- **Windows XP Encrypting File System (EFS)** (http://support.microsoft.com/kb/307877) – The Windows XP EFS is a file or folder level encryption capability supported directly by the Windows XP operating system.
- **TrueCrypt** (http://www.truecrypt.org/) – TrueCrypt is a third-party disk encryption utility for various operating systems. It supports the encryption of full disk volumes.
- **EncFS** (http://www.arg0.net/encfs) – EncFS is a file or directory level encryption level system for Linux.

The Secure XML-RPC Server Library may be configured to use a secure logger.

4.2 Clear Sensitive Information

4.2.1 Intent

It is possible that sensitive information stored in a reusable resource may be accessed by an unauthorized user or adversary if the sensitive information is not cleared before freeing the reusable resource. The use of this pattern ensures that sensitive information is cleared from reusable resources before the resource may be reused.

This secure design pattern is based on the MEM03-CPP "Clear sensitive information stored in reusable resources returned for reuse" CERT Secure Coding recommendation [Seacord 2008].

4.2.2 Motivation (Forces)

Reusable resources include things such as the following:

- dynamically allocated memory
- statically allocated memory
- automatically allocated (stack) memory
- memory caches
- disk
- disk caches

In many cases the action that returns a reusable resource to the pool of resources available for use, such as freeing dynamically allocated memory or deleting a file, simply marks the resource as available for reuse. The current contents of the reusable resource are left intact until the resource is actually reused and new data is written to the resource. This means that an unauthorized user may be able to access the old information left behind in a freed resource, leading to a leak of potentially sensitive information.

4.2.3 Applicability

The Clear Sensitive Information pattern is applicable if the application stores sensitive information in a reusable resource.

4.2.4 Structure

Figure 17 shows the structure and basic behavior of the Clear Sensitive Information pattern.

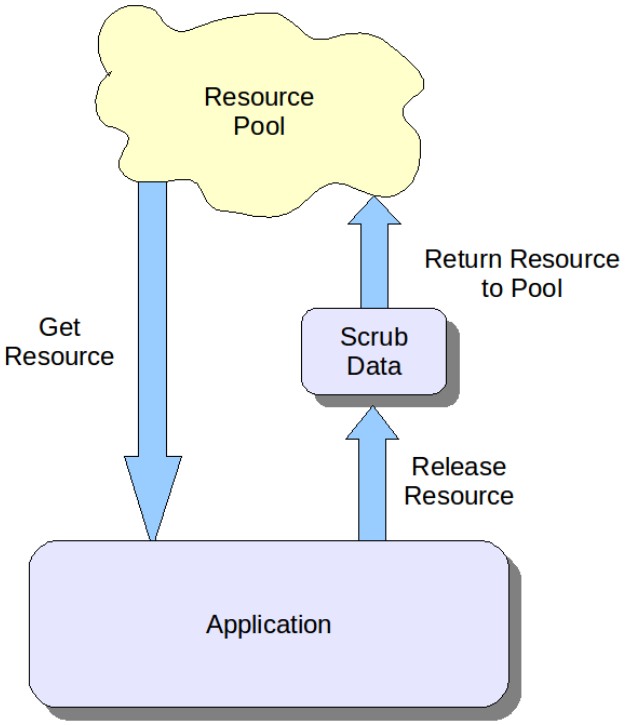

Figure 17: Clear Sensitive Information Pattern Structure

The main focus of this pattern is ensuring that all sensitive data has been cleared prior to returning the reusable resource to the available resource pool.

4.2.5 Participants

- **Resource** – The reusable resource used by the application. As previously mentioned, the resource may be a chunk of dynamically allocated memory, a file on the disk, etc.
- **Application** – The main application writes sensitive information to a resource taken from the reusable resource pool.
- **Resource Pool** – The source of the reusable resources used by the application.
- **Scrub Data** – The application must make use of some mechanism to clear sensitive information from the reusable resource before returning the resource to the resource pool.

4.2.6 Consequences

An unauthorized user who gains access to a reusable resource that has been returned by the application to the resource pool will be unable to read any sensitive information.

