The actual source code for the qmail system is omitted here; see the qmail website [Bernstein 2008] for examples.

2.1.10 Known Uses

- The qmail mail system [Bernstein 2008].
- The Postfix mail system uses a similar pattern [Postfix].
- Microsoft mentions this general pattern when discussing how to run applications with administrator privileges [MSDN 2009b].
- Distrustful decomposition for Windows Vista applications using user account control (UAC) is explicitly addressed in [Massa 2008].

2.2 PrivSep (Privilege Separation)

2.2.1 Intent

The intent of the PrivSep pattern is to reduce the amount of code that runs with special privilege without affecting or limiting the functionality of the program. The PrivSep pattern is a more specific instance of the Distrustful Decomposition pattern.

2.2.2 Motivation

In many applications, a small set of simple operations require elevated privileges, while a much larger set of complex and security error-prone operations can run in the context of a normal unprivileged user. For a more detailed discussion of the motivation for using this pattern, please see the motivation for the more general Distrustful Decomposition pattern.

Figure 3 provides a detailed view of a system where the PrivSep pattern could be applied and the security problems that can occur if the PrivSep pattern is not used [Provos 2003]. An implementation of `ftpd` is used as an example.

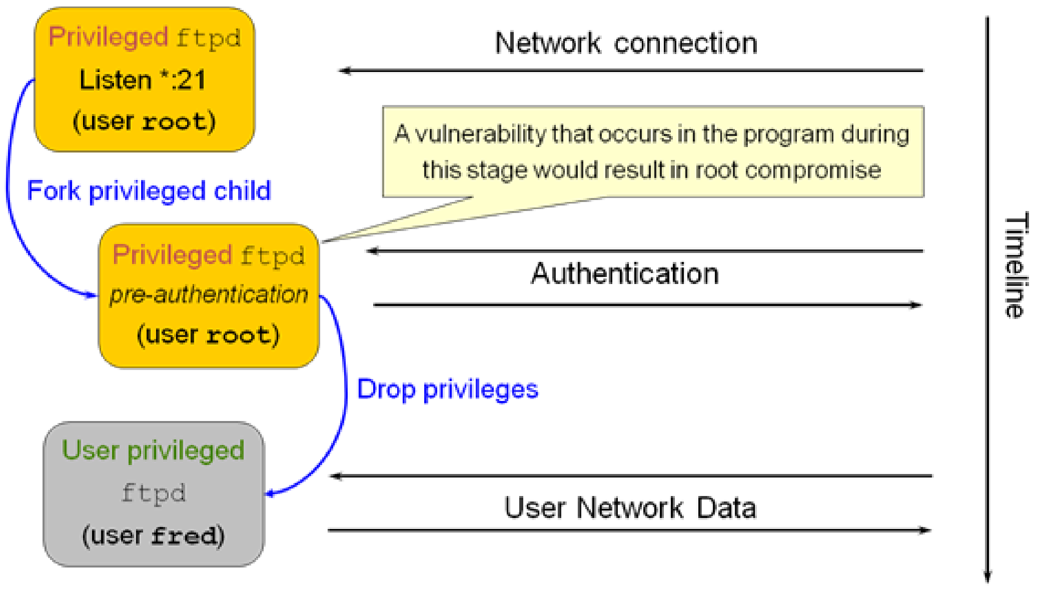

Figure 3: Vulnerable `ftpd` Program

The security flaw occurs when the privileged server establishes a connection with the as-yet untrusted system user and attempts to authenticate the user with a child possessing the same elevated privileges as the server. A malicious user could at this point exploit security holes in the privileged child and gain control of or access to a process with elevated privileges.

2.2.3 Applicability

In general, this pattern is applicable if the system performs a set of functions that

- do *not* require elevated privileges
- have relatively large attack surfaces in that the functions
  - have significant communication with untrusted sources
  - make use of complex, potentially error-prone algorithms

In particular, this pattern is especially useful for system services that must authenticate users and then allow the users to run interactive programs with normal, user-level privileges. It may be also be useful for other authenticating services.

