Secure Pre-forking - A Pattern for Performance and Security

Munawar Hafiz
University of Illinois at Urbana-Champaign
e-mail: mhafiz@uiuc.edu

January 27, 2006

Abstract

In a multi-tasking environment, the standard practices for performance improvement are pre-forking and resource pooling [11] of processes. However, if the processes in the resource pool have infinite lifetime, a malicious attacker can utilize this feature by compromising one process and then using it to attack and compromise other processes. This paper presents a performance and security pattern for resource pooling. The secure pre-forking pattern prescribes that limited lifetime is an essential feature for the processes in the resource pool. Older processes are periodically evicted from the resource pool and new processes are spawned in their place. This pattern therefore mitigates the trade-off between performance and security.

Background

This pattern is listed in [7] with other security patterns from the domain of Mail Transfer Agent (MTA) architecture. A *Mail Tranfer Agent* is a computer program that transfers electronic mail messages from one computer to another. The most popular MTA is *sendmail*. Other Unix compatible MTAs are *qmail, Postfix, exim, Courier, ZMailer etc.*The main requirement for an MTA architecture is security. But sendmail was not designed with security in mind, because the primary architectural requirement at the time of its development (early 1980) was flexibility. The architecture of qmail is motivated by the series of security breaches in sendmail. However, qmail is not only more secure than sendmail, it is also more efficient and easier to understand. The security of qmail is based on a few patterns [9] and understanding its architecture can help people make other applications secure. Postfix has gained popularity as an MTA because it has the same interface like sendmail but it does not have problems with security and reliability. The architecture of Postfix closely follows the design principles of qmail. A lot of security patterns of the qmail architecture are found in Postfix. Additionally, Postfix improves performance. It shows better performance than qmail and sendmail in benchmark tests.

The pattern presented in this paper comes from Postfix architecture. The performance advantage of Postfix comes from its adoption of security patterns that also add to the overall system performance. The server process that handles incoming mails from remote MTAs binds and listens to the SMTP port. When a mail send request comes from a remote MTA, the listener process forks a Postfix child process to handle this task, hands the task over to that child process and goes back listening to the port. To reduce the process creation overhead, child prcesses are pre-forked and kept in a resource pool in Postfix. However, if the child processes have infinite lifetime it creates a vulnerability that can be exploited by malicious attackers. Secure Pre-forking pattern removes this vulnerability by assigning a limited lifetime to the processes.

The example of Apache web server is used in the pattern as a motivating example. This illustrates the applicability of this pattern to solve a more general problem rather than a specific problem pertinent to the domain of MTA architecture.

Secure Pre-forking

Performance, Security

Pattern Name
Secure Pre-forking

Intent
In a multitasking server environment, pre-forking is a widely used mechanism for improved performance. If the pre-forked processes run as daemon processes, then they pose security and reliability risks. When an attacker compromises one of the daemon processes, he can use this process to try to infect other processes more effectively because the daemon processes never die. To avoid this vulnerability, all the pre-forked processes should be created with a limited lifetime and new processes are forked in their places after this pre-defined time span.

Also Known As
Mortal Daemon.

Motivation
Apache is a well-known web server that runs on Unix as a pre-forking server. On startup, it creates a pool of child processes ready to handle incoming clients. As requests are processed, Apache tries to make sure that there are at least a few spare servers running for subsequent requests. If the process spawning cost is negligible, procsses do not need to be pre-forked. Instead a bunch of listeners listen to web requests and fork processes lazily. In practice, process creation has significant overhead. Pre-forking enhances the performance of the server by reducing the process creation overhead when a connection request comes through. However, this architecture has some security issues.

The pre-forked processes reside in a resource pool. When a request comes, the listener process hands the task to an idle process taken from the resource pool. Once the task is finished, the process returns to the resource pool. These processes remain resident until the server is shut down. The consequences of a security compromise with these processes are more severe, because the processes give the attacker a working ground that remains resident in the system for a long time. This can be utilized to further exploit other processes.

How can the vulnerability associated with daemon processes be minimized?

The solution is to limit the lifetime of daemon processes. The processes are monitored so that they are killed after serving a pre-configured number of requests. The identification criterion for the processes that need to be killed is not a single paramter like the number of requests served. Multiple parameters are considerd for this. The identified process is killed and a new process is forked and included in the resource pool in its place. Information about the new process is passed to the monitor mechanism for control purpose.