4.2.7 Implementation

The implementation of the Clear Sensitive Information secure design pattern proceeds as follows:

1. Identify any sensitive information stored by the application.
2. Analyze the sensitive information stored by the application to identify sensitive information that is stored in reusable resources that will eventually be returned by the application to the resource pool.

3. Identify the points in the application where the reusable resources containing sensitive information are returned to the resource pool.
4. At each point identified in the previous step, ensure that the application clears the sensitive information before returning the reusable resource to the resource pool.

If the application is written in a language that supports the Resource Acquisition is Initialization (RAII) pattern (Section 4.6), the RAII pattern may be used to ensure that sensitive information is always cleared before returning the reusable resource to the resource pool. An example of using RAII to clear sensitive information will be provided in the Sample Code section.

4.2.8 Sample Code

The sample code provided in this section is C++ code showing how to use the RAII pattern to ensure that sensitive information is always cleared before returning a reusable resource to the resource pool. In this example sensitive information is cleared from memory before freeing the memory.

The sample code in this section has been taken from an implementation of a Secure XML-RPC server. The code presented here is used to track information about the known clients of the XML-RPC server. The information stored by the server about a client is

- the IP address of the client
- the level of trust the XML-RPC server has in the client
- the number of faulty XML-RPC requests that have been made by the client

The information about all of the known clients of the XML-RPC server is stored in a client information store. This information may be of potential interest to an attacker of the XML-RPC server, so it is important to clear the information when it is no longer needed.

The definition of the class for objects storing the information about a single client is as follows:

```
/**
 * A ClientInfo object stores information about a single XML-RPC
 * client.
 *
 * The information tracked includes:
 * - The IP address from which the client connects.
 * - The trust level associated with the clients IP address.
 * - The number of faulty requests from the client.
 */
class ClientInfo {

private:

  //! The IP address from which the client connected.
  __be32 ipAddr;

  //! The trust level of the client.
  TrustLevel trustLevel;

  //! The number of faulty requests made by this client;
  unsigned int numFaultyRequests;

public:
```

```
  //! The # of faulty requests before a client is banned.
  static unsigned int maxBadRequests;

  //! The trust level to assign to banned clients.
  static TrustLevel bannedTrustLevel;

  /**
   * Create a new client information object for the given IP address
   * with the given trust level.
   *
   * @param newIpAddr The IP address of the client.
   *
   * @param trust The trust level of the client.
   *
   * The trust level enumerated type is defined in Server.cpp.
   */
  ClientInfo(__be32 newIpAddr, TrustLevel trust);

  /**
   * Copy constructor.
   *
   * @param info The client information object to copy.
   */
  ClientInfo(const ClientInfo &info);

  /**
   * Overwrite all of the fields of the ClientInfo object with zeros
   * prior to destroying the object.
   */
  ~ClientInfo();

  /**
   * Get the # of faulty requests submitted by the client.
   *
   * @return The # of faulty requests submitted by the client.
   */
  unsigned int getNumFaultyRequests() const;

  /**
   * Increment the # of faulty request submissions for the
   * client. The faulty request count will be incremented by 1.
   */
  void trackFaultyRequest();

  /**
   * Get the trust level of the client.
   *
   * @return The trust level of the client.
   */
  TrustLevel getTrustLevel() const;

  /**
   * Get the IP address of the client.
   *
   * @return The IP address of the client.
   */
  __be32 getIpAddr() const;
};
```

