```
      // 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].

The primary difference between the Defer to Kernel pattern and the Reference Monitor pattern is that the Defer to Kernel pattern focuses on the use of user verification functionality provided by the OS kernel, whereas the Reference Monitor pattern does not specify the authorization method.

2.3.2 Motivation

A primary motivation for this pattern is to reduce or avoid the need for user programs that run with elevated privileges and are consequently susceptible to privilege escalation attacks. In UNIX-based systems, this means the reduction or avoidance of `setuid` programs. Under Windows, this means the avoidance of user programs running as administrator.

In addition, this pattern focuses on the reuse of user verification functionality provided by the OS kernel. The reuse of existing kernel functionality to verify users has these advantages:

- Developers do not have to write their own user identification and verification functionality.
- Testing and validation has already been performed on the existing kernel user identification and verification functionality.
- It is a more portable solution because it allows each OS to verify users in a manner consistent with each platform.

For a more detailed discussion of the motivation for using this pattern, please see the motivation for the more general Distrustful Decomposition pattern.

2.3.3 Applicability

The Defer to Kernel pattern is applicable if the system has the following characteristics:

- The system is run by users who do not have elevated privileges.
- Some (possibly all) of the functionality of the system requires elevated privileges.
- Prior to executing functionality that requires elevated privileges, the system must verify that the current user is allowed to execute the functionality.

In particular, for systems running on UNIX-based operating systems, the Defer to Kernel pattern is applicable if the system has the following characteristics:

- The program must run under a special UID to perform some or all of its tasks.
- The program accepts files or job requests submitted by users.
- For local users, the program needs to know which UID or GID submitted each file or job request, for access control or for accounting.
- For non-local users, the program uses some other user verification and logging mechanism.

2.3.4 Structure

The Defer to Kernel pattern implements a basic client-server architecture. The server runs with elevated privileges, accepts user job requests from clients, and, when possible, uses existing kernel functionality to verify users.

The general structure of the Defer to Kernel pattern is shown in Figure 5.

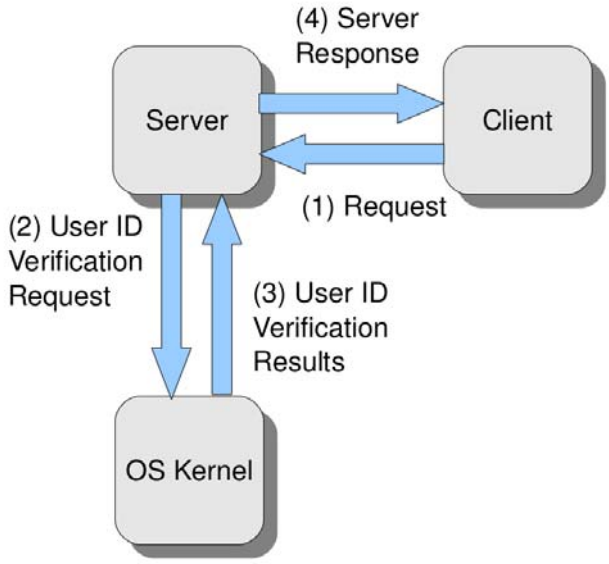

Figure 5: General Structure of the Defer to Kernel Pattern

Figure 6 shows the structure of the Defer to Kernel pattern when the system has the following characteristics:

- The system is implemented under a UNIX-based OS.
- The system uses `getpeereid()` for user verification.
- The server only accepts files and job requests from local users.

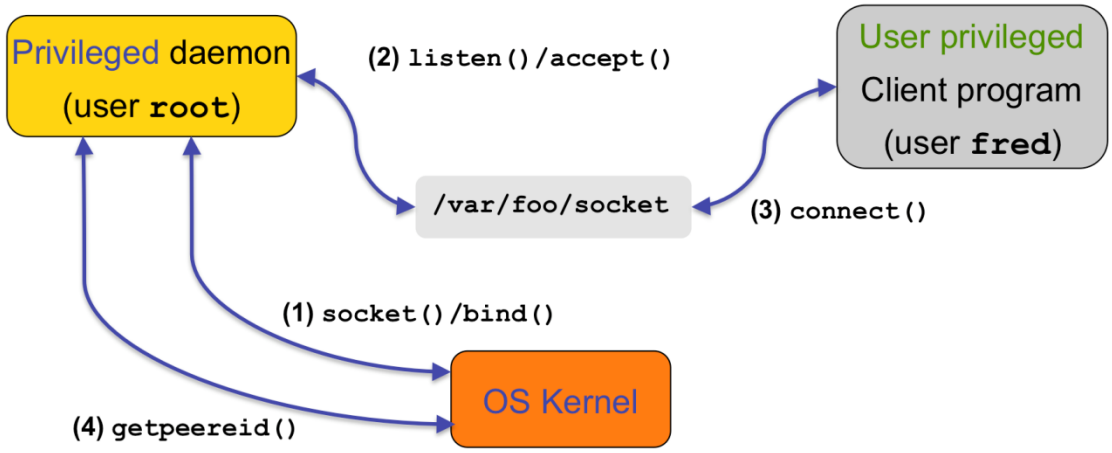

Figure 6: Example Structure of Defer to Kernel Pattern

2.3.5 Participants

These are the participants in the Defer to Kernel pattern:

- **Client program**. The client program runs with standard user-level privileges. It sends job requests to the server to perform work for which the client lacks sufficient privileges.

- **System kernel**. The system kernel provides the following:
  - an inter-process communication mechanism used for communication between the client and the server
  - user identity verification and access functionality. Preexisting functionality implemented in the kernel is used to get the ID of a user connected to the server and to check to see if the user's submitted job request is permitted to run on the server.
- **Server program**. The server program monitors an allocated instance of the IPC mechanism, reads incoming job requests from clients, and checks to see if a client's submitted jobs should be run on the server.

2.3.6 Consequences

- Applications that previously relied on a single executable (`setuid` executable on UNIX-based OSs, executable running as administrator under Windows) must be re-architected as a client/server system.
- Additional system complexity is added because of the added communication between the client and server.

2.3.7 Implementation

The general implementation of the Defer to Kernel pattern is as follows:

1. The server starts up. It accepts client requests via some known mechanism.
2. The client submits a request to the server. Included with the request is information identifying the client. This information is encoded and/or sent using an existing user identification mechanism inherent to the OS's kernel.
3. The server gets the user request and uses some kernel-level mechanism to determine whether to satisfy the user's request or to reject the request.

A more specific implementation suitable for UNIX-based OSs is as follows:

1. The server (cron job, print job, etc.) opens a UNIX domain socket at a known path. All client requests are directed to this UNIX domain socket.
2. The client connects to this socket to submit a request. Because UNIX domain sockets are being used, information about the client user's UID and GID is automatically included with the message. Note that this is performed by the underlying socket code and does not have to be explicitly programmed into the client.
3. The server, upon receiving the `connect()`, invokes a system call such as `getpeereid()` to identify the user making the request.
4. As with the general pattern, the server uses the user identification information from the previous step to determine whether to satisfy the user's request.

Note that this system only addresses local users and not those connecting remotely over a network.

Linux does not include a `getpeereid()` function. However, `getpeereid()` can easily be implemented as follows:

```
/**
```

```
 * Get the user ID and group ID of the user connected to the other end
 * of the given UNIX domain socket.
 *
 * @param sd The UNIX domain socket.
 * @param uid Where to store the user ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for uid must be
 * allocated by the caller.
 * @param gid Where to store the group ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for gid must be
 * allocated by the caller.
 *
 * @returns -1 on failure, 0 on success.
 **/
int getpeereid(int sd, uid_t *uid, gid_t *gid) {
  struct ucred cred;
  int len = sizeof (cred);

  if (getsockopt(sd,SOL_SOCKET,SO_PEERCRED,&cred,&len)) {
    return -1;
  }

  *uid = cred.uid;
  *gid = cred.gid;

  return 0;
}
```

The underlying design of Windows security makes it simple to implement the Defer to Kernel pattern. Under Windows, every process or thread has an associated *access token* containing (among other things) the security identifier (SID) for the user owning the process's account and the SIDs for the user's groups. A server can be set up as a Windows service. A Windows service can be secured by turning it into a *securable object*. When creating a securable object it is possible to associate an *access control list* with the securable object. The access control list contains the SIDs of the client processes that are allowed to connect to the server, that is, the Windows service.

For more information about Windows securable objects, see the online tutorial "Access Control Story: Part I" [Tenouk 2009].

2.3.8 Sample Code

Under Linux, a sketch of the server portion of the Defer to Kernel design pattern is similar to the following:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <linux/un.h>
```

```
#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define QUEUE_SIZE          5
#define SOCKET_PATH     "/tmp/myserver"

/**
 * Get the user ID and group ID of the user connected to the other end
 * of the given UNIX domain socket.
 *
 * @param sd The UNIX domain socket.
 * @param uid Where to store the user ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for uid must be
 * allocated by the caller.
 * @param gid Where to store the group ID of the user connected to the
 * other end of the given UNIX domain socket. Memory for gid must be
 * allocated by the caller.
 *
 * @returns -1 on failure, 0 on success.
 **/
int getpeereid(int sd, uid_t *uid, gid_t *gid) {
  struct ucred cred;
  socklen_t len = sizeof (cred);

  if (getsockopt(sd,SOL_SOCKET,SO_PEERCRED,&cred,&len)) {
    return -1;
  }

  *uid = cred.uid;
  *gid = cred.gid;

  return 0;
}

/* This refers to a user validation function. This will not be
   implemented in this example.

   The purpose of this function is to check to see if the request of
   the connecting user should be read, that is, it checks to see
   if the connecting user is allowed to submit requests (any request)
   to the server. Note that validateUser() makes use of the user ID and
   the group ID of the user, both of which are gathered using the
   kernel-level getpeereid() function.

   The validity of the actual user request will be checked with the
   validateRequest() function.
*/
extern int validateUser(uid_t uid, gid_t gid);

/*
   The purpose of this function is to see if the server should honor
   the request of a connected user. Note that as with validateUser(),
   validateRequest() makes use of the user ID and the group ID of the user
   to check to see if the connected user has the rights to make the server
```

```
   handle the request.

   This will not be implemented in this example.
*/
extern int validateRequest(uid_t uid, gid_t gid, char *request);

int main(int argc, char* argv[]) {
  int hSocket,hServerSocket;  /* handle to socket */
  struct hostent* pHostInfo;   /* holds info about a machine */
  struct sockaddr_un Address; /* Internet socket address struct */
  int nAddressSize=sizeof(struct sockaddr_in);
  char pBuffer[BUFFER_SIZE];

  /* Make a UNIX domain socket for incoming client requests. */
  hServerSocket=socket(AF_UNIX,SOCK_STREAM,0);