Apache uses this approach. Apache has various configuration parameters that are used during server startup and process management. Startup parameters are used by the master server to spawn off a number of child processes and to put them in a resource pool. Memory resident data structures are maintained by master server for process accounting. Based on these data values, processes are replaced after their limited lifetime.

Applicability

Network servers handling concurrent requests usually exhibit multitasking architecture with resource pooling. This pattern is used to make the resource pooling mechanism more secure. Normally a system architect uses this pattern.

A simple example of such a multitasking handler is a master server process acting as a gatekeeper to the system. It waits for a connection request and when the request arrives, it creates the connection and forks itself to create a child process. The child server process handles the request and runs in parallel to the master server that returns to listening for incoming requests.

Web servers handle a lot of incoming requests; hence performance is the key issue. If process forking takes a significant amount of time, the lazy forking by the master server means that it cannot accept incoming requests during this period. This lack of availiability if the main reason why on-demand forking architecture cannot be used for network servers.

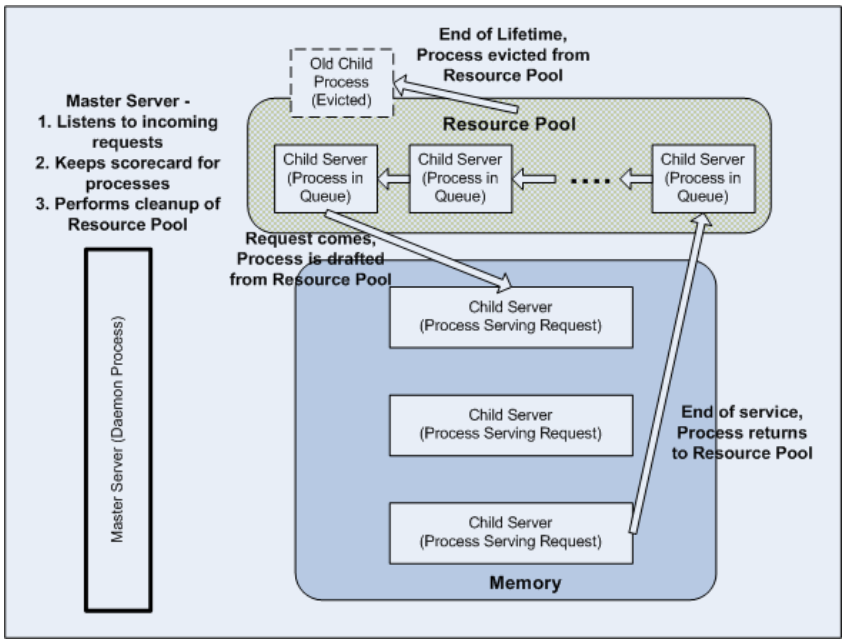
The forces that need to be considered when choosing to use this pattern are as follows.

- *Performance.* Released resources should be reused. This reduces the overhead associated with repititive acquisition and release.
- *Latency.* The latency for serving an incoming request should be minimal.
- *Security.* The impact of a security breach should be minimal so that the compromise of one process is limited and cannot be used for further perpetration.
- *Lifetime of Processes.* Processes should not be reused infinitely. They should be released after some time. Long running processes have the same vulnerability. Such processes have to be monitored to find if they are compromised or not.
- *Availability.* The clients expect the server to be available and they expect that their service requests will not be refused.
- *Simplicity and Loose Coupling.* The implementation should not introduce more bugs into the system. The monitoring logic should be separated from the main functionality of the system.
- *Overhead of process monitoring.* The execution overhead for running the secure pre-forking mechanism should be minimal. Process monitoring should not take up the CPU cycles saved by resource pooling. The cost for releasing resources during execution time and forking new processes in their place should be minimal.

The resource pooling pattern resolves the performance and latency aspects. Without the limitation of process lifetime the system remains vulnerable to a security breakdown attempt.

Structure

Here the structure of the pattern illustrates the key components of the pattern.



Participants

- **Master Server Process**
  The master server process listens to incoming requests. It keeps usage statistics of pre-forked processes and sends cleanup commands.
- **Child Server Process**
  The child server processes are pre-forked. They serve the incoming requests. Upon creation, the child server processes reside in the resource pool. When needed, they are transferred to main memory. When the task is finished, the processes are returned to the resource pool where they wait to serve future requests. After a limited lifetime, the child server processes are evicted.
- **Resource Pool**
  The pre-forked child-server processes wait in the resource pool.
- **Memory**
  The child server processes run in the main memory when they are handed incoming requests. The processes return to resource pool after the service completion.

