Privilege Separation

A Security Pattern

Dan Forsberg, <dan.forsberg@hut.fi>
Helsinki University of Technology;
Nokia Research Center

Abstract

When *Privilege Separation* pattern is used it divides one functional element into smaller functional elements with different privileges and restricted interfaces. The intent is to separate privileges of functional entities and thus restrict the area where the functional entity has rights to act. When properly and wisely applied makes the system more secure, modular, and easier to analyze by dividing an entity into multiple entities.

CONTEXT

Server software programming in many times requires special privileges for doing certain operations like binding the server into a service ports, accessing files with confidential information like passwords, and managing cryptographical operations like data signing and encryption with confidential session keys.

PROBLEM

The problem that this pattern solves is *how to minimize the effects of vulnerable exploited code* (like buffer overflows).

FORCES

There exists a system with multiple functionalities and valuable information and only part of the system functionalities need to access this information. Unintended leakage of the information must not happen between functional elements.

A system needs to access secure information (continuously), which means that some kind of access control to the information must be implemented. This pattern becomes useful if the whole system does not need the information as it is, but a derivation of it (like authentication result, handle to a socket/file descriptor, etc.).

On the other hand if this pattern is not used at all the system modularity does not exist and probability for security vulnerabilities increases. A bad example would be a server that needs high privileges for a small amount of time to do a small task, but the whole server runs all its lifetime with high privileges.

When dividing the system into smaller systems, the complexity of the combined system increases. As a result of the division into multiple entities the interfaces between the entities must be specified. The more interfaces, the more work is needed. In both of these cases a balance between complexity and sound security need to be found.

SOLUTION

Normally modularity is based on functional aspects. Privilege separation pattern brings additional modularity based on different privileges. In addition to using this pattern the least privilege security principle should be applied to the resulted divided entities for providing a full solution to the problem.

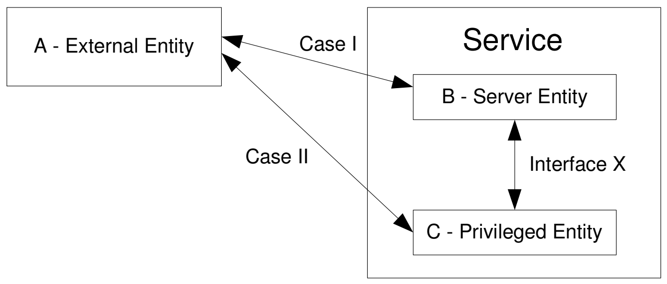

Figure 1 An example of privilege separation pattern structure

Figure 1 illustrates a simple structure of the privilege separation pattern. External Entity (A) uses a service which consists of a combination of the Server Entity (B) and the Privileged Entity (C). Server Entity and the Privileged Entity co-operate together via interface X for serving the External Entity.

Privileged entity has access privileges to sensitive or valuable resources. When applying this pattern the service entity is divided into two entities with separated privileges: server entity and privileged entity. The privileged entity holds privileges to access valuable resources and the server entity communicates with it.

The privileged entity derives information and/or resources for the usage of the server entity. In Case II this information can be a result of processing between the External Entity and the Privileged Entity, like authentication result from SIM card or a file descriptor from operating system to a user space program. In Case II the external entity communicates with the server entity. The privileged entity provides information like checking if the supplied password was correct according to a passwd file in the system.

Depending on the resources that the privileged entity provides, the interface to the service consumer entity can be through the server entity (Case I, see Figure 2) or through the privileged entity (Case II, see Figure 3).

