4 The Implementation-Level Patterns

4.1 Secure Logger

4.1.1 Intent

The intent of the Secure Logger pattern is to prevent an attacker from gathering potentially useful information about the system from system logs and to prevent an attacker from hiding their actions by editing system logs.

4.1.2 Motivation (Forces)

System logs usually contain a great deal of information that is useful to the people administering a system or debugging a new system. In addition, the information contained in a system log may contain information that could be used by an attacker to devise new vectors for attacking the system.

An attacker may also edit the logging information to hide their attacks.

4.1.3 Applicability

The Secure Logger pattern is applicable if

- The system logs information to a log file or some other form of logging subsystem.
- The information contained in the system log could be used by an attacker to devise attacks on the system.
- System logs are used to detect and diagnose attacks on the system.

4.1.4 Structure

Figure 16 shows the structure and basic behavior of the Secure Logger pattern.

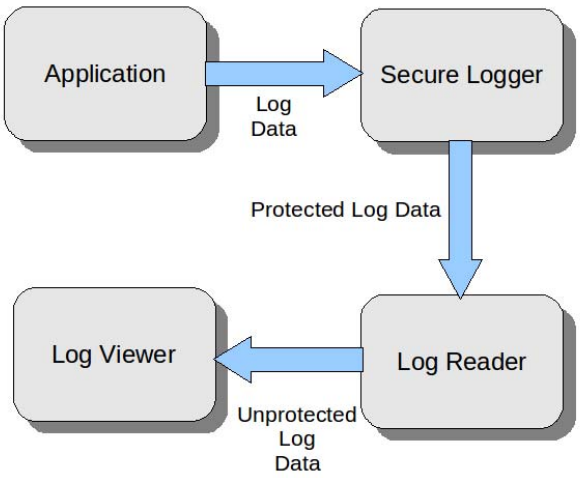

Figure 16: Secure Logger Pattern Structure

4.1.5 Participants

- **Application** – The main application generates logging data which is handled and stored by some form of secure logging system.
- **Secure Logger** – The secure logging system is responsible for storing the logging information in a manner that makes it difficult or impossible for an unauthorized user to access the logging data.
- **Log Reader** – Because the secure logging system stores the logging data in a protected manner, a reading mechanism specific to the secure logging system will need to be used to read the log contents. Standard mechanisms for reading log files, such as reading the file in a standard text editor or word processing application will not work for log data stored by the secure logging system. Note that it may not be necessary to actually have a separate application for reading the protected log data. The secure logging system itself may provide a mechanism for allowing authorized users to read the log data.
- **Log Viewer** – An authorized user may read the log data using some sort of log-viewing application.

4.1.6 Consequences

- An adversary who gains access to the log data managed by the secure logging subsystem will have limited or no visibility of the actual log data content. The adversary will be unable to use information in the log data to plan more sophisticated attacks on the system.
- Modifications to the log data by an attacker will be detectable by an authorized user.

4.1.7 Implementation

The implementation of the Secure Logger pattern is fairly straightforward.

1. Select a secure logging subsystem to use. Several potential secure logging subsystems are mentioned in the Known Uses section.
2. In the system logging the messages, always use the chosen secure logging subsystem to log data. Note that this implies that a delivered version of the system should never output messages to standard out or standard error.

To make the particular chosen secure logging subsystem transparent to the system generating logging data, it is possible to define an abstract interface that hides the details of interacting with the chosen secure logging subsystem. In addition, it is possible to use the Abstract Factory pattern to generate logging objects, which makes it relatively easy to make use of different logging subsystems as needed. For example, during internal development and testing of the system using a simple, insecure logging subsystem may be appropriate. Using an Abstract Factory to generate system loggers will make it relatively simple to switch over to using a secure logger in an official release of the system.

The use of abstract interfaces to hide the details of interacting with the logging subsystem and the use of the Abstract Factory pattern to generate loggers will be explored in more detail in the Sample Code section.

4.1.8 Sample Code

The sample code provided in this section is C++ code showing how to use an abstract interface to hide the details of interacting with the logging subsystem and how to use the Abstract Factory pattern to generate logger instances. A very simple, relatively insecure logger will be used for the purposes of this example. A real-world implementation of this pattern should probably make use of an existing secure logging facility.

The sample code in this section has been taken from an implementation of a Secure XML-RPC server.

A simple logger interface is as follows:

```
/**
 * The SecureLogger interface defines the methods that any concrete
 * secure logging class must implement.
 */
class SecureLogger {

public:

  /**
   * Log the given message to the secure log.
   *
   * @param msg The message to log.
   */
  virtual void log(string msg) = 0;
};
```

This interface states that a logger object must be able to log a text message to the logging subsystem. No other information about the specific logging subsystem used is needed.

A somewhat secure implementation of the SecureLogger interface could be a logger that encrypts each individual log message prior to writing the log message to a central log file. Note that for illustrative purposes this logger uses a very simple, insecure XOR-based encryption algorithm.

The class definition for the encrypted logger class is as follows.

```
/**
 * This logger will encrypt each log message before writing it to a
 * log file. It is somewhat secure.
 */
class EncryptLogger : public SecureLogger {

 private:

  //! The log file.
  ofstream logFile;

public:

  /**
   * Create a new encrypted logger that will write encrypted log
   * messages to the given file.
   *
   * @param logFileName The name of the file to which to log
   * messages.
   *
```

```
   * @throws XmlRpcException An exception will be thrown if the log
   * file cannot be opened for writing.
   */
  explicit EncryptLogger(const string logFileName);

  //! Close the log file when destroying the logger.
  ~EncryptLogger();

  void log(const string msg);
};
```

The implementation of the simple encrypted logger class is as follows.

```
// *****************************************************
/**
 * This encrypts a string using a very simple XOR based
 * encryption algorithm.
 *
 * @param inputString The string to encrypt.
 *
 * @return The encrypted version of the input string.
 *
 * @attention <b>This is not a good encryption method. Replace this
 * if needed.</b>
 */
string encrypt(const string &inputStr) {

  unsigned int x;
  string r = "";
  for (x = 0; x < inputStr.size() ; x++) {
    char newChar =
      ((unsigned short) inputStr[x]) ^ ((unsigned short) x);
    if (newChar == '\n') {
      r += "<NEWLINE_CHAR>";
    }
    else {
      r += string(1, newChar);
    }
  }

  return r;
}

// *****************************************************
EncryptLogger::EncryptLogger(const string logFileName) {

  // Open the log file.
  logFile.open(logFileName.c_str(), ios::out | ios::trunc | ios::binary);
  if (!logFile.is_open()) {

    // Opening the log file failed. Fail.
    throw XmlRpcException("Cannot open log file " + logFileName +
                          " for writing.");
  }
}

// *****************************************************
EncryptLogger::~EncryptLogger() {
  logFile.close();
}

// *****************************************************
void EncryptLogger::log(const string msg) {
```

```
  // Output an unencrypted time stamp into the log file and then the
  // encrypted log message.
  logFile << getTime() << "\n"
          << encrypt(msg) << "\n";
  logFile.flush();
}
```

To facilitate switching loggers in a system, the Abstract Factory pattern may be used to gain access to the desired logger generation factory. The secure logger factory will provide the appropriate concrete implementation of a logger generation factory, which may then be used to generate a logger.

An abstract secure logger factory may be implemented as follows:

```
/**
 * This interface defines a factory for classes that provide an
 * instance of a secure logger to the caller. The
 * SecureLoggerFactory class implements the Abstract Factory
 * pattern.
 */
class SecureLoggerFactory {

private:

  //! The current default concrete logger factory.
  static SecureLoggerFactory *instance;

public:

  /**
   * Get the logger.
   *
   * @return A pointer to a secure logger object.
   *
   * @throw XmlRpcException An exception will be thrown if
   * allocation of memory for the logger fails.
   */
  virtual SecureLogger *getLogger() throw (XmlRpcException) = 0;

  /**
   * Get the current default concrete logger factory.
   *
   * @return The current active logger factory.
   */
  static SecureLoggerFactory *getInstance();

  /**
   * Set the current default concrete logger factory. Use this to
   * use a logger factory other than the default logger factory
   * (SimpleLoggerFactory).
   *
   * @param newFactory The new logger factory instance to use.
   */
  static void setInstance(SecureLoggerFactory *newFactory);
};
```

The implementation of the abstract secure logger factory is as follows:

```
// ****************************************************
// We will initially use the simple encrypted logger.
```

```
SecureLoggerFactory *SecureLoggerFactory::instance = new EncryptedLoggerFac-
tory();

// ****************************************************
SecureLoggerFactory *SecureLoggerFactory::getInstance() {
  return instance;
}

// ****************************************************
void SecureLoggerFactory::setInstance(SecureLoggerFactory *newFactory) {
  instance = newFactory;
}
```

Finally, a logger factory that returns instances of the previously defined simple encrypted logger may be specified as follows:

```
/**
 * The encrypt logger factory returns an instance of a somewhat
 * secure logger when asked for a logger.
 */
class EncryptLoggerFactory : public SecureLoggerFactory {

private:

  //! The encrypt logger always returned by getLogger().
  EncryptLogger *theLogger;

public:

  //! The name of the log file for encrypted loggers.
  static string logFileName;

  EncryptLoggerFactory() {theLogger = NULL;};
  SecureLogger *getLogger() throw (XmlRpcException);
};
```

The implementation of the methods in the simple encrypted logger factory is as follows:

```
// ****************************************************
// This value should be set at application start time.
string EncryptLoggerFactory::logFileName = "xml_rpc_server.log";

// ****************************************************
SecureLogger *EncryptLoggerFactory::getLogger() throw (XmlRpcException) {

  // Do we need to create a new logger?
  if (theLogger == NULL) {

    try {

      // Yes, create a new encrypt logger.
      theLogger = new EncryptLogger(logFileName);
    }

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

```
  return theLogger;
}
```

As noted in the Participants section, depending on the secure logger used it may be necessary to use a separate application for reading the log data handled by the secure logger. In the case of the simple encrypted logger described above, a utility is needed to decrypt the log entries. A command line utility for decrypting and displaying the log files generated by the simple encrypted logger may be implemented in Python as follows:

```
#! /usr/bin/env python

import sys

def decrypt(s):
    currPos = 0
    r = ""
    for c in s:
        r += chr(ord(c) ^ currPos)
        currPos += 1
    return r

if (len(sys.argv) != 2):
   print "USAGE: decrypt_log.py ENCRYPTED_LOG_FILE"
   sys.exit(1)

logFileName = sys.argv[1]
inFile = open(logFileName, 'r')
decryptLine = False
for line in inFile.readlines():
    line = line[:-1]
    if (decryptLine):
        line = line.replace("<NEWLINE_CHAR>", "\n");
        sys.stdout.write(decrypt(line) + "\n")
    else:
        sys.stdout.write(line + ": ")
    decryptLine = not decryptLine

inFile.close()
```

4.1.9 Known Uses

Several secure logging facilities exist:

- **syslog-ng** (http://www.balabit.com/network-security/syslog-ng/) – The syslog-ng logging application provides a secure centralized logging system that may be used by multiple applications running on a Linux server.
- **SmartInspect** (http://www.gurock.com/smartinspect/) – The SmartInspect logging library supports AES encrypted log files for Java, Delphi, and .NET.
- **CLogIt** (http://www.codeproject.com/KB/files/logit.aspx?msg=686567) – The CLogIt application is a small library, with source code, that implements the reading and writing of encrypted log files. It is written in C++ for Windows.

It is also possible to use general file system encryption utilities to support secure logging. In this case the log files would be written to and read from an encrypted file or disk volume.

- **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.