Collaborations

Initially, the master server pre-forks a number of child processes and puts them in the resource pool. This pre-forking stage usually happens before the master server starts listening for client requests. The number of pre-forked processes is configured by parameters set by the system administrator. This number can change during runtime based on the number of requests coming from the client. Clients submit their connection requests to the master server. The request is assigned to one child process from the resource pool. After finishing, the child process returns to the resource pool. The master server keeps track of information about child processes, *e.g.*the length of time they are serving, the number of requests serviced *etc.*These parameters

are configured through some external configuration file.

Consequences
The pattern has the following benefits.

1. *Increased security.* In a process based system architecture, the key principle for a secure architecture is the insulation of processes. An example is the architecture of qmail. qmail is a secure mail transfer agent. The primary tasks of a mail transfer agent are receiving mail from local and remote hosts, storing the mail, and delivering the mail to local and remote recipients. In the qmail architecture, the processes are partitioned according to their functionalities. There are separate processes for local and remote mail recipients, the queue manager and local and remote mail senders. These processes do not trust each other and validate the communication payload between them. Hence even if some process is compromised, other parts of the system remain unharmed because they run in a separate address space. In qmail, most of the processes are not daemon processes, and so even if some process is compromised, after a brief period it dies and is garbage collected. The attacker cannot use this process to try to compromise other processes. Even if some attacker does not try to infect other processes, he can send a garbage payload to other processes and create an internal Denial of Service (DoS) scenario. Secure pre-forking adds to the security of the overall system by limiting the lifetime of the daemon processes so that the effect of a process compromise becomes transient.

2. *Improved Performance.* Secure pre-forking improves performance because processes do not have to be spawned for each incoming request. If the pre-forked processes run as daemons, it should have even better performance because the cost of process creation is entirely eliminated. However, in this scenario security is an equally important requirement along with performance. The limited lifetime of processes adds some overhead of forking more processes, but this limit adds a lot to the entire system being secure.

3. *Availability.* The main problem with a server that forks processes on demand is the process creation overhead incurred every time a request comes in. When the master server is forking some child server process, the requests that come at that moment are not served. Normally this happens only if the delay for forking a process is not negligible. However if the load is high the delay for forking cannot be avoided. It is inconvenient for the communicating entity to find their requests refused by the server. This is not a serious problem in mail transfer agent architecture as processes communicate with the server and they can start again. However, for web server architecture, the communicating entity is a human user and it is a major disturbance for the client to see his web page access request denied because the server was unavailable.

4. *Defense in Depth.* When the processes run under an owner with a low privilege level, an attacker after compromising a process cannot do a lot of harm. In that case it can be argued that running daemon processes may not be harmful. The counter argument is that daemon processes can still be vulnerable and it might still contain a hole that comes from bad programming. Running the processes as daemon even at a low privilege level is dangerous. The secure pre-forking pattern follows the defense in depth principle [12] by limiting the vulnerability associated with processes with long lifetime.

The pattern has the following liabilities.

1. *Complexity of pre-forking architecture.* The major problem with secure pre-forking architecture is the complexity. This means more faults, less portability, and a larger binary. The performance improvement of pre-forking only becomes evident in the case of heavy load. In the case of light load, the pre-forked processes

occupy memory and become a bottleneck instead.

2. *Negative impact on availability.* This pattern can lead to limited availability. Let us suppose that a batch of pre-forked processes have reached their threshold and should be killed. However, at that moment the server is experiencing heavy load. If the master server begins pre-forking new processes, some incoming requests may go unserved. Two things can be done to counter this scenario, a) killing the processes but not forking new processes in the resource pool or b) temporarily halting the killing of processes. If the processes are killed but new processes are not spawned off then the processes remaining in the resource pool may be insufficient to handle the incoming load. In that case, after some time the server may have to lazily fork new processes. Halting the killing of processes is another option. However, this solution is temporary and after a short delay the marked-down processes are killed. If the stoppage is not temporary, it leads to another vulnerability. An attacker, after the compromise of one process, may launch a DoS attack to keep the master server in heavy load. A heavy load means the server cannot kill processes. So the processes effectively become daemon processes. To avoid this, after some time the processes that have passed their life limit are killed anyway. This is an example of trade-off between availability and security.

Implementation

Here are the issues of implementing the pattern.

1. *Criteria for process killing.* A key issue in pre-forking is to determine what parameters to consider when replacing the pre-forked processes. An obvious parameter is the number of requests served by a process or the duration of service. However, this parameter alone is insufficient as attackers are tempted to launch their attacks during low-traffic or off-peak hours. Hence the idle time of processes should also be considered. Another related parameter is the overall lifetime of processes. This parameter alone cannot be used. Processes are pre-forked as a group and if the only parameter is the total lifetime then several of them reach the threshold together. The key is to consider multiple parameters.