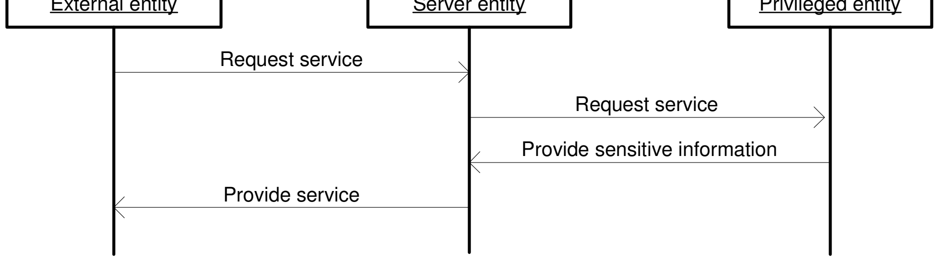

Figure 2 Case I

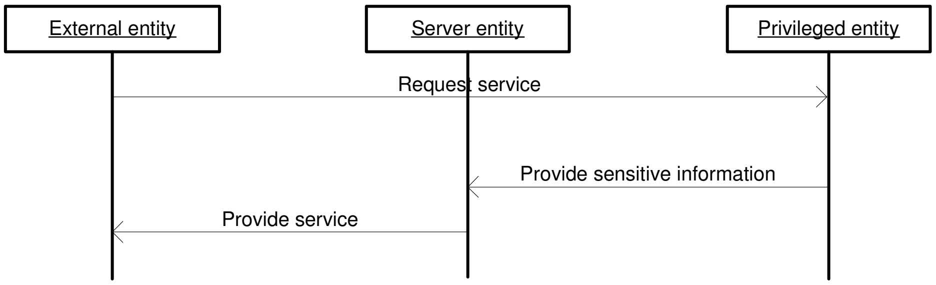

Figure 3 Case II

If the pattern is recursively (see Figure 4) applied too many times, the system becomes cumbersome and complicated to manage. An example would be a system that implements different access control lists (ACL) for each and every file, application, and resources. Using the same access rights for multiple files (access rights group) is thus used to make the system more manageable.

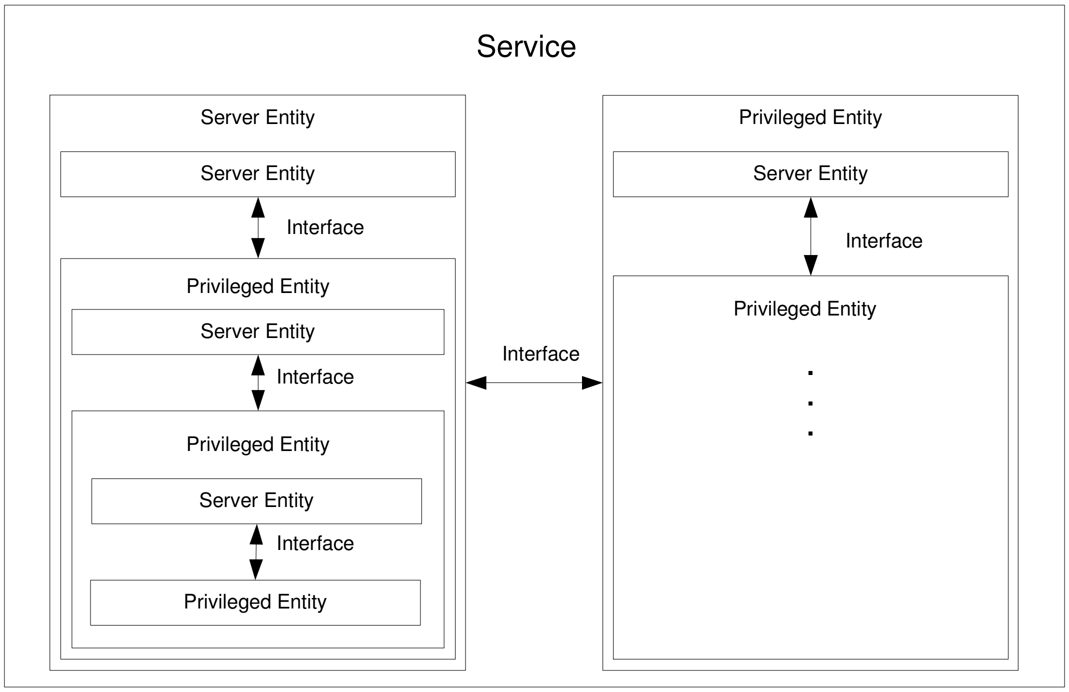

Figure 4 Simple example of the recursive nature of the *Privilege Separation* pattern

RESULTING CONTEXT

As a result the systems privileges are divided and only part of the system is permitted to access privileged information. If wisely applied vulnerable code in one entity does not break the security of the other entities or at least makes it harder to exploit the code vulnerabilities.

