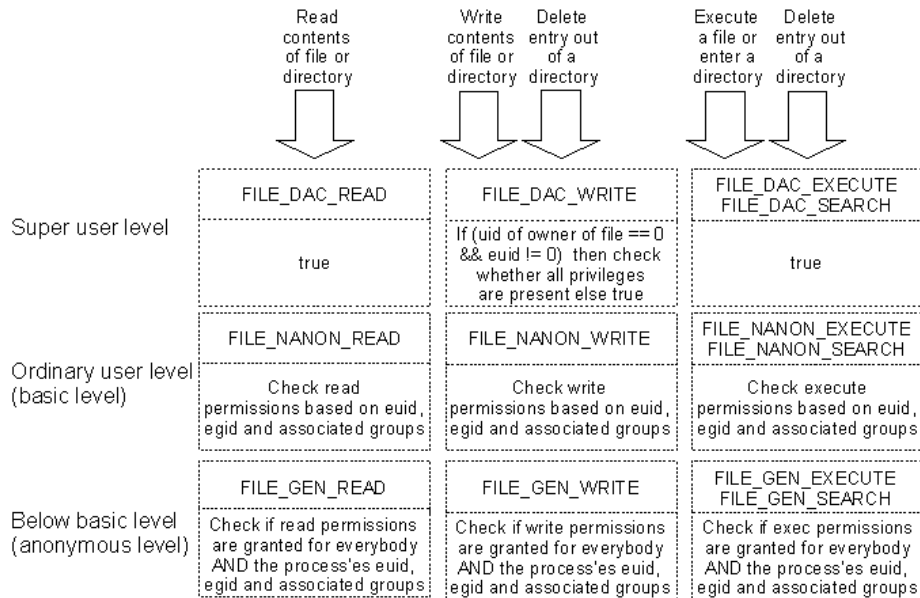


Figure 9: Proposed OpenSolaris File Access Behaviour

it can only act like an 'anonymous process' and cannot benefit from its euid, egid or group memberships. Additional rights gained through FILE_DAC_READ will not be restricted if FILE_NANON_READ is not set.

- **FILE_NANON_WRITE:**

Without having set FILE_NANON_WRITE, a process is only able to perform write operations on globally writable files or directories because it can only act like an 'anonymous process' and cannot benefit from its euid, egid or group memberships. Additional rights gained through FILE_DAC_WRITE will not be restricted if FILE_NANON_WRITE is not set.

- **FILE_NANON_EXECUTE:**

Without having set FILE_NANON_EXECUTE, a process is only able to perform execute operations on globally executable files because it can only act like an 'anonymous process' and cannot benefit from its euid, egid or group memberships. Additional rights gained through FILE_DAC_EXECUTE will not be restricted if FILE_NANON_EXECUTE is not set.

- **FILE_NANON_SEARCH:**

Without having set FILE_NANON_SEARCH, a process is only able to perform search operations on globally searchable directories because

it can only act like an 'anonymous process' and cannot benefit from its euid, egid or associated groups. Additional rights gained through FILE_DAC_SEARCH will not be restricted if FILE_NANON_EXECUTE is not set.

- *FILE_GEN_READ:*

Without having set FILE_GEN_READ, file or directory operations that require read permissions will generally not be granted unless the FILE_DAC_READ or FILE_NANON_READ privilege is set. Additional rights gained through FILE_DAC_READ or FILE_NANON_READ will not be restricted if FILE_GEN_READ is not set.

- *FILE_GEN_WRITE:*

Without having set FILE_GEN_WRITE, file or directory operations that require write permissions will not be granted unless the FILE_DAC_WRITE or FILE_NANON_WRITE privilege is set. Additional rights gained through FILE_DAC_WRITE or FILE_NANON_WRITE will not be restricted if FILE_GEN_WRITE is not set.

- *FILE_GEN_EXECUTE:*

Without having set FILE_GEN_EXECUTE, file operations that require execute permissions will not be granted unless the FILE_DAC_EXECUTE or FILE_NANON_EXECUTE privilege is set. Additional rights gained through FILE_DAC_EXECUTE or FILE_NANON_EXECUTE will not be restricted if FILE_GEN_EXECUTE is not set.

- *FILE_GEN_SEARCH:*

Without having set FILE_GEN_SEARCH, directory operations that require search permissions will not be granted unless the FILE_DAC_SEARCH or FILE_NANON_SEARCH privilege is set. Additional rights gained through FILE_DAC_SEARCH or FILE_NANON_SEARCH will not be restricted if FILE_GEN_SEARCH is not set.

As you may have noticed from the descriptions, the presence or non-presence of the new privileges is ignored if a behaviour brick on a higher level is present in the same brick column. If for example, a behaviour brick on super user level is present that grants read permissions for every file, it does not matter whether the process is anonymous or not. If the behaviour brick on basic level is present that allows writing the content of files the user associated with the effective uid of the process has write access to, it does not matter whether the behaviour brick on anonymous level was removed to switch write access totally off. This conforms to the behaviour associated with the action/behaviour brick notation explained in section 2.

You may have also noticed that the lower the level of behaviour bricks, the more restrictive is their condition to (partially) grant the associated actions.

This is no surprise. Imagine a file that grants read permissions to everybody but not to Bob. Of course, a process started by Bob must not be able to access the file because it drops `FILE_NANON_READ` – otherwise this would result in a privilege escalation¹⁷.

The new removable behaviour bricks introduced up to this point were only associated with actions that operate on already existing files. Should an anonymous process be able to create files or change meta information of files owned by the user associated with the processes' effective uid? To my mind it does not because an anonymous process has given up its identity and therefore cannot create or modify meta information of files that are owned by this identity. It may even be desirable for a process to only give up the ability to create or change permissions of files¹⁸ but still be able to read, write and execute owned files.

To implement this behaviour, I introduced another new basic privilege on basic level: `FILE_NANON_OWNER`. This privilege allows a process to perform operations that require ownership on a file, link or directory. Because `FILE_OWNER` is a behaviour brick in the same column on a higher level, it may always substitute `FILE_NANON_OWNER`. Additional rights gained through `FILE_OWNER` will not be restricted if `FILE_NANON_OWNER` is not set.