2.2.4 Structure

Figure 4 shows the structure and behavior of the PrivSep pattern. Note that this diagram makes reference to the UNIX `fork()` function for creating child processes. When implementing the PrivSep pattern in a non-UNIX-based OS, a different, OS-specific function would be used in place of `fork()`. For example, under various versions of Windows, the `CreateProcess()` function is used to spawn a child process.

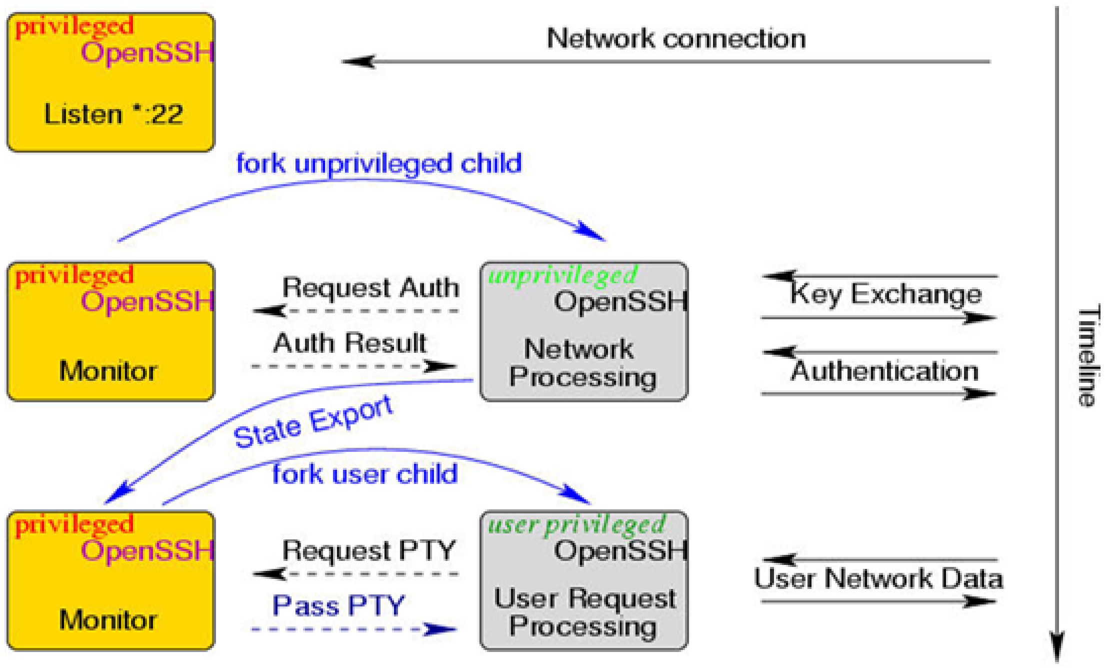

Figure 4: OpenSSH PrivSep Implementation[2]

[2] Source: http://www.citi.umich.edu/u/provos/ssh/priv.jpg.

2.2.5 Participants

Privileged Server Process

The privileged server process is responsible for fielding the initial requests for functionality that will eventually be handled by a child process with non-elevated privileges. The privileged server has an associated privileged userid (often `root` under UNIX-derived OSs, or `administrator` under various versions of Windows).

System User

The system user asks the system to perform some action. This initial request for functionality is directed by the user to the privileged server. The user can be local or remote. The user can communicate with the privileged server via an inter-process communication mechanism such as sockets or SOAP.

Unprivileged Client Process

The unprivileged client is responsible for handling the authentication of the user's request. Because it is not yet known if this is a valid request from a trusted user, the privileges of the child process handling authentication are limited as follows:

- The child process is given the minimal set of privileges allowed by the host OS. Under the UNIX privilege model, this is implemented by setting the user ID (UID) of the process to an unprivileged user ID.
- The root directory of the child process is set to an unimportant, empty directory or a jail [Seacord 2008]. This prevents the untrusted child process from accessing any of the files on the machine running the untrusted child.

User-Privileged Client Process

Once the system user and their request have been authorized, a child process with appropriate user-level privileges is spawned from the privileged server. The user-privileged child process actually handles the system user's request. The user-privileged child has its UID set to a local user ID.

