# Server Sandbox

(a.k.a. Privilege Drop, Untrusted Server, Constrained Execution Environment, Unprivileged/Restricted User Account, Run as Nobody, Demilitarized Zone)

## Abstract

Many site defacements and major security breaches occur when a new vulnerability is discovered in the Web server software. Yet most Web servers run with far greater privileges than are necessary. The *Server Sandbox* pattern builds a wall around the Web server in order to contain the damage that could result from an undiscovered bug in the server software.

## Problem

A server-based application is typically exposed to a huge number of potentially malicious users. Any application that processes user input could potentially be tricked into performing actions that it was never intended to perform. For example, many Web servers contain logic errors that can be exploited to allow private files to be served over the Internet. Other servers contain undiscovered buffer overflow errors that can allow client-provided malicious code to be executed on the server.

While every attempt should be made to prevent these types of errors, it is impossible to anticipate every possible attack beforehand. Therefore, it is prudent to deploy a server application in a manner that minimizes the damage that can occur if the server is compromised by a hacker.

Web applications generally require little in the way of privileges once they are started. But by default, many servers and applications install in a manner that gives them unnecessary and dangerous privileges, that if compromised could lead to significant security breach.

For instance, Web servers running on the UNIX operating system must be started with administrative privileges in order to listen on port 80 – the standard HTTP port – which is a privileged port. Likewise, the Microsoft IIS default installation executes the Web server using the privileged SYSTEM user. If a Web server running with administrative privileges is compromised, an attacker will have complete access to the entire system. This is widely considered the single greatest threat to Web site security [1].

## Solution

The *Server Sandbox* pattern strictly limits the privileges that Web application components possess at run time. This is most often accomplished by creating a user account that is to be used only by the server. Operating system access control mechanisms are then used to limit the privileges of that account to those that are needed to execute, *but not administer or otherwise alter*, the server.

This approach accommodates systems that require administrative privileges to start the application, but do not need those privileges during normal operation. The most common example of this is a UNIX server application that must listen on a privileged port. The application can start with additional privileges, but once those privileges are no longer needed, it executes a *privilege drop,* from which it cannot return, into the less privileged operating mode.

There are a number of different operating system specific privilege drop mechanisms. Some of the more common are:

- An application can switch the user account under which it is executing at run-time. For example, a UNIX application can switch from running with administrator privileges to a specific server account or even the *nobody* account.

- An application can inform the operating system that it wishes to drop certain privileges dynamically. This is common in capability-based systems, where the operating system dynamically maintains a list of application capabilities. In Linux, an application can ask the operating system to make entire APIs invisible for the remainder of the lifetime of that process.

- An application can instruct the operating system to no longer accept any changes that it requests. For example, once a Linux system has fully booted, it can instruct the operating system to no longer allow kernel modules to be dynamically loaded, even by the administrative account.

- An application can be executed within a virtualized file system. The UNIX chroot option allows the application to think it can see the actual file system, when in fact it only sees a small branch set aside for that application. Any changes to the system files it sees will not affect the actual system files.

The *Server Sandbox* pattern also requires that the remainder of the system hosting the server be hardened. Many operating systems allow all user accounts to access certain global resources. A server sandbox should remove any global privileges that are not essential and replace them specific user and group privileges. A compromised Web server will allow an external hacker to gain access to all global resources. Eliminating the global privileges will ensure that the hacker will not have access to useful (and potentially vulnerable) utilities and operating system features.

The *Server Sandbox* pattern partitions the privileges required by the server between those needed at server startup and those needed during normal operation. For example, UNIX systems require administrative privileges to create a server listening on port 80, the standard HTTP port. However, the server should not possess administrative privileges at run-time. A server sandbox allows dangerous privileges to be used to create the server but then revoked before the server is exposed to client input.

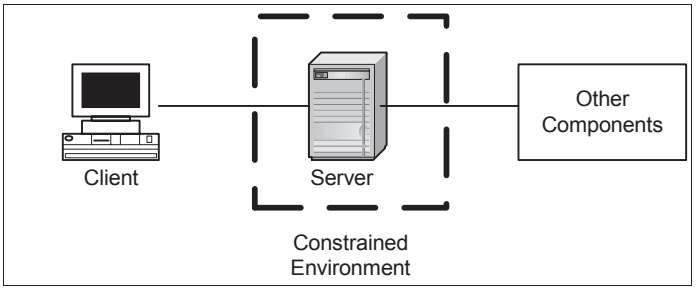
While the most common implementation of the *Server Sandbox* pattern relies on a restricted user account, other (additional) implementations are possible, including:

- Creating a virtual file system and restricting the server so that it cannot see files outside of

this space (chroot).

- Putting wrappers around dangerous components that limit the application's ability to access resources and operating system APIs

- Using operating system network filtering to prevent the server from initiating connections to other machines




## Issues

It is critical that the application be developed within the envisioned constrained environment. Attempting to add the constrained environment after the fact generally breaks the application and often results in the constrained environment being unnecessarily relaxed in order to resolve the problem. For example, most IIS applications are developed using the standard, insecure configuration, in which IIS executes as SYSTEM. If an individual administrator attempts to configure his or her server more securely and run IIS using a less privileged account, many of these applications will fail to execute properly.