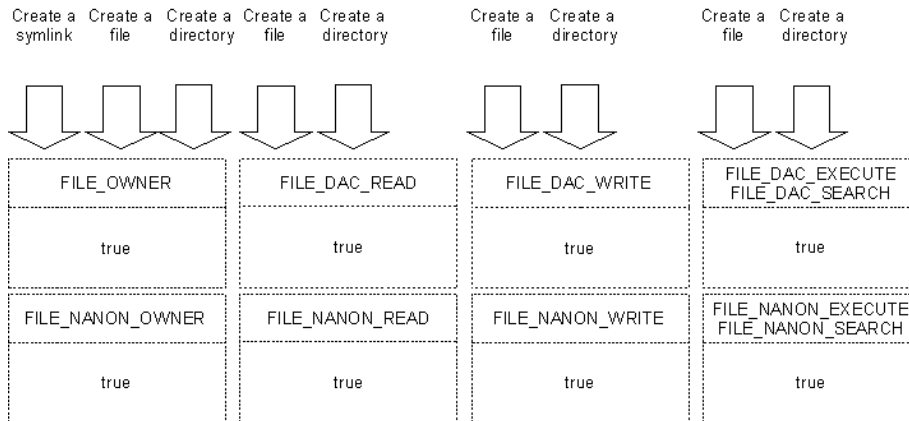


Figure 10: Proposed OpenSolaris File and Symlink Creation Behaviour

Without having set at least one of both privileges, a process is considered to

¹⁷Privilege escalations occur if actions performed by the process may lead to the ability to perform actions that should normally only be possible if one or more privileges currently not available in the processes' effective privilege set would be present. The Solaris 10 privilege model takes care that privilege escalations will not happen. Potentially dangerous operations are only granted if further privileges are present. For certain actions like changing files owned by the super user, all defined privileges of the processes' zone have to be present in the effective privilege set.

¹⁸Removing the ability to change permissions would lead from discretionary access control (every user may define who may access its files) to mandatory access control (only the super user may decide who may access whose files).

be 'anonymous' and cannot benefit from its effective uid. Affected operations are

- creating of a new file, directory or symbolic link (shown in figure 10)
- changing permission bits/ACLs (shown in figure 11)
- changing access and modification times (shown in figure 11)
- changing the gid of an owned file or directory when a process is member of this group (shown in figure 13)
- creating hardlinks to files owned by a uid equal to the processes' effective uid (shown in figure 12)
- renaming or moving in a directory with the sticky bit
- mounting a file system on a mount point owned by a uid equal to the processes' effective uid

To avoid privilege escalations, some of these operations require further behaviour bricks being present. If a new file should be created FILE_NANON_READ or FILE_DAC_READ, FILE_NANON_WRITE or FILE_DAC_WRITE and FILE_NANON_EXECUTE or FILE_DAC_EXECUTE have to be set as well.

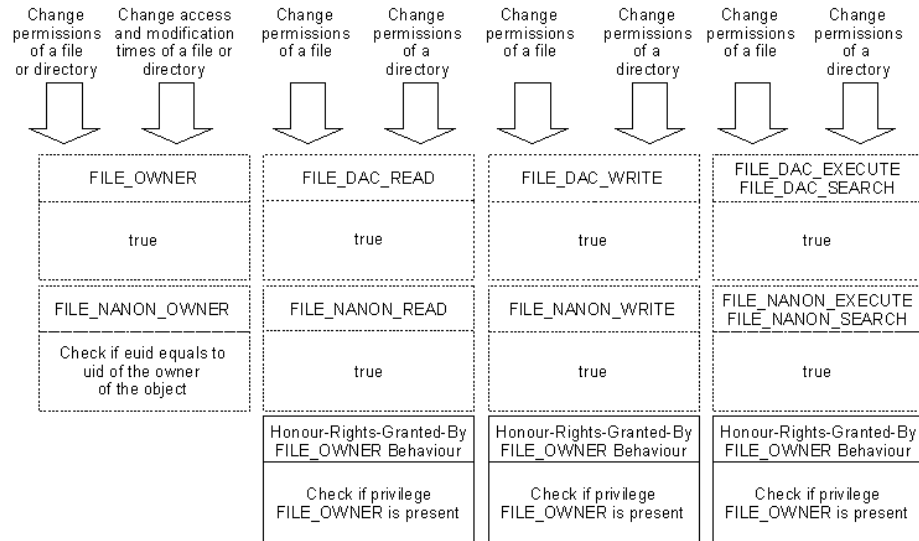


Figure 11: Proposed OpenSolaris Meta Information Behaviour

If a directory should be created FILE_NANON_READ or FILE_DAC_READ, FILE_NANON_WRITE or FILE_DAC_WRITE and FILE_NANON_SEARCH or FILE_DAC_SEARCH have to be set as well. So a process that was able to

create a file or directory is always able to reopen or enter it later as long as no privileges are removed from its effective set.

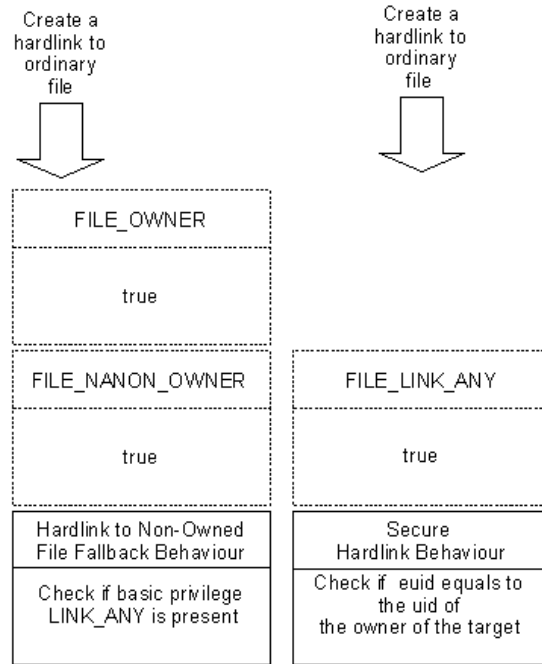


Figure 12: Proposed OpenSolaris Hardlink Creation Behaviour

To change permission bits/ACLs of an owned file `FILE_NANON_READ` or `FILE_DAC_READ`, `FILE_NANON_WRITE` or `FILE_DAC_WRITE` and `FILE_NANON_EXECUTE` or `FILE_DAC_EXECUTE` have to be set as well. Otherwise, an exploit of an anonymous process may change the permissions of a file owned by the user associated with the processes' effective uid in a way that everybody (and this anonymous process) is allowed to access it.

To change permission bits/ACLs of an owned directory `FILE_NANON_READ` or `FILE_DAC_READ`, `FILE_NANON_WRITE` or `FILE_DAC_WRITE` and `FILE_NANON_SEARCH` or `FILE_DAC_SEARCH` have to be set as well. So privilege escalation due to permission changes are not possible.

If a hardlink to a file owned by a uid equal to the processes' effective uid should be created without having set `FILE_OWNER` or `FILE_NANON_OWNER`, the request is treated as if the hardlink would target a file owned by a different user and therefore the `FILE_LINK_ANY` privilege is needed to grant the operation.