2. *Setting up the pre-forking parameters.* A number of factors related to the pre-forking architecture have to be configured. Separate parameters are maintained for controlling these issues externally. For example, the number of pre-forked processes can be controlled with parameters. If the value is too high then the server startup time is increased. If the value is too low then soon after startup the server will require more processes to handle incoming requests and it will have to spawn new processes. The minimum and maximum value of parameters are set according to some empirical values seen from trial runs of the software.

3. *Smart pre-forking.* In many systems, pre-forking is handled by parameters specified through configuration files (see the example of parameters in Apache implementation example). A simple approach to pre-forking uses the parameters directly. For example, if ‘x’ is the number of resources specified through some external configuration, then ‘x’ resources are pre-forked. A smart pre-forking approach reduces the overhead of pre-forking by creating processes based on load. The system can use an algorithm that incrementally increases the number of processes spawned dynamically in response to increasing load. A pre-forking server may spawn a fixed number of processes and then double this number when more requests come and so on. An example is given in the Apache implementation section.

4. *Process monitoring.* The processes have to be monitored to do accounting on the creation time and use of processes. A separate data structure has to be maintained in memory to do that. Careful monitoring has to be done so that it does not become a performance overhead.

5. *Pre-forking of resources.* The idea of resource pooling can be extended to pre-fork instances of threads, sockets etc. Windows implementation of resource pooling is primarily based on thread pooling.

6. *Freeing of resources.* Process eviction is an important aspect of the pre-forking architecture. The process eviction task can be entrusted to a daemon process. The process is simple and does nothing else than freeing of resources. This simplicity means that it can be thoroughly tested and made bug free.

7. *Deadlock recovery in secure pre-forking.* Pre-forking works best with systems where the incoming requests are for short-lived tasks. If the processes handle tasks that take a long time it also makes the system vulnerable. The processes therefore go through an auditing phase and the processes that are blocked for a long time are identified and killed. This has an additional advantage. The pre-emptive killing of a process that is blocked for a long time provides a deadlock recovery mechanism. However, the most difficult issue is to identify whether a process, that is handling a task that naturally takes a long time, is compromised or not. If a process that was running under normal circumstances is killed, the task entrusted to the process is not finished. This has severe reliability and availability issues.

Known Uses

We provide here two examples from Unix server domain. One of them is the Apache web server. The other is the Postfix mail transfer agent.

1. *Implementation in Apache.* Apache runs on Unix platforms as a pre-forking server. On startup, it creates a pool of child processes ready to handle incoming client requests. As requests are processed, Apache tries to make sure that there are at least a few spare servers running for subsequent requests. Apache provides three directives to control the resource pool.

StartServers $<$ number $>$ (*default* 5)
This determines the number of child processes Apache will create on startup.

MinSpareServers $<$ number $>$ (*default* 5)
This determines the minimum number of Apache processes that must be available at any one time; if processes become busy with client requests, Apache will start up new processes to keep the pool of available servers at the minimum value.

MaxSpareServers $<$ number $>$ (*default* 10)
This determines the maximum number of Apache processes that can be idle at one time; if many processes are started to handle a peak in demand and then the demand tails off, this directive will ensure that excessive numbers of processes will not remain running.

These directives used to be more significant than they are now. Since version 1.3 Apache has a very responsive algorithm for handling incoming requests, starting from 1 to a maximum of 32 new processes each second until all client requests are satisfied. The objective of this is to prevent Apache from starting up excessive numbers of processes all at once unless it is actually necessary because of the performance cost. The server starts with one, then doubles the number of new processes started each second, so only if Apache is genuinely experiencing a sharp rise in demand will it start multiple new processes.

Apache's smart and dynamic handling of the server pool makes it capable of handling large swings in demand. Adjusting these directives has little actual effect on Apache's performance except in extremely busy sites.

Apache has another two directives related to the control of processes:

MaxClients $<$ number $>$ (*default* 256)
Irrespective of how busy Apache gets, it will never create more processes than the limit set by MaxClients, either to maintain the pool of spare servers or to handle requests. Clients that try to connect when all processes are busy will get 'Server Unavailable' error messages.