KNOWN USES

- OpenSSH privilege separation [1], [2] is an example where the SSH server was divided into two functional entities for better security. Also `vsftpd` uses privilege separation to limit the effect of programming errors [3].
- When designing network architecture isolating long-term security credentials into separate servers in the network architecture to better protect them. This separate server would then have an interface to other selected network interfaces that are eligible to contact the server. Examples include isolated databases with access control like passwd file and programs/libraries that can access it and a HLR register in telecommunications systems architectures.
- When managing and using security credentials. Caging security credentials with hardware approaches, like SIM cards in mobile phones.
- It is important to restrict the rights of an executing process in an Operating System. Patterns like *File Access Control*, *Controlled Virtual Address Space*, and *Controlled Execution Environment* [4] are tools when providing privilege separation, which helps minimizing the scope of security threats for network servers. An extreme example of privilege separation is `chroot(1)`("change system root") system capability in Unix systems. It provides strong process isolation and actually implements also the patterns listed above.
- Applying this pattern for the software design and development tools, especially to the graphical user interfaces, could mean that the developer would be able to separate privileges by painting areas of code with mouse or selecting files. Then the compiler together with the target system could provide different privileges to these areas. This seems to be a novel idea. A target system could be an operating system for example. Separation could be a combination of processes and files. On the other hand the compiler and OS kernel could support privilege changing for a process or thread by inserting code in the compilation phase into the executable binary. This inserted code would then automatically change the privileges of the process/thread. How to select the privileges is out of the scope of this paper. A configuration file could be used for example.

Related Security Patterns

Single Access Point security pattern creates a single interface for communication with external entities. After our privilege separation pattern has been used *Single Access Point* security pattern can be used for the communication between resulting entities. However, the divided entity may have other interfaces towards the external entities, which thus breaks the *Single Access Point* security pattern model.

Applying Layered Security pattern makes the system to have multiple levels of security checks. When *Privilege Separation* pattern is applied it may provide or create another security layer, which fulfills the goal of the *Layered Security* pattern. On the other hand the same layer may be used multiple times (for example file access rights), even if the *Privilege Separation* pattern is used.

Reference Monitor security pattern [4] can be applied to the privileged entity on our pattern. It defines a process that intercepts all requests for resources and checks if the requests are authorized or not. When applying *Privilege Separation* pattern the privileged entity becomes a reference monitor for the valuable information, in our example case the server entity (see Figure 1). The *Authenticator* security pattern [5] can also be applied to the privileged entity if the origin of the request needs to be authenticated.

Related Principles

Least privileges security principle should be applied to the resulting entities after the *Privilege Separation* pattern has been applied. *Privilege Separation* pattern supports the *defense in depth* security principle, since it creates new entities and separates their privileges.

Acknowledgements

We would like to thank Eduardo Fernandez for kindly sheperding this paper through and VikingPLoP´05 conference participants for giving very valuable feedback for a pattern writer newbie.

References

[1] Niels Provos, Peter Honeyman; “Preventing Privilege Escalation”; 12th USENIX Security Symposium Proceedings, 2003; URL: http://www.usenix.org/publications/library/proceedings/sec03/tech/provos_et_al.html

[2] David Brumley and Dawn Song; “Privtrans: Automatically Partitioning Programs for Privilege Separation”; 13th USENIX Security Symposium Proceedings 2004; URL: http://www.usenix.org/publications/library/proceedings/sec04/tech/brumley.html

[3] Chris Evans, “Probably the most secure and fastest FTP server for UNIX-like systems”, URL: http://vsftpd.beasts.org/

[4] E.B.Fernandez, "Patterns for operating systems access control", Procs. of PLoP 2002, http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html

[5] E.B.Fernandez and J.C.Sinibaldi, "More patterns for operating systems access control", Procs. EuroPLoP'03, 381-398, http://hillside.net/europlop/europlop2003/