To change the gid of an owned file or directory when a process is member of this group, at least one of `FILE_NANON_OWNER`, `FILE_OWNER`, `FILE_CHOWN` or `FILE_CHOWN_SELF` has to be set. Figure 13 shows how

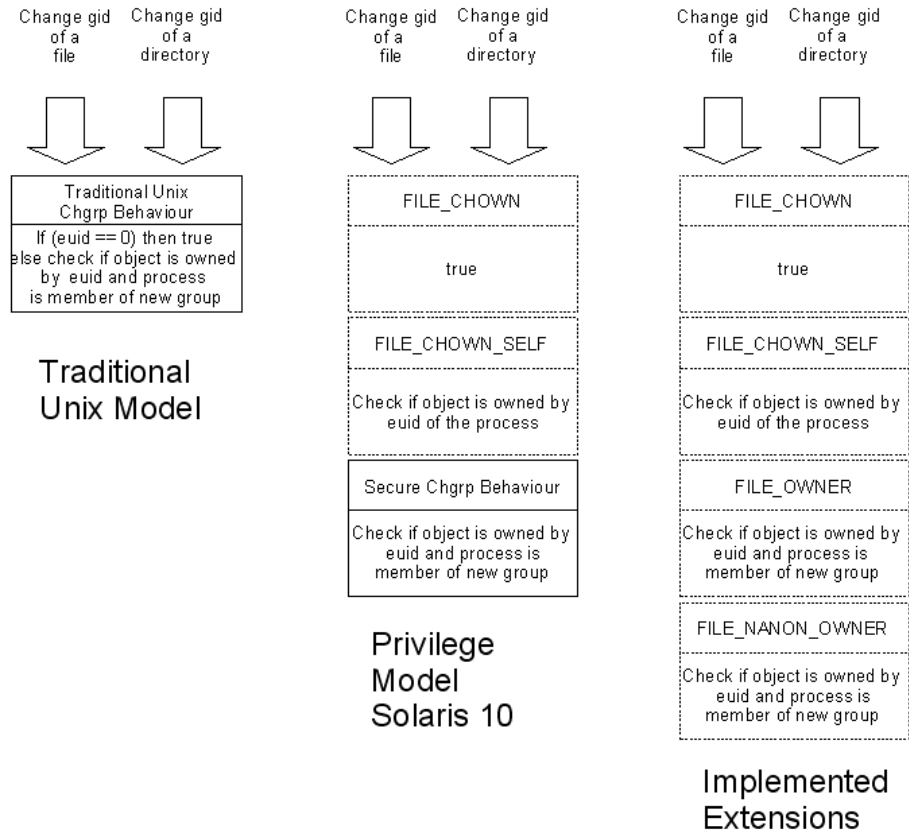


Figure 13: Unix, Solaris and Proposed OpenSolaris Chgrp Behaviour

much the chgrp behaviour has changed from the traditional Unix approach over the Solaris 10 privilege model to the behaviour implemented during the Google Summer of Code project. At the beginning, there were only non-removable behaviour bricks that were all arranged on one single layer. Implementing the least privilege principle was hardly possible. After applying my changes, there are only removable behaviour bricks on three different layers so that processes have fine grained control to only keep the power they really need.

On <http://myhpi.de/~nicolai/webrev/> you may review the changes made to the OpenSolaris kernel (build onnv_45) to introduce the presented privileges and check them with appropriate secpolicy functions further explained in appendix B. I also modified two file system drivers (UFS and TMPFS) to let them call these functions when deciding whether an associated action should be granted. In sum, I added about 1050 lines of code to the OpenSolaris kernel that all conform to the coding style guidelines and pass lint without any warnings. Additional details how to change further file systems can be found in appendix

C.

The expected behaviour of the privileges was already successfully tested with a self written test suite. A useful example how and when to drop the new introduced privileges is presented in appendix A.

Up to the time of writing, the proposed changes have not yet completed the review process so it is not sure, whether they will find there way in the official OpenSolaris kernel. If they do, further file system drivers like ZFS, PCFS, NFS and DEVFS might be changed to use them as well. Furthermore, the man pages of the system calls that are used to request the affected actions and the privileges man page itself have to be changed to reflect the new behaviour¹⁹ Finally, an investigation which daemons may benefit from dropping the new privileges should be taken. Appendix A gives some advices how to find out.

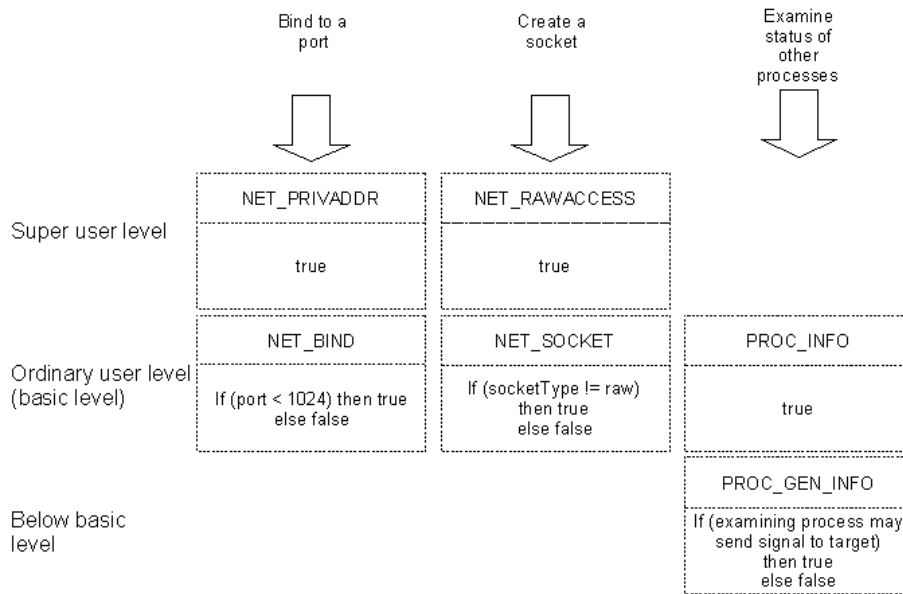


Figure 14: Proposed Network and future Process Observation Behaviour

To my mind, introducing the new basic privileges in OpenSolaris is just a consequent step in a long transition from the inflexible traditional Unix approach to an approach that fully supports the least privilege principle. I only made some already present solid behaviour bricks removable by expanding the basic privilege set. There is a certain chance that further limitations of the Solaris 10 privilege model (see previous section) may be solved with the same approach in the future. The next section shows some potential candidates.

¹⁹Of course, the default behaviour has not changed because all new introduced privileges are basic privileges and therefore present in the initial privilege sets of all processes.