Building the application within the constrained environment also ensures that any performance or resource usage impact will be uncovered early in development.

It is important to document the security configuration in which the system is expected to execute. If the application requires specific privileges to specific files and services, this information must be provided to the administrator configuring the system. It is not sufficient to merely provide an installation script that sets all the appropriate options, because many administrators need to fine-tune the installation afterwards or install other applications that may alter the security configuration of the system. If the administrator is not aware of the minimum required privileges, he or she may give the application unneeded – and potentially dangerous – privileges. This often translates to executing the application with full administrative privilege.

Many operating systems install in an insecure state. Employ general hardening techniques to eliminate weaknesses. On many systems, the Operating System access control model can be bypassed. If an outsider is able gain control over a general user account, it can be fairly straightforward to exploit a weakness in a system application to gain root/administrator privileges. If possible, the restricted user account should be limited to executing only those programs that it requires.

**Possible Attacks**

There are a number of possible attacks that could be perpetrated against this pattern:

- Buffer overflow attacks – buffer overflow attacks on the server are the most common approach to remote compromise of the server. The sandbox is intended to contain the damage of such an attack.

- Privilege escalation – if an attacker is able to compromise a Web server, even one running as nobody, they will be able to execute code on the system. Attackers typically attempt to break out of the sandbox by exploiting vulnerabilities in other privileged applications, such as sendmail. If a vulnerable, privileged application is accessible to the restricted user account, a privilege escalation attack is possible.

- Breaking out of the sandbox – if the sandbox mechanism contains bugs, an attacker may be able to exploit them to break out of the sandbox. If the attacker can somehow gain root privilege, many sandbox features (such as chroot) are reversible.

- Snooping – if an attacker is able to exploit a server vulnerability and gain a toehold on the system, they may have enough privilege to monitor further server operations. They could capture passwords or other sensitive data. If the server has privileges to access a back-end database, the attacker will have those same privileges.

- Application level exploits – even if the server is perfectly sandboxed, it may still suffer from application-level vulnerabilities. The remote attacker may not have to compromise the server in order to misuse its services.

# Examples

At the code level, Java provides the most widely known implementation of a sandbox. It prevents the user from using features and functions that are outside of the Java security policy [2], [3].

At the system level, the canonical example of this pattern is the Apache Web server, which by default runs as user nobody. Although root privileges are required to start the server on port 80, the server drops into the nobody account after initialization.

The nobody account is able to read (but not write) all of the public html files on the server. But a well-configured server will disallow the nobody account from executing any commands or reading any other files.

Similarly, the Netscape Enterprise Server (iPlanet Web server) for UNIX uses the nobody account. If it is instructed to listen on a privileged (<1024) port, it must be started as root. However, once the port is established, it switches to the nobody account before accepting client connections.

At the network level, it is common practice to place a Web server outside the corporate firewall, or in a *Demilitarized Zone* (DMZ) between the Internet and the internal network. In either case, a firewall separates the Web server from the rest of the internal network. This is an example of a network-level server sandbox: the Web server is only allowed to connect to a handful of specific ports on one or more specific trusted machines on the internal network. In some configurations, the connections must be initiated from the internal network—in this case, the DMZ represents a sandbox in the purest sense.

## Trade-Offs

| | |
|---|---|
| **Accountability** | No direct effect. |
| **Availability** | No direct effect. |
| **Confidentiality** | No direct effect. |
| **Integrity** | This pattern will greatly enhance integrity by preventing component vulnerabilities from causing the entire server to be compromised. |
| **Manageability** | This pattern will affect the manageability of the software in question because constrained execution environments often incur overhead to setup and maintain. |
| **Usability** | No effect. |
| **Performance** | This pattern will often have a negative effect on performance, but this will depend on the specific techniques used. Using chroot or unprivileged user accounts do not affect performance. Other techniques that impose additional runtime validity checks will incur a performance penalty. |
| **Cost** | This pattern will increase development costs somewhat. This can be minimized if the application is developed with the constraints already in place. Retrofitting an existing application is much more difficult. |

## Related Patterns

- *Minefield* – a related pattern; any atypical behavior by the restricted user account should be a source of intrusion information.

- *Partitioned Application* – a related pattern; a complex application may require some dangerous privileges throughout its execution time. In that case, the application should be partitioned so that only a minimal component has the dangerous privileges. The other

component(s) should run using restricted user account(s). Components that communicate directly with clients should have the bare minimum privileges. Components with dangerous privileges should be buffered from client requests by other components.

# References

[1] Stein, L. and J. Stewart. "The World Wide Web Security FAQ – Version 3.1.2". http://www.w3.org/Security/Faq, February 2002.

[2] Sun Microsystems. "Secure Computing with Java: Now and the Future". http://java.sun.com/marketing/collateral/security.html, 1998.

[3] Venners, B. "Java's security architecture: An overview of the JVM's security model and a look at its built-in safety features". http://www.javaworld.com/javaworld/jw-08-1997/jw-08-hood.html, August 1997.