  if (hServerSocket == SOCKET_ERROR) {
    puts("\nCould not make a socket\n");
    return 0;
  }

  /* fill in address structure defining how to set up the
     UNIX domain socket. */
  Address.sun_family = AF_UNIX;
  strcpy(Address.sun_path, SOCKET_PATH);
  unlink(Address.sun_path);

  /* Bind the incoming request socket to a "well-known" path. */
  /* In this simple example the "well-known" path is hard-coded. */
  if(bind(hServerSocket,(struct sockaddr*)&Address,sizeof(Address))
     == SOCKET_ERROR) {
    puts("\nCould not connect to host\n");
    return 0;
  }

  /*  get port number */
  getsockname(hServerSocket,
              (struct sockaddr *) &Address,
              (socklen_t *)&nAddressSize);

  /* Establish the listen queue for the incoming request socket. */
  if (listen(hServerSocket,QUEUE_SIZE) == SOCKET_ERROR) {
    puts("\nCould not listen\n");
    return 0;
  }

  /* Get and handle client requests. */
  for(;;) {

    /* Get a user request via an incoming connection. */
    hSocket=accept(hServerSocket,(struct sockaddr*)&Address,
                   (socklen_t *)&nAddressSize);
```

```
/* Figure out who just connected. */
uid_t connectedUID;
gid_t connectedGID;
if (getpeereid(hSocket, &connectedUID, &connectedGID) != 0) {

  /* We cannot figure out who connected. Boot the connection. */
  puts("Cannot figure out who connected. Booting them.");
  if(close(hSocket) == SOCKET_ERROR) {
    puts("ERROR: Could not close socket\n");
    return 0;
  }

  /* Get more incoming connections. */
  continue;
}

/* Validate the user that is going to make a request. */
if (!validateUser(connectedUID, connectedGID)) {

  puts("User not validated. Booting them.");
  if(close(hSocket) == SOCKET_ERROR) {
    puts("ERROR: Could not close socket\n");
    return 0;
  }

  /* Get more incoming connections. */
  continue;
}

/* Get the user's request. */
char *currRequest;
/* .
   . (The request is pointed to by currRequest.)
   . */

/* Validate the connected user’s request. */
if (!validateRequest(connectedUID, connectedGID, currRequest)) {

  puts("User issued invalid request. Booting them.");
  if(close(hSocket) == SOCKET_ERROR) {
    puts("ERROR: Could not close socket\n");
    return 0;
  }

  /* Get more incoming connections. */
  continue;
}


/* Process the user's request. */
/* .
   .
   . */
```

```
    /* Close the socket connected to the current user. */
    if (close(hSocket) == SOCKET_ERROR) {
      puts("ERROR: Could not close socket\n");
      return 0;
    }
  }
```

Under Linux, a sketch of the client portion of the Defer to Kernel design pattern is similar to the following:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <linux/un.h>
#include <stdlib.h>

#define SOCKET_ERROR        -1
#define BUFFER_SIZE         100
#define SOCKET_PATH     "/tmp/myserver"

int  main(int argc, char* argv[]) {
  int hSocket;                 /* handle to socket */
  struct sockaddr_un Address;  /* internet socket address struct */
  char pBuffer[BUFFER_SIZE];
  unsigned nReadAmount;

  /* Make a UNIX domain socket to use to talk with the server. */
  hSocket=socket(AF_UNIX,SOCK_STREAM,0);
  if (hSocket == SOCKET_ERROR) {
    puts("\nCould not make a socket\n");
    return 0;
  }

  /* fill in address structure defining how to set up the
     UNIX domain socket. */
  Address.sun_family=AF_UNIX;
  strcpy(Address.sun_path, SOCKET_PATH);

  /* Connect to host via a "well known" path. */
  /* In this simple example the "well-known" path is hard-coded. */
  if (connect(hSocket,(struct sockaddr*)&Address,sizeof(Address))
      == SOCKET_ERROR) {
      puts("\nCould not connect to host\n");
      return 0;
  }

  /* Communicate request to the server. */
  /* .
     .
```

```
     . */

  /* Close socket used to communicate with the server. */
  if (close(hSocket) == SOCKET_ERROR) {
    puts("\nCould not close socket\n");
    return 0;
  }
}
```

See “Securable Objects” [MSDN 2009a] for information regarding how to implement the Defer to Kernel pattern under Windows.

2.3.9 Known Uses

ucspi-unix

Securable Objects in Windows