6 The future: What functionality to expect next in the basic privilege set

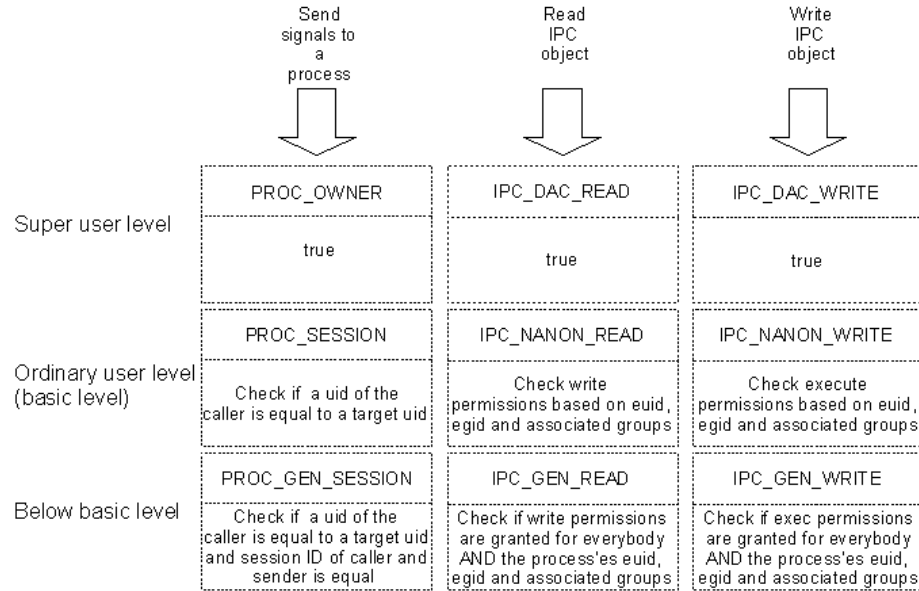


Figure 15: Future IPC Behaviour

If you followed the examples given to demonstrate the inflexibility of the traditional Unix approach, you may have noticed that some problems are still not solved in Solaris 10 and my contributions. For example, it is not yet possible to

1. deny a process to accept or initiate network connections.
2. deny a process to query the state of other processes at all.
3. deny a process to send signals at all.

At the beginning of my project, I focused on the first problem and had a solution comparable to the one illustrated in figure 14 in mind. As already told in section 1, the OpenSolaris community rejected this proposal because it seems as if basic privileges are not the best mechanism to restrict network access.

The other leftover problems are perfectly suited to be solved by introducing new basic privileges. In the figure mentioned above, you can also find a solution proposal to solve the second limitation. The third limitation is related to interprocess communication (IPC). Message queues, semaphores, doors, shared memory blocks, named pipes and processes are typical objects, IPC related actions operate on. All IPC objects are associated with permissions that define

<i>Operations that may be enforced by setting or removing non-solid behaviour bricks with the traditional Unix approach, Solaris 10 privileges, Open Solaris Proposals and in a future system</i>	<i>Traditional Unix</i>	<i>Solaris 10</i>	<i>Open Solaris Proposals</i>	<i>Future</i>
enable a process running under the context of an ordinary user to give away non-owned files	✗	✓	✓	✓
enable a process running under the context of an ordinary user to give away owned files	✗	✓	✓	✓
deny a process to fork itself	✗	✓	✓	✓
enable a process running under the context of an ordinary user to send a signal to a process it does not own	✗	✓	✓	✓
deny a process to send signals to processes in other sessions	✗	✓	✓	✓
deny a process to create a hardlink to a file owned by a different user	✗	✓	✓	✓
deny a process to execute a file if it has sufficient access privileges	✗	✓	✓	✓
deny a process to query the state of other processes running under a different user id	✗	✓	✓	✓
enable a process running under the context of an ordinary user to access any file	✗	✓	✓	✓
enable a process running under the context of an ordinary user to change meta informations of non-owned files	✗	✓	✓	✓
enable a process to become anonymous regarding file operations	✗	✗	✓	✓
deny a process to access files at all	✗	✗	✓	✓
deny a process to perform any IPC operation	✗	✗	✗	✓
deny a process to query the state of other processes at all	✗	✗	✗	✓
enable a process to become anonymous regarding IPC operations	✗	✗	✗	✓
deny a process to initiate or accept network connections	✗	✗	✗	✗

Figure 16: Development of the support to implement the least privilege principle with the help of removable behaviour bricks

which processes may perform which actions on them. Consequently, IPC objects are very similar to files. So a duplication of the new introduced privileges to restrict file access seems only logical. Figure 15 shows a possible solution.

Figure 16 summarizes the steps made during the transition from the traditional Unix security model to the proposed changes of the basic privilege set in the future. The table only compares the support for operations presented in the examples of this article. These are only a small selection out of all supported operations of the Solaris operating system. Many other operations are still only granted by solid behaviour bricks that may be replaced by introducing even further basic privileges. It is very likely that the basic privilege set will grow in the future. Actions that are granted to every process nowadays will require special behaviour bricks later, but because all basic privileges are present in the effective privilege set per default, there is no problem for legacy applications.

A How to use the new basic privileges: Confining ssh-agent

Introducing new basic privileges only helps if programs decide to drop them. The nine privileges explained in section 5 may be used to restrict file access. This section demonstrates how ssh-agent uses these new features to come closer to the least privilege principle and thus become much harder to exploit²⁰. The changes made to ssh-agent can be reviewed at <http://myhpi.de/~nicolai/webrev-ssh-agent/>.

ssh-agent(1) is an authentication agent that holds private keys used for public authentication. It is often used in conjunction with ssh to enter passwords for encrypted keys only once. After starting up and setting up a socket in a protected directory that in turn is typically created in the /tmp directory, ssh-agent only listens on its socket for new keys to be stored in memory or to pass a key to a program like ssh started by the same user that started ssh-agent. Because ssh-agent runs under the security context of the user who started it, an exploit of ssh-agent would enable the attacker to perform all operations the user is able to do. To protect against these kinds of attacks, ssh-agent implements the least privilege principle and permanently drops almost all privileges, including the privileges to read files. This is done by removing these privileges from the permitted privilege set:

```
1 /*
2  * Drop unneeded privs, including basic ones like fork/exec.
3  *
4  * Idiom: remove from 'basic' privs we know we don't want,
5  * invert the result and remove the resulting set from P.
```