2.2.6 Consequences

- An adversary who gains control over the child
  - is confined in its protection domain and does not gain control over the parent
  - does not gain control of a process possessing elevated privileges, thereby limiting the damage that the adversary can inflict
- Additional verification, such as code reviews, additional testing, and formal verification techniques, can be focused on code that is executed with special privilege, which can further reduce the incidence of unauthorized privilege escalation.
- System administration overhead is usually increased to accommodate the management of new unprivileged user IDs.

2.2.7 Implementation

The PrivSep pattern consists of two phases, pre-authentication and post-authentication.

- **Pre-authentication**. A user has contacted a system service but is not yet authenticated; the unprivileged child has no process privileges and no rights to access the file system.

  The pre-authentication stage is implemented using two entities: a privileged parent process that acts as the monitor and an unprivileged child process that acts as the slave. The privileged parent can be modeled by a finite-state machine (FSM) that monitors the progress of the unprivileged child.
- **Post-authentication**. The user has successfully authenticated to the system. The child has the privileges of the user, including file system access, but does not hold any other special privilege.

The general process implemented in the PrivSep pattern is as follows:

1. Create a privileged server. Initial user requests will be directed to this server.
2. When a user request arrives at the server, the server will spawn off an untrusted, unprivileged child to handle the user interaction required during the authentication process.
3. After the user has been authenticated, the server will spawn off another child process with the appropriate UID to actually handle the user's request.

The unprivileged child is created by changing its UID or group ID (GID) to otherwise unused IDs. This is achieved by first starting a privileged monitor process that forks a slave process. To prevent access to the file system, the untrusted child changes the root of its file system to an empty directory in which no files may be written. The untrusted child process changes its UID or GID to the UID of an unprivileged user so as to lose its process privileges.

Slave requests to the monitor are performed using a standard inter-process communication mechanism.

2.2.8 Sample Code

A simple implementation of the PrivSep pattern using `fork()`, `chroot()`, and `setuid()` under Linux is as follows.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <stdlib.h>

// Define an unused UID.
#define UNPRIVILEGED_UID 123456789

// Hardcode the UID of the user. In reality the UID should not be
// hard coded.
#define USER_UID 1000

// The location of the empty directory to use as the root directory
// for the untrusted child process.
#define EMPTY_ROOT_DIR "/home/sayre/empty_dir"
```

```
/**
 * This defines the behavior for the spawned child, both the one with
 * no privileges and the one with user privileges.
 *
 * The parameters are:
 *
 * childUid - The UID to which to assign the spawned child.
 *
 * sock - The socket the child process will use for communication with
 * the privileged parent.
 */
void handleChild(uid_t childUid, int sock) {

  // A buffer to read in messages from the socket.
  char buffer[100];

  // Change the root of the untrusted child’s file system to an empty
  // directory, if we are the untrusted child.
  if (childUid != USER_UID) {
    if (chroot(EMPTY_ROOT_DIR) != 0) {
      printf("Cannot change root directory to %s.\n", EMPTY_ROOT_DIR);
      exit(7);
    }
  }
  // Immediately set the UID of the child to the user or
  // unprivileged UID.
  if (setuid(childUid) < 0) {
    printf("Cannot set UID to %d\n", childUid);
    exit(6);
  }

  // At this point the child no longer has the full privileges of the
  // privileged parent.

  // Are we the unprivileged child that is used to check
  // authorizations?
  if (childUid != USER_UID) {

    // Yes, we are the unprivileged child.