MaxRequestsPerChild $<$ number $>$ (*default* 0)
This limits the maximum number of requests a given Apache process will handle before voluntarily terminating. The objective of this is to prevent memory leaks causing Apache to consume increasing quantities of memory; while Apache is well behaved in this respect the underlying platform might not be. Setting this to zero means that processes will never terminate themselves, but this has security consequences.

A low value for the MaxRequestsPerChild directive will cause performance problems as Apache will be frequently terminating and restarting processes. A more reasonable value for platforms that have memory leak problems is 1000 or 10000: MaxRequestsPerChild 10000

The Unix version of Apache also runs in a multi-threaded mode, therefore the ThreadsPerChild directive is also significant. This is used for secure resource pooling for threads.

Detailed documentation of Apache implementation can be found in [6] and [1].

2. *Unix Implementation example in Postfix.* Postfix [2] uses this pattern to implement the remote mail recipient. Detailed description of Postfix's use of this pattern is described in [7].

qmail [3] was the first Mail Transfer Agent (MTA) with security as one of the primary requirements. Postfix follows qmail architecture in many places [7, 8], but it has some clever performace hacks that contribute to overall performance improvement. In qmail, processes are forked on demand and their lifetime is limited for the duration of serving the request. Postfix improves this by resource pooling. The size of the resource pool is specified by the system administrator. Postfix does not have sophisticated modules for resource pooling and load balancing like Apache. Instead, the pre-forked processes serve a fixed number of requests and then die. This number is also specified by the system administrator. After the process dies, the parent process forks off a replacement. To limit the vulnerabilities associated with pre-forked processes, the pre-forking parameters have to be set carefully.

The performance improvement in Postfix is evident from benchmark tests in comparison with other MTAs [4, 5].

Related Patterns
The Pooling [11] pattern focuses on the pre-forking and resource pool creation issues. The Resource Lifecycle Manager [11] pattern decouples the management of lifecycle of resources from their use by introducing a separate resource lifecycle manager, whose sole responsibility is to manage and maintain the resources of an application. This can be applied for process monitoring. Another important aspect of the

secure pre-forking pattern is the release of resources. The Evictor [11] pattern describes how and when to release resources to optimize resource management.

The interesting fact about security patterns is that there are analogous mechanisms in human immune systems. The key thing about the secure pre-forking pattern is the killing of processes that are long lived. The Programmatic Cell Death (PCD) mechanism known as Apoptosis [10] is an analogous mechanism where healthy cells are programmatically killed for some benefit. The cells that are killed are not degenerated. Similarly, in secure pre-forking processes that are killed do not have any problem. This is done for security purpose only.

Acknowledgement
Amir Raveh helped me a lot as a shepherd and I thank him for his support. I also thank my PC member Paul Adamczyk for his comments about the paper. The participants in PLoP 2005 writer's workshop came up with a lot of suggestions that has made this paper better. Finally, I thank Professor Ralph Johnson who supervised the research on security patterns.

References

[1] Apache HTTP server project. `http://httpd.apache.org/docs-project/`.

[2] Postfix home page. *Maintained by Wietse Zweitze Venema*. `http://www.postfix.org`.

[3] qmail home page. *Maintained by Daniel Julius Bernstein*. `http://cr.yp.to/qmail.html`.

[4] M. Andree. MTA benchmark. `http://www.dt.e-technik.uni-dortmund.de/~ma/postfix/bench2.html`.

[5] M. Andree. Postfix vs. qmail - performance. `http://www.dt.e-technik.uni-dortmund.de/~ma/postfix/vsqmail.html`.

[6] B. Gröne, A. Knöpfel, R. Kugel, and O. Schmidt. The apache modeling project. `http://apache.hpi.uni-potsdam.de/document/`, May 16 2003.

[7] M. Hafiz. Security architecture of mail transfer agents. Master's thesis, University of Illinois at Urbana-Champaign, 2005.

[8] M. Hafiz. Security patterns and evolution of MTA architecture. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 142–143, New York, NY, USA, 2005. ACM Press.

[9] M. Hafiz, R. Johnson, and R. Afandi. The security architecture of *qmail*. In PLoP 2004 Proceedings, 2004.

[10] J. F. Kerr, A. H. Wyllie, and A. R. Currie. Apoptosis: A basic biological phenomenon with ranging implications in tissue kinetics. *British journal of cancer*, 26(4):239–257, August 1972.

[11] M. Kircher and P. Jain. *Pattern-Oriented Software Architecture, Patterns for Resource Management*. John Wiley and Sons, June 8, 2004.

[12] J. Viega and G. McGraw. *Building Secure Software - How to Avoid Security Problems the Right Way.* Addison-Wesley, 2001.