²⁰This is not a detailed introduction into the user level interfaces provided to manipulate privilege sets. On the other hand, it is very likely that you could fully understand how to do this by just reading the provided code. See `privileges(5)`, `priv_set(3C)`, `ppriv(1)` and appendix D for further information.

```

6  *
7  * None of the priv_delset() calls, nor the setppriv call
8  * below can fail, so their return values are not checked.
9  */
10 if ((myprivs = priv_str_to_set("basic", ",", NULL)) == NULL)
11     fatal("priv_str_to_set failed: %m");
12 (void) priv_delset(myprivs, PRIV_PROC_EXEC);
13 (void) priv_delset(myprivs, PRIV_PROC_FORK);
14 (void) priv_delset(myprivs, PRIV_FILE_LINK_ANY);
15 (void) priv_delset(myprivs, PRIV_PROC_INFO);
16 (void) priv_delset(myprivs, PRIV_PROC_SESSION);
17 (void) priv_delset(myprivs, PRIV_FILE_NANON_READ);
18 (void) priv_delset(myprivs, PRIV_FILE_GEN_READ);
19 (void) priv_delset(myprivs, PRIV_FILE_NANON_EXECUTE);
20 (void) priv_delset(myprivs, PRIV_FILE_GEN_EXECUTE);
21 (void) priv_delset(myprivs, PRIV_FILE_GEN_WRITE);
22 priv_inverse(myprivs);
23 (void) setppriv(PRIV_OFF, PRIV_PERMITTED, myprivs);
24 (void) priv_freeset(myprivs);

```

As you can conclude from the code, ssh-agent permanently removes all behaviour bricks on super user level and the basic behaviour bricks that grant to

- successfully call the exec system call (line 12).
- fork the process (line 13).
- make a hardlink to a file not owned by the user that started ssh-agent (line 14).
- query the status of processes not owned by the user that started ssh-agent (line 15).
- send signals to processes outside the session of ssh-agent (line 16).
- perform read operations on files and directories that are not (just) granted to everybody but to the user that started ssh-agent (line 17).
- perform read operations on world readable files and directories that are also granted to the user that started ssh-agent (line 18).
- perform execute operations on files that are not (just) granted to everybody but to the user that started ssh-agent (line 19).
- perform execute operations on world readable files that are also granted to the user that started ssh-agent (line 20).

- perform write operations on world readable files that are also granted to the user that started ssh-agent²¹ (line 21).

Because ssh-agents needs to remove its socket and the protected directory containing the socket, it cannot permanently remove the power to modify and delete files or directories. Because the protected directory is located in the /tmp directory with the sticky bit being set, this also applies to operations that require ownership permissions. Therefore, the privileges that grant these actions are only temporarily removed from the effective privilege set but rest in the permitted privilege set:

```
/*
 * We can temporarily drop PRIV_FILE_NANON_OWNER, PRIV_FILE_NANON_WRITE
 * and PRIV_FILE_NANON_SEARCH from the effective set because we only
 * need them when cleaning up
 */
priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_FILE_NANON_OWNER,
        PRIV_FILE_NANON_WRITE, PRIV_FILE_NANON_SEARCH, NULL);
```

Only when the additional power is really needed at cleanup time²², the permissions get temporarily back into the effective privilege set. This technique is called *privilege bracketing*²³:

```
static void
cleanup_socket(void *p)
{
    if (socket_name[0]) {
#ifdef HAVE_SOLARIS_PRIVILEGE
        /*
         * Now we have to set PRIV_FILE_NANON_WRITE,
         * PRIV_FILE_NANON_SEARCH in the effective privilege set
         * to be able to delete the socket and the directory.
         */
        priv_set(PRIV_ON, PRIV_EFFECTIVE, PRIV_FILE_NANON_WRITE,
                PRIV_FILE_NANON_SEARCH, NULL);
```

²¹The missing of this privilege is ignored if FILE_NANON_WRITE is present in the effective privilege set (see section 5 and 2 for details.).

²²To delete the protected directory out of the /tmp directory, it would suffice to have FILE_GEN_WRITE instead of FILE_NANON_WRITE in the effective privilege set because /tmp is world writable but because removing the socket requires FILE_NANON.WRITE we use this privilege twice.

²³Privilege bracketing does not hinder an exploit to move the privileges back into the effective privilege set, but often it is very difficult for an exploit to execute arbitrary code and not just try to start a shell. It is really important that the temporarily added privileges are withdrawn in the same function they are set immediately after the privileged operation is performed even if the program normally exits after calling that function. Otherwise, the exploit may exactly jump into that function to get the additional power and then return to its damage routine by manipulating the stack. In any case, privilege bracketing helps a process to protect for programming mistakes and 'standard exploits'.

```

#endif /* HAVE_SOLARIS_PRIVILEGE */
        unlink(socket_name);
#ifdef HAVE_SOLARIS_PRIVILEGE
        /*
         * Drop privileges again
         */
        priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_FILE_NANON_WRITE,
                PRIV_FILE_NANON_SEARCH, NULL);
#endif /* HAVE_SOLARIS_PRIVILEGE */
    }
    if (socket_dir[0]) {
#ifdef HAVE_SOLARIS_PRIVILEGE
        /*
         * Now we have to set PRIV_FILE_NANON_OWNER and
         * PRIV_FILE_NANON_WRITE in the effective privilege set
         * to be able to delete the directory out of the sticky
         * /tmp/ directory.
         */
        priv_set(PRIV_ON, PRIV_EFFECTIVE, PRIV_FILE_NANON_OWNER,
                PRIV_FILE_NANON_WRITE, NULL);
#endif /* HAVE_SOLARIS_PRIVILEGE */
        rmdir(socket_dir);
#ifdef HAVE_SOLARIS_PRIVILEGE
        /*
         * Drop privileges again
         */
        priv_set(PRIV_OFF, PRIV_EFFECTIVE, PRIV_FILE_NANON_OWNER,
                PRIV_FILE_NANON_WRITE, NULL);
#endif /* HAVE_SOLARIS_PRIVILEGE */
    }
}

```

Finally, I like to show you a really useful tool when it comes to privilege debugging: `ppriv`. With `ppriv` you can

- turn on privilege debugging for a process (error messages will occur on the console if a needed privilege to grant an action was not present in the effective privilege set).
- modify the privilege sets of running processes.
- display the privilege sets of running processes.
- start processes with privilege debugging turned on and modified privilege sets²⁴.
- display whether a process is aware of privileges.

²⁴You can only change the privileges in the inheritable and limit privileges set.