    // Ask the privileged parent to verify the credentials of the
    // child. Note that for the purposes of this simple example code
    // the "credentials" are represented very simply. In a real
    // application of the PrivSep pattern the credentials would be
    // handled in a much more robust fashion.
    send(sock, "VERIFY: MY_CREDS", 17, 0) ;

    // Read in the credential verification results from the privileged
    // parent.
    int size = recv(sock, buffer, sizeof(buffer), 0) ;

    // Was there an error reading the verification results from the
    // parent?
    if (size < 0) {
      printf("Read error in parent.\n");
      exit(2);
    }

    // Make sure the results string we have been sent is null
    // terminated.
    buffer[sizeof(buffer)-1] = '\0';

    // Were our credentials "authenticated"?
    if (strcmp("yes", buffer) != 0) {

      // Authorization denied. Kill the child process with an
      // appropriate error code.
      printf("Authorization denied.\n");
      exit(5);
    }

    // Our credentials were authorized.
    printf("Authorization approved.\n");

    // The unprivileged child now terminates. The privileged parent
    // will now spawn a child with user privileges.
    exit(0);
  }

  // We are the child with the user's UID. Our authorization has
  // already been approved.
  else {
    // Do the actual work of the verified child here...
    // ...
    // ...
    // ...
  }
}

int main(int argc, const char* argv[]) {

  // Create the socket pair that the parent and child will use to
  // communicate.
  int sockets[2];
  if (socketpair(PF_UNIX, SOCK_STREAM, AF_LOCAL, sockets) != 0) {

    // Creating the socket pair failed. Terminate the process.
    exit(1);
  }

  // A buffer to read in messages from the socket.
  char buffer[100];

  // Initially the spawned child should change its UID to an ID with
  // no privileges.
```

```
uid_t childUid = UNPRIVILEGED_UID;

// Fork into a parent and an unprivileged child process.
pid_t pID = fork();

// Am I the child?
if (pID == 0) {
  // Use an unprivileged child to do the authorization.
  handleChild(childUid, sockets[0]);
}

// Did the fork fail?
else if (pID < 0) {
  printf("Fork failed\n");
  exit(3);
}

// I am the parent.
else {

  // As this point the parent expects the untrusted child to try to
  // get authorized.

  // Get the socket for the parent process.
  int sock = sockets[1];

  // Receive an authorization request from the child.
  int size = recv(sock, buffer, sizeof(buffer), 0) ;

  // Was there an error reading the authorization request message?
  if (size < 0) {
    printf("Read error in parent.\n");
    exit(4);
  }

  // Make sure the string we have been sent is null terminated.
  buffer[sizeof(buffer)-1] = '\0';

  // Do the "authorization" of the child. Note that in this simple
  // example the authorization process has been trivialized. In a
  // real application of the PrivSep pattern a much more robust
  // authorization process would be used.
  if (strcmp("VERIFY: MY_CREDS", buffer) == 0) {

    // Authorization succeeded. Tell the child.
    send(sock, "yes", 4, 0);

    // Because the "authorization" succeeded, spawn off a new child
    // with the user's UID that will do the real work.
    childUid = USER_UID;

    // Fork into a parent and an unprivileged child process.
    pid_t pID = fork();
```

```
      // Am I the child?
      if (pID == 0) {
        handleChild(childUid, sockets[0]);
      }

      // Did the fork fail?
      else if (pID < 0) {
        printf("Fork failed\n");
        exit(3);
      }

      // I am the parent.
      else {

        // Do some other parent operations, if needed...
        // ...
        // ...
        // ...
      }
    }
    else {
      // Authorization failed. Tell the child.
      send(sock, "no", 4, 0) ;
    }
  }
}
```

2.2.9 Known Uses

OpenBSD: `sshd`, `bgpd`/`ospfd`/`ripd`/`rtadvd`, X window server, `snmpd`, `ntpd`, `dhclient`, `tcpdump`, etc.

Secure XML-RPC Server Library

2.3 Defer to Kernel

2.3.1 Intent

The intent of this pattern is to clearly separate functionality that requires elevated privileges from functionality that does not require elevated privileges and to take advantage of existing user verification functionality available at the kernel level. Using existing user verification kernel functionality leverages the kernel's established role in arbitrating security decisions rather than reinventing the means to arbitrate security decisions at the user level.

The Defer to Kernel pattern is a specialization of the following patterns:

- CERT's Distrustful Decomposition secure design pattern
- the Reference Monitor security pattern by Schumacher et al.
  - The Reference Monitor is a general pattern that describes how to define an abstract process that intercepts all requests for resources and checks them for compliance with authorizations [Schumacher 2006].