Extracted from:

Release It! Second Edition

Design and Deploy Production-Ready Software

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Release It!

Second Edition

Design and Deploy
Production-Ready Software

Michael T. Nygard
Edited by Katharine Dvorak

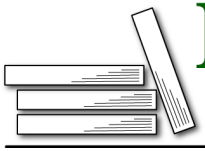
Release It! Second Edition

Design and Deploy Production-Ready Software

Michael T. Nygard

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Managing Editor: Brian MacDonald
Supervising Editor: Jacquelyn Carter
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Molly McBeath
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-239-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—January 2018

The Principle of Least Privilege

The principle of "least privilege" mandates that a process should have the lowest level of privilege needed to accomplish its task. This never includes running as root (UNIX/Linux) or administrator (Windows). Anything application services need to do, they should do as nonadministrative users.

I've seen Windows servers left logged in as administrator for weeks at a time —with remote desktop access—because some ancient piece of vendor software required it. (This particular package also was not able to run as a Windows service, so it was essentially just a Windows desktop application left running for a long time. That is *not* production ready!)

Software that runs as root is automatically a target. Any vulnerability in root-level software automatically becomes a critical issue. Once an attacker has cracked the shell to get root access, the only way to be sure the server is safe is to reformat and reinstall.

To further contain vulnerabilities, each major application should have its own user. The "Apache" user shouldn't have any access to the "Postgres" user, for example.

Opening a socket on a port below 1024 is the only thing that a UNIX application might require root privilege for. Web servers often want to open port 80 by default. But a web server sitting behind a load balancer (see *Load Balancing*, on page ?) can use any port.

Containers and Least Privilege

Containers provide a nice degree of isolation from each other. Instead of creating multiple application-specific users on the host operating system, you can package each application into its own container. Then the host kernel will keep the containerized applications out of each others' filesystems. That's helpful for reducing the containers' level of privilege.

Be careful, though. People often start with a container image that includes most of an operating system. Some containerized applications run a whole init system inside the container, allowing multiple shells and processes. At that point, the container has its own fairly large attack surface. It must be secured. Sadly, patch management tools don't know how to deal with containers right now. As a result, a containerized application may still have operating system vulnerabilities that IT patched days or weeks ago.

The solution is to treat container images as perishable goods. You need an automated build process that creates new images from an upstream base

and your local application code. Ideally this comes from your continuous integration pipeline. Be sure to configure timed builds for any application that isn't still under active development, though.

Configured Passwords

Passwords are the Brazil nut of application security; every mix has them, but nobody wants to deal with them. There's obviously no way that somebody can interactively key in passwords every time an application server starts up. Therefore, database passwords and credentials needed to authenticate to other systems must be configured in persistent files somewhere.

As soon as a password is in a text file, it is vulnerable. Any password that grants access to a database with customer information is worth thousands of dollars to an attacker and could cost the company thousands in bad publicity or extortion. These passwords must be protected with the highest level of security achievable.

At the absolute minimum, passwords to production databases should be kept separate from any other configuration files. They should especially be kept out of the installation directory for the software. (I've seen operations zip up the entire installation folder and ship it back to development for analysis, for example, during a support incident.) Files containing passwords should be made readable only to the owner, which should be the application user. If the application is written in a language that can execute privilege separation, then it's reasonable to have the application read the password files before downgrading its privileges. In that case, the password files can be owned by root.

Password vaulting keeps passwords in encrypted files, which reduces the security problem to that of securing the single encryption key rather than securing multiple text files. This can assist in securing the passwords, but it is not, by itself, a complete solution. Because it's easy to inadvertently change or overwrite file permissions, intrusion detection software such as Tripwire should be employed to monitor permissions on those vital files.[22]

AWS Key Management Service (KMS) is useful here. With KMS, applications use API calls to acquire decryption keys. That way the encrypted data (the database passwords) don't sit in the same storage as the decryption keys! If you use Vault, then it holds the database credentials directly in the vault.

In every case, it's important to expunge the key from memory as soon as possible. If the application keeps the keys or passwords in memory, then

22. www.tripwire.com

memory dumps will also contain them. For UNIX systems, core files are just memory dumps of the application. An attacker that can provoke a core dump can get the passwords. It's best to disable core dumps on production applications. For Windows systems, the "blue screen of death" indicates a kernel error, with an accompanying memory dump. This dump file can be analyzed with Microsoft kernel debugging tools; and depending on the configuration of the server, it can contain a copy of the entire physical memory of the machine—passwords and all.