Let's display the privilege sets of the modified ssh-agent:

```
bash-3.00$ eval 'ssh-agent'
Agenten-PID 17772
bash-3.00$ ppriv -v 17772
17772: ./ssh-agent
flags = PRIV_AWARE
E: file_gen_search
I: file_gen_execute,file_gen_read,file_gen_search,file_gen_write,
  file_link_any, file_nanon_execute,file_nanon_owner,
  file_nanon_read,file_nanon_search, file_nanon_write,
  proc_exec,proc_fork,proc_info,proc_session
P: file_gen_search,file_nanon_owner,file_nanon_search,
  file_nanon_write
L: contract_event,contract_observer,cpc_cpu,dtrace_kernel,
  dtrace_proc,dtrace_user,file_chown,file_chown_self,
  file_dac_execute,file_dac_read,file_dac_search,
  file_dac_write, file_downgrade_sl,file_gen_execute,
  file_gen_read,file_gen_search, file_gen_write,
  file_link_any,file_nanon_execute,file_nanon_owner,
  file_nanon_read,file_nanon_search,file_nanon_write,
  file_owner,file_setid,file_upgrade_sl,
  graphics_access,graphics_map,ipc_dac_read,
  ipc_dac_write,ipc_owner,net_bindmlp,net_icmpaccess,
  net_mac_aware,net_privaddr,net_rawaccess,proc_audit,
  proc_chroot,proc_clock_highres,proc_exec,proc_fork,
  proc_info,proc_lock_memory, proc_owner,proc_prioctl,
  proc_session,proc_setid,proc_taskid,proc_zone,sys_acct,
  sys_admin,sys_audit,sys_config,sys_devices,
  sys_ipc_config,sys_linkdir,sys_mount,sys_net_config,
  sys_nfs,sys_res_config, sys_resource,sys_suser_compat,
  sys_time,sys_trans_label,win_colormap,win_config,
  win_dac_read,win_dac_write,win_devices,win_dga,
  win_downgrade_sl,win_fontpath,win_mac_read,win_mac_write,
  win_selection,win_upgrade_sl
bash-3.00$
```

As you can see from ppriv's output, the only privilege that is permanently present in ssh-agent's effective privilege set is FILE_GEN_SEARCH. This privilege is needed because ssh-agent changes in a world searchable directory after having dropped most of its privileges. To remove the socket and the protected directory, the privileges FILE_NANON_OWNER, FILE_NANON_SEARCH and FILE_NANON_WRITE are still in the permitted privilege set. Of course, FILE_GEN_SEARCH is also present here because all privileges in the effective set are located in the permitted set as well. The inherited set contains all privileges, a program started by ssh-agent would get. All basic privileges are displayed here. However, because ssh-agent has permanently dropped

PROC_EXEC (it is not in the permitted privilege set any more) and therefore cannot execute any program any more you may ignore this set. The limit set defines the upper limit of privileges a process started by ssh-agent may inherit and contains all currently defined privileges in my OpenSolaris kernel.

You may also noticed the first line of ppriv's output. The privilege flag tells us that ssh-agent knows what privileges are and uses them to implement the least privilege principle. If you query legacy applications with ppriv, you will find out that they will not have set any privilege flags. ssh-agent is the first program that was adapted to use the new privileges. There are many daemons and programs that do not need the full file access power of the users who started them. Please feel free to modify them as well and discuss your results on security-discuss@opensolaris.org. Good candidates are the ones that are already privilege aware. Use ppriv to find out.

B Overview of the new introduced kernel functions: `secpolicy_vnode_gen_*` and `secpolicy_vnode_nanon_*`

All nine privileges introduced in section 5 were added to `common/os/priv_defs` so that they can be referenced inside the kernel. Because kernel modules do not directly check for the presence of privileges in certain privilege sets of the current process but use the secpolicy functions declared in `common/sys/policy.h` instead, I added declarations for the following functions that are all defined in `common/os/policy.c` there:

```
/*
 * Name:          secpolicy_vnode_gen_access()
 *
 * Parameters:    Process credential
 *               vnode
 *               permission bits requested by file related operation
 *               permission bits that were not granted by
 *               PRIV_FILE_GEN_{READ|WRITE|EXECUTE|SEARCH} or
 *               PRIV_FILE_NANON_{READ|WRITE|EXECUTE|SEARCH}
 *
 * Normal:       Checks whether required file access mode is generally allowed
 *               by probing whether the appropriate
 *               PRIV_FILE_GEN_{READ|WRITE|EXECUTE|SEARCH} privileges are set.
 *               Since PRIV_FILE_NANON_{READ|WRITE|EXECUTE|SEARCH} may substitute
 *               the former privileges, their presence is also checked.
 *               Permissions that were not fulfilled are stored
 *               in ng_mode
 *
 */
void secpolicy_vnode_gen_access(const cred_t *, vnode_t *, mode_t, mode_t *);
```

```

/*
 * Name:          secpolicy_vnode_nanon_access()
 *
 * Parameters:    Process credential
 *               vnode
 *               permission bits requested by file related operation
 *               permission bits that were not granted by
 *               PRIV_FILE_NANON_{READ|WRITE|EXECUTE|SEARCH}
 *
 * Normal:       Checks whether required file access mode is allowed for a
 *               (non) anonymous process by probing whether the appropriate
 *               PRIV_FILE_NANON_{READ|WRITE|EXECUTE|SEARCH} privileges are set.
 *               Permissions that were not fulfilled are stored
 *               in the last parameter.
 */
void secpolicy_vnode_nanon_access(const cred_t *, vnode_t *, mode_t, mode_t *);

/*
 * Name:          secpolicy_vnode_nanon_setdac()
 *
 * Parameters:    Process credential
 *
 * Normal:       Checks whether permissions of a file may be changed by
 *               evaluating the PRIV_FILE_NANON_{OWNER|READ|WRITE|SEARCH|EXECUTE}
 *               and PRIV_FILE_DAC_{READ|WRITE|EXECUTE} privileges.
 *
 * Output:       EPERM - if not privileged, 0 if not
 */
int secpolicy_vnode_nanon_setdac(const cred_t *, vnode_t *);

/*
 * Name:          secpolicy_vnode_nanon_create_directory()
 *
 * Parameters:    Process credential
 *
 * Normal:       Checks whether a directory may be created by
 *               evaluating the PRIV_FILE_NANON_{OWNER|READ|WRITE|SEARCH}
 *               and PRIV_FILE_DAC_{OWNER|READ|WRITE|SEARCH} privileges.
 *
 * Output:       EPERM - if not privileged.
 */
int secpolicy_vnode_nanon_create_directory(const cred_t *);

/*