The implementation of the class for storing information about a single client is as follows:

```
// ****************************************************
// This value should be set by the server when reading in the server
```

```
// configuration file.
unsigned int ClientInfo::maxBadRequests = 20;

// *********************************************************
// This value should be set by the server when reading in the server
// configuration file.
TrustLevel ClientInfo::bannedTrustLevel = BANNED;

// *********************************************************
// Create a new client information object.
ClientInfo::ClientInfo(__be32 newIpAddr, TrustLevel trust) {
  ipAddr = newIpAddr;
  trustLevel = trust;
  numFaultyRequests = 0;
}

// *********************************************************
// Create a copy of a client information object.
ClientInfo::ClientInfo(const ClientInfo &info) {
  this->ipAddr = info.ipAddr;
  this->trustLevel = info.trustLevel;
  this->numFaultyRequests = info.numFaultyRequests;
}

// *********************************************************
// Clear out the object fields to prevent an attacker from reading
// them.
ClientInfo::~ClientInfo() {
  this->ipAddr = 0;
  this->trustLevel = BOGUS;
  this->numFaultyRequests = 0;
}

// *********************************************************
unsigned int ClientInfo::getNumFaultyRequests() const {
  return numFaultyRequests;
}

// *********************************************************
void ClientInfo::trackFaultyRequest() {
  numFaultyRequests++;

  // Has this client exceeded the maximum number of allowed faulty
  // requests? Also check to see if we are actually dropping clients
  // if they exceed the maximum number of bad requests.
  if ((numFaultyRequests > maxBadRequests) &&
      (maxBadRequests > 0)) {

    // The client is now banned.
    trustLevel = bannedTrustLevel;
  }
}

// *********************************************************
TrustLevel ClientInfo::getTrustLevel() const {
  return trustLevel;
}

// *********************************************************
__be32 ClientInfo::getIpAddr() const {
  return ipAddr;
}
```

Note that the destructor of the ClientInfo class overwrites all of the fields of the object with non-sense values as part of the process of destroying the object. This ensures that an attacker will not be able to read the memory previously allocated for a ClientInfo object to find out information about a client of the XML-RPC server.

The XML-RPC server stores information about all known clients in a central client information store. The definition of the ClientInfoStore class is as follows:

```
/**
 * The ClientInfoStore will maintain a map between IP addresses and
 * a pointer to the corresponding ClientInfo object. If a ClientInfo
 * object for an unknown IP address is requested, a new ClientInfo
 * object for that IP address will be created, its pointer added to
 * the map, and returned.
 *
 * The ClientInfo class will use the RAII pattern in conjunction
 * with the Clear Sensitive Information pattern to automatically
 * clear out the client information in the map when the
 * ClientInfoStore object is destroyed.
 */
class ClientInfoStore {

private:

  //! The default trust level to use for clients w. unknown IP addresses.
  TrustLevel defaultTrustLevel;

  //! A map from client IP addresses to their information.
  map<__be32, ClientInfo *> clientInfoMap;

  /**
   * Free all the ClientInfo entries, scrub their memory, and empty
   * the map. Also overwrite the default trust level with a garbage
   * value.
   */
  void clear();

public:

  /**
   * Create a client information store with an initial list of
   * client information. The IP address is included as part of the
   * ClientInfo object. This will make copies of the ClientInfo
   * objects in the list. The default trust level for clients with
   * unknown IP addresses is given.
   *
   * @param defaultTrust The default trust level to assign to
   * clients with unknown IP addresses.
   *
   * @param initialInfo The list of clients that we already know
   * about.
   *
   * @throw XmlRpcException An exception will be thrown if
   * allocation of memory for a client information object fails.
   */
  ClientInfoStore(TrustLevel defaultTrust, list<ClientInfo> initialInfo)
    throw (XmlRpcException);

  /**
   * Create an empty ClientInfoStore.  The default trust level for
   * clients with unknown IP addresses is given.
```

```
   *
   * @param defaultTrust The default trust level to assign to
   * clients with unknown IP addresses.
   */
  explicit ClientInfoStore(TrustLevel defaultTrust);

  /**
   * Clear the sensitive client information before destroying the
   * client information store.
   */
  ~ClientInfoStore();

  /**
   * Get the ClientInfo object for the requested IP address,
   * creating a new one if needed with the default trust level. The
   * caller should never free the returned ClientInfo object.
   *
   * @param ipAddr The IP address of the desired client;
   *
   * @return The client information of the desired client.
   *
   * @throw XmlRpcException An exception will be thrown if
   * allocation of memory for a client information object fails.
   */
  ClientInfo *getClientInfo(__be32 ipAddr) throw (XmlRpcException);
};
```

