This document explains the design goals and decisions behind vsftpd.

The importance of a secure design

In a world full of good, careful coders who do not make mistakes, a secure design would not be necessary. After all, in the absence of any programming errors, security would not differ no matter how the program is arranged.

Unfortunately, this is not an ideal world, and coders make plenty of mistakes. Even the careful coders make mistakes. Code auditing is important, and goes some way towards eliminating coding mistakes after the fact. However, we have no guarantee that an audit will catch all the flaws.

So, a secure design acknowledges the possibility of undiscovered flaws, and takes steps to minimise the security impact these flaws can have. An obvious example of something we want to do is to apply the principle of least privilege, which ensure that every part of the program runs with the privilege it needs and no more.

An example of insecure design

Examples of insecure design may be found in most other ftpd's. That's one of the reasons vsftpd has been written. We'll pick on wu-ftpd as a specific example, since it is rumoured to run about half of all ftp services.

If I log on to wu-ftpd as an anonymous user, a process is run on my behalf to serve my ftp session. Unfortunately, this process typically runs with full root privileges on the remote machine. This means that any security flaw present in parsing the copious ftp protocol will lead to full compromise of that machine. Two concrete examples are the recent wu-ftpd format string bug (June 1999), and a buffer overflow dealing with large paths a few months beforehand.

Even OpenBSD's ftpd-BSD had a format string bug leading to remote root compromise of the affected machine, illustrating an earlier point about the requirement for secure design even in the presence of heavy auditing.

Secure design under UNIX

vsftpd is written to run under UNIX-like operating systems, and so its secure

design is constrained by the facilities offered by UNIX. Ideally, UNIX would
have a proper security model which would offer fine grained access control
to all system interactions (files, network, etc). It doesn't, but it does
offer some useful and often overlooked facilities which help us to implement
the principle of least privilege:

- Strong inter-process communication facilities

In UNIX, the process is a strongly defined boundary. Different privilege
credentials may be assigned to different processes, which are not able to
interfere with each other. This is a very basic facility of UNIX.

It makes sense to use this facility to totally separate parts of a program
which do not need to be privileged (most) from those parts that do (typically
minimal).

The privileged and unprivileged parts of the program then communicate via
one of many UNIX IPC mechanisms - perhaps a socketpair or IPC (the former
is attractive because UNIX lets you pass file handles over a socket).

The minimal privileged process exercises the "principle of distrust" - it
carefully filters what the unprivileged process asks it to do, so that even
if the unprivileged process is compromised, it cannot ask the privileged
process to do anything we don't want to allow.

- chroot()

chroot() is an often overlooked but useful tool. It can be used very
effectively as a damage limitation tool.

Imagine a remotely compromised process which does not run as root, but also
does not use chroot(). Now look at what the attacker can do. Amongst the worst
items are pilfering of all publicly readable files, and also attempting to
execute any publicly executable suid-root programs to try and elevate
privilege.

Now imaging the same compromised process with a chroot() to an empty directory.
The attackers options to do unpleasant things are substantially diminished.

No, chroot() is not the ideal way to do what we have just accomplished, but
it is what we have got to work with. In an ideal environment with fine
grained security, we would default to having access to _no_ files at all, and
deliberately not ask for access to any.

- Capabilities (Linux 2.2+)

Like chroot(), capabilities are essentially a damage limitation excercise.
They are also much less widespread than the other UNIX facilities detailled
above. Nonetheless, they warrant mentioning because Linux has them, and they
are used in vsftpd because that is the primary devlopment platform.

Capabilities split up the all powerful root privilege into lots of sometimes
orthogonal privileges. Some of the capabilities represent privileges which
are often the basis for requiring a program to run with full root privileges.
Examples include CAP_NET_RAW (ping, traceroute) and CAP_NET_BIND_SERVICE
(rlogin).

By using capabilities to ensure we only have the privilege we need (within
the somewhat disappointing granularity they offer), we again limit the
potential damage of security holes.

Presenting vsftpd's secure design
==========

vsftpd employs a secure design. The UNIX facilities outlined above are used
to good effect. The design decisions taken are as follows:

1) All parsing and acting on potentially malicious remote network data is
done in a process running as an unprivileged user. Furthermore, this process
runs in a chroot() jail, ensuring only the ftp files area is accessible.

2) Any privileged operations are handled in a privileged parent process. The
code for this privileged parent process is as small as possible for safety.

3) This same privileged parent process receives requests from the unprivileged
child over a socket. All requests are distrusted. Here are example requests:
- Login request. The child sends username and password. Only if the details
are correct does the privileged parent launch a new child with the appropriate
user credentials.
- chown() request. The child may request a recently uploaded file gets
chown'ed() to root for security purposes. The parent is careful to only allow
chown() to root, and only from files owned by the ftp user.
- Get privileged socket request. The ftp protocol says we are supposed to
emit data connections from port 20. This requires privilege. The privileged
parent process creates the privileged socket and passes it to child over
the socket.

4) This same privileged parent process makes use of capabilities and chroot(),

to run with the least privilege required. After login, depending on what
options have been selected, the privileged parent dynamically calculates what
privileges it requires. In some cases, this amounts to no privilege, and the
privileged parent just exits, leaving no part of vsftpd running with
privilege.

5) vsftpd-2.0.0 introduces SSL / TLS support using OpenSSL. ALL OpenSSL
protocol parsing is performed in a chroot() jail, running under an unprivileged
user. This means both pre-authenticated and post-authenticated OpenSSL protocol
parsing; it's actually quite hard to do, but vsftpd manages it in the name of
being secure. I'm unaware of any other FTP server which supports both SSL / TLS
and privilege separatation, and gets this right.

Comments on this document are welcomed.