```

```

* Name:          secpolicy_vnode_nanon_create_new_file()
*
* Parameters:    Process credential
*
* Normal:       Checks whether a new file may be created by
*               evaluating the PRIV_FILE_NANON_{OWNER|READ|WRITE|EXECUTE}
*               and PRIV_FILE_DAC_{OWNER|READ|WRITE|EXECUTE} privileges.
*
* Output:       EPERM - if not privileged.
*/
int secpolicy_vnode_nanon_create_new_file(const cred_t *);

/*
* Name:          secpolicy_vnode_nanon_create_symlink()
*
* Parameters:    Process credential
*
* Normal:       Checks whether a symlink link may be created by
*               evaluating the PRIV_FILE_NANON_OWNER and PRIV_FILE_OWNER
*               privileges.
*
* Output:       EPERM - if not privileged.
*/
int secpolicy_vnode_nanon_create_symlink(const cred_t *);

/*
* Name:          secpolicy_vnode_nanon_create_hardlink()
*
* Parameters:    Process credential
*
* Normal:       Checks whether a hard link to an owned file may be created by
*               evaluating the PRIV_FILE_NANON_OWNER and PRIV_FILE_OWNER
*               privileges.
*
* Output:       EPERM - if not privileged.
*/
int secpolicy_vnode_nanon_create_hardlink(const cred_t *);

/*
* Name:          secpolicy_vnode_nanon_sticky_remove_access()
*
* Parameters:    Process credential
*
* Normal:       Checks if removing, renaming a file or directory that
*               is contained in a sticky directory is allowed if ownership
*               of the file or directory is given and not otherwise privileged.

```

```

*           Checks presence of PRIV_FILE_NANON_OWNER.
*
* Output:   EPERM - if not privileged.
*/
int secpolicy_vnode_nanon_sticky_remove_access(const cred_t *);

```

The semantics of the new functions should be clear from section 5 and the comments above the function declarations. The parameters are self explanatory, in most cases only the process credential has to be passed²⁵. When querying for the new privileges, `priv_policy_choice` is used – so missing privileges will not be logged because they are ignored if behaviour bricks on higher levels in the same brick column are present²⁶.

Most of these functions are directly called by the file system drivers (see appendix C) but in some cases, already existent secpolicy functions had to be slightly changed to check some of the new privileges. This applied to the following functions:

- `secpolicy_fs_common()`: This function decides whether a given credential can act on a given mount. Overlay mounts may now require the `FILE_NANON_OWNER` and `FILE_NANON_WRITE` privileges if `FILE_OWNER` or `FILE_DAC_WRITE` are not present in the effective privilege set.
- `secpolicy_vnode setattr()`: This function checks the policy decisions surrounding the `vop setattr` call²⁷. Changing meta information may now require the `FILE_NANON_OWNER` privilege if `FILE_OWNER` is not present in the effective privilege set.

Some further helper functions locally declared in `common/os/policy.c` are not explained here. You may consult <http://myhpi.de/~nicolai/webrev/> for the complete source code.

C Porting guide: How to use the new kernel functions in a file system driver with the UFS implementation as example

If you would like to change a file system driver in order to support the new privileges explained in section 5, two options exist, depending on whether your driver is already conformant to Solaris 10 privileges or not. If it is a legacy

²⁵A process credential is a data structure that carries (besides many other things) the processes' effective privilege set.

²⁶If appropriate higher privileges are not present as well, their missing is reported as reason why the requested action was not granted by the operating system.

²⁷The `vop setattr` call is done by all file system drivers to decide whether a process is allowed to change meta information like permissions, access and modification times, ownership, owning group of a file or directory.

driver, you will have a lot of work to do, but if it already recognizes the privileges currently defined in Solaris 10, the integration process should be straight forward because for every new introduced privilege, a comparable privilege already exists in the same behaviour brick column on a higher row (level). Consequently, you only have to check your code for appropriate secpolicy function calls to find the places where to call the new introduced kernel functions illustrated in appendix B as well.

Typically, a Solaris 10 compliant file system driver calls the following kernel functions that are related with privileges in the same brick column as the privileges you like to integrate:

- `secpolicy_basic_link()`: At the place where this function is called, you also have to call `secpolicy_vnode_nanon_create_hardlink()` as done in `ufs_link()` in `ufs_vnops.c` for the UFS file system driver.
- `secpolicy_vnode_remove()`: At the place where this function is called, you also have to call `secpolicy_vnode_nanon_sticky_remove_access()` as done in `ufs_sticky_remove_access()` in `ufs_subr.c`.
- `secpolicy_vnode_access()`: At the place where this function is called, you also have to call `secpolicy_vnode_nanon_access()` and `secpolicy_vnode_gen_access()` as done in `ufs_iaccess()` in `ufs_inode.c` and `ufs_acl_access()` in `ufs_acl.c`.
- `secpolicy_vnode_setdac()`: At the place where this function is called, you also have to call `secpolicy_vnode_nanon_setdac()` before changing the ACLs of your files²⁸. See `ufs_acl_set()` in `ufs_acl.c` for an example how this was done in the UFS file system driver.

Furthermore, you have to add calls to the following new kernel functions:

- `secpolicy_vnode_nanon_create_symlink()` in the function you use to create symlinks. See `ufs_symlink` in `ufs_vnops.c` for an example how this was done in the UFS file system driver.
- `secpolicy_vnode_nanon_create_new_file()` in the function you use to create new files. See `ufs_direnter_cm()` in `ufs_dir.c` for an example how this was done in the UFS file system driver.
- `secpolicy_vnode_nanon_create_new_directory()` in the function you use to create new directories. See `ufs_direnter_cm()` in `ufs_dir.c` for an example how this was done in the UFS file system driver.

Of course, you may also consult the changes made in the TMPFS file system driver that are also available at <http://myhpi.de/~nicolai/webrev/> but

²⁸Because there is no single common standard for ACLs, it was not possible to integrate a general ACL granting procedure in `secpolicy_vnode_setattr()` that is called by all file systems when permissions should be changed. `secpolicy_vnode_setattr()` was already modified to adapt to the new privileges.