The implementation of the ClientInfoStore class is as follows:

```
// ****************************************************
// Create a new store with some existing client information.
ClientInfoStore::ClientInfoStore(TrustLevel defaultTrust,
                                 list<ClientInfo> initialInfo)
  throw (XmlRpcException) {

  // Store the default trust level.
  defaultTrustLevel = defaultTrust;

  // Create the map mapping client IP addresses to client info.
  clientInfoMap = map<__be32, ClientInfo *>();

  try {

    // Add the known client information to the map.
    list<ClientInfo>::iterator i;
    for (i=initialInfo.begin(); i != initialInfo.end(); ++i) {
      clientInfoMap[i->getIpAddr()] = new ClientInfo(*i);
    }
  }

  // Memory allocation failed.
  catch (bad_alloc& ba) {
    throw XmlRpcException(string("") +
                          "Allocating memory for a new ClientInfo " +
                          "object failed.");
  }
}

// ****************************************************
// Create a new empty store.
ClientInfoStore::ClientInfoStore(TrustLevel defaultTrust) {

  // Store the default trust level.
```

```
  defaultTrustLevel = defaultTrust;

  // Create the map mapping client IP addresses to client info.
  clientInfoMap = map<__be32, ClientInfo *>();
}

// ****************************************************
// Clear out the store.
void ClientInfoStore::clear() {

  // Free all the client info objects.
  for(map<__be32, ClientInfo *>::iterator ii=clientInfoMap.begin();
      ii != clientInfoMap.end();
      ++ii) {
    delete (*ii).second;
  }

  // Clear the map.
  clientInfoMap.clear();

  // Set the default trust level to a nonsense value.
  defaultTrustLevel = BOGUS;
}

// ****************************************************
ClientInfoStore::~ClientInfoStore() {
  clear();
}

// ****************************************************
// Get the client info for a client.
ClientInfo *ClientInfoStore::getClientInfo(__be32 ipAddr)
  throw (XmlRpcException) {

  try {

    // Is this a client we know about?
    if (clientInfoMap.find(ipAddr) == clientInfoMap.end()) {

      // No, we don't know about it. Track an empty client info object
      // for this IP address with the default trust level for the
      // client.
      ClientInfo *newClient = new ClientInfo(ipAddr, defaultTrustLevel);

      // Add the new client to the store.
      clientInfoMap[ipAddr] = newClient;
    }
  }

  // Memory allocation failed.
  catch (bad_alloc& ba) {
    throw XmlRpcException(string("") +
                          "Allocating memory for a new ClientInfo " +
                          "object failed.");
  }

  // Return the desired client information.
  return clientInfoMap[ipAddr];
}
```

Note that the destructor of a ClientInfoStore deletes all of the ClientInfo objects tracked by the store, which results in the fields of the ClientInfo objects being overwritten with nonsense values

(see the previously defined destructor of the ClientInfo class for more information). The default trust level assigned by the XML-RPC server to unknown clients is also overwritten with a non-sense value.

4.2.9 Known Uses

Secure XML-RPC Server Library

Other uses of this pattern probably exist. Find them.

4.3 Secure Directory

4.3.1 Intent

The intent of the Secure Directory pattern is to ensure that an attacker cannot manipulate the files used by a program during the execution of the program. See "FIO15-C. Ensure that file operations are performed in a secure directory" in *The CERT C Secure Coding Standard* [Seacord 2008] for additional information regarding this issue.

4.3.2 Motivation

A program may depend on a file for some length of time during program execution. The program developers usually assume that the files used by the program will not be manipulated by outside users during the execution of the program. However, if this assumption is false, a file may be modified by multiple users, which means that a malicious user may modify or delete the file during a critical time when the program relies on the file remaining unmodified, causing a race condition in the program.

The Secure Directory pattern ensures that the directories in which the files used by the program are stored can only be written (and possibly read) by the user of the program.

4.3.3 Applicability

The Secure Directory pattern is applicable for use in a program if

- The program will be run in an insecure environment; that is, an environment where malicious users could gain access to the file system used by the program.
- The program reads and/or writes files.
- Program execution could be negatively affected if the files read or written by the program were modified by an outside user while the program was running.

4.3.4 Structure

Programmatically, the structure of the Secure Directory pattern is fairly simple. Prior to opening a file for reading or writing, the Secure Directory pattern states that the program must

1. find the canonical pathname of the directory of the file (see Section 4.4, "Pathname Canonicalization")
2. check to see if the directory, as referenced by the canonical pathname, is secure

The structure of the secure directory is such that the directory has write permissions limited to the user and the superuser. No other users may modify files in the secure directory. Furthermore, all