```
  /* Handle error */
}

if (remove(file_name) != 0) {
  /* Handle error */
}
```

4.4 Pathname Canonicalization

4.4.1 Intent

The intent of the Pathname Canonicalization pattern is to ensure that all files read or written by a program are referred to by a valid path that does not contain any symbolic links or shortcuts, that is, a canonical path.

4.4.2 Motivation

Because of symbolic links and other file system features, a file may not actually reside in the directory indicated by a path. Therefore, performing string-based validation on the pathname may yield false results. Having the true, canonical pathname is particularly important when checking a directory to see if it is secure.

4.4.3 Applicability

The use of the Pathname Canonicalization pattern is applicable if all of the following conditions are true:

- the program accepts pathnames from untrusted sources
- an attacker could provide a pathname to the system that non-obviously refers to a directory or file to which the attacker should not have access
- the program runs in an environment where each file has a unique canonical pathname

4.4.4 Structure

Programmatically, the structure of the Pathname Canonicalization pattern involves calling an OS-specific pathname canonicalization function on the given pathname prior to opening the file. The canonicalized pathname is used when operating on the file.

The canonicalized pathname itself has a structure such that every element of the canonicalized path, except the last, is the genuine directory, and not a link or shortcut. The last element is the genuine filename, and not a link or shortcut.

4.4.5 Participants

The participants in the Pathname Canonicalization pattern are

- the program opening file(s)
- the file system (potentially, depending on the implementation of the OS-specific canonicalization function)

4.4.6 Consequences

- Pathname canonicalization guarantees that textual analysis of the canonicalized pathname yields accurate results, which improves the accuracy and security of file access.
- The program speed is degraded due to the canonicalization of pathnames. To reduce the overhead of canonicalization, it is possible to cache the canonicalized pathname. Note that such caching assumes that the directory structure accessed by the program is not changed during program execution.

4.4.7 Implementation

The core of the implementation of this pattern is an OS-specific function for performing pathname canonicalization. The canonicalization function is a routine that would ensure that every directory in a pathname is a genuine directory rather than a link or shortcut. The result of the canonicalization function is a canonicalized path such that string-based validation of the path always yields valid results. For instance, a canonicalized path that begins with the pathname to a user's home directory will guarantee that the path's file lives in the user's home directory or a subdirectory below the user's home directory.

As discussed in the Structure section, given the canonicalization function, the implementation of the Pathname Canonicalization pattern is fairly simple:

1. The program calls the OS-specific pathname canonicalization function on the given pathname prior to opening a file.
2. The canonicalized pathname is used when operating on the file.

Canonicalization routines should be provided by the platform; a program should simply call the platform's canonicalization routine before performing textual analysis on a pathname. Some OS-specific canonicalization functions are

- POSIX-compliant OSs: `realpath()`
- systems with `glibc`: `canonicalize_file_name()`, a GNU extension provided in `glibc`

See FIO02-C, "Canonicalize path names originating from untrusted sources for implementation details" in *The CERT C Secure Coding Standard* [Seacord 2008].

4.4.8 Sample Code

The following sample code canonicalizes a user-supplied pathname before verifying and opening the file.

```
/* Verify argv[1] is supplied */

char *canonical_filename = canonicalize_file_name(argv[1]);
if (canonical_filename == NULL) {
  /* Handle error */
}

/* Verify filename */

if (fopen(canonical_filename, "w") == NULL) {
  /* Handle error */
}
```

```
/* ... */

free(canonical_filename);
canonical_filename = NULL;
```

4.4.9 Known Uses

xarchive-0.2.8-6

Secure XML-RPC Server Library

4.5 Input Validation

4.5.1 Intent

Many vulnerabilities can be prevented by ensuring that input data is properly validated. Input validation requires that a developer correctly identify and validate all external inputs from untrusted data sources.

4.5.2 Motivation

The use of unvalidated user input by an application is the root cause of many serious security exploits, such as buffer overflow attacks, SQL injection attacks, and cross-site scripting attacks.

Given the prevalence of applications with a client-server architecture, one issue faced by system designers is where to perform the input validation, on the client side or on the server side. Problems in input validation occur when only client-side validation is performed.

Client-side validations are inherently insecure. It is easy to spoof a web page submission and bypass any scripting on the original page. This is more or less true for any type of client-server architecture. However, while you cannot rely on client-side validation, it is still useful. Immediate user feedback can