TMPFS does not support ACLs. While integrating the new privileges you should be aware that file system permission checks are done very frequently and therefore have to be very fast. Querying the new privileges should be only done when they are really needed and if possible only once. This appendix ends with a code snippet that was taken out of `ufs_acl.c` and demonstrates a quite efficient algorithm how to integrate the new kernel functions in an ACL checking routine:

```

/*
 * Check the inode's ACL's to see if this mode of access is
 * allowed; return 0 if allowed, EACCES if not.
 *
 * We follow the procedure defined in Sec. 3.3.5, ACL Access
 * Check Algorithm, of the POSIX 1003.6 Draft Standard.
 * After that, we also check the new introduced PRIV_FILE_NANON_*
 * and PRIV_FILE_GEN_* privileges
 */
int
ufs_acl_access(struct inode *ip, int mode, cred_t *cr)
/*
 *      ip      parent inode
 *      mode    mode of access read, write, execute/examine
 *      cr      credentials
 */
{
    ufs_ic_acl_t *acl;
    int ismask, mask = 0;
    int gperm = 0;
    int ngroup = 0;
    si_t      *sp = NULL;
    uid_t uid = crgetuid(cr);
    uid_t owner;
    mode_t general_access = 0;
    mode_t anonymous_access = 0;
    mode_t ordinary_access = 0;

    ASSERT(ip->i_ufs_acl != NULL);

    sp = ip->i_ufs_acl;

    ismask = sp->aclass.acl_ismask ?
        sp->aclass.acl_ismask : NULL;

    if (ismask)
        mask = sp->aclass.acl_maskbits;
    else

```

```

        mask = -1;

/*
 * (1) If user owns the file, obey user mode bits
 */
owner = sp->aowner->acl_ic_who;
if (uid == owner) {
    /*
     * Set ordinary_access to the subset of mode that was not
     * granted. Subtract the permissions in ordinary_access
     * from mode.
     */
    ordinary_access = (~(sp->aowner->acl_ic_perm << 6)) & mode;
    mode &= (~ordinary_access);
    goto anonymous_checks;
}

/*
 * (2) Obey any matching ACL_USER entry
 */
if (sp->ausers)
    for (acl = sp->ausers; acl != NULL; acl = acl->acl_ic_next) {
        if (acl->acl_ic_who == uid) {
            /*
             * Set ordinary_access to the subset of mode
             * that was not granted. Subtract the
             * permissions in ordinary_access from mode.
             */
            ordinary_access = (~((mask & acl->acl_ic_perm)
                << 6)) & mode;
            mode &= (~ordinary_access);
            goto anonymous_checks;
        }
    }

/*
 * (3) If user belongs to file's group, obey group mode bits
 * if no ACL mask is defined; if there is an ACL mask, we look
 * at both the group mode bits and any ACL_GROUP entries.
 */
if (groupmember((uid_t)sp->agroup->acl_ic_who, cr)) {
    ngroup++;
    gperm = (sp->agroup->acl_ic_perm);
    if (!ismask) {
        /*
         * Set ordinary_access to the subset of mode

```

```

        * that was not granted. Subtract the
        * permissions in ordinary_access from mode.
        */
        ordinary_access = (~(gperm << 6)) & mode;
        mode &= (~ordinary_access);
        goto anonymous_checks;
    }
}

/*
 * (4) Accumulate the permissions in matching ACL_GROUP entries
 */
if (sp->agroups)
    for (acl = sp->agroups; acl != NULL; acl = acl->acl_ic_next)
    {
        if (groupmember(acl->acl_ic_who, cr)) {
            ngroup++;
            gperm |= acl->acl_ic_perm;
        }
    }

if (ngroup != 0) {
    /*
     * Set ordinary_access to the subset of mode that was not
     * granted. Subtract the permissions in ordinary_access
     * from mode.
     */
    ordinary_access = (~((gperm & mask) << 6)) & mode;
    mode &= (~ordinary_access);
    goto anonymous_checks;
}

/*
 * (5) Finally, use the "other" mode bits
 */
/*
 * Set ordinary_access to the subset of mode that was not
 * granted. Subtract the permissions in ordinary_access
 * from mode.
 */
ordinary_access = ~(sp->aother->acl_ic_perm << 6) & mode;
mode &= (~ordinary_access);

anonymous_checks:
if (mode != 0) {
    /*

```

```

    * Check mode for execute/search/read/write permissions and
    * check whether PRIV_FILE_GEN_{READ|WRITE|SEARCH|EXECUTE} or
    * its substitutes (PRIV_FILE_NANON_{READ|WRITE|SEARCH|EXECUTE})
    * are set.
    * Record missing permissions in general_access.
    */
secpolicy_vnode_gen_access(cr, ITOV(ip), mode, &general_access);

/*
 * Subtract the permissions in general_access from mode
 */
mode &= (~general_access);

if (mode != 0) {
    /*
     * Check whether the access with mode would be possible
     * if the permissions gained through the process'
     * euid/egid and associated groups of the process
     * would not be present.
     * Set mode to the subset of mode
     * that was not granted.
     */
    mode &= ~(sp->aother->acl_ic_perm << 6);

    if (mode != 0)
        /*
         * For every permission contained in mode
         * check whether the corresponding
         * PRIV_FILE_NANON_{READ|WRITE|SEARCH|EXECUTE}
         * privilege is set. If not, add the permission
         * to anonymous_access.
         */
        secpolicy_vnode_nanon_access(cr, ITOV(ip),
        mode, &anonymous_access);
    }
}

/*
 * Set mode to the missing permissions and check if these are granted
 * due to special privileges (PRIV_FILE_DAC_{READ|WRITE|EXECUTE|SEARCH})
 */
mode = anonymous_access | ordinary_access | general_access;
if (mode == 0)
    return (0);
return (secpolicy_vnode_access(cr, ITOV(ip), owner, mode));
}

```

D Sources: Where to find further information

The Google Summer of Code project is located at <http://code.google.com/soc/>.

The OpenSolaris communities and the OpenSolaris source code are located at <http://www.opensolaris.org>. For my job, especially the

- security community (responsible for all security related things in OpenSolaris and therefore also for the privileges, provided lots of input while discussing my new privileges)
- Nevada Project community (maintains the kernel)
- UFS community (provides information about the internals of the UFS file system driver)
- ZFS community (provided lots of input while discussing my new privileges)
- network community (provided lots of input while discussing my new privileges)
- immigrants community (provided help for people like me who came to Solaris from other Unices)
- documentation community (pool for OpenSolaris related information)

were very helpful.

If you are interested in further information about programming traditional Unices and how to cope with its security models, take a look into *Advanced Programming in the Unix Environment* by Richard Stevens (<http://www.kohala.com/start/apue.html>).

If you like a deeper sight into Solaris 10 privileges I can recommend the `privileges(5)` manual page and the blog of Casper Dik (you may start reading with this entry: <http://blogs.sun.com/casper/20040722>).

If you really like to dig into the OpenSolaris kernel you may have a look into the book *Solaris Internals 2* (<http://solarisinternals.com>) that was written by Jim Mauro, Richard McDougall and Brendan Gregg. Here you may also find further information about the Solaris 10 privilege model.

My contributions to the OpenSolaris kernel in a reviewable form can be found at <http://myhpi.de/~nicolai/webrev> and perhaps later in the official OpenSolaris kernel. This paper is available on <http://myhpi.de/~nicolai/GSoC.pdf>. The modified `ssh-agent` sources may be reviewed at <http://myhpi.de/~nicolai/webrev-ssh-agent>. For further information or discussions just contact me at jonico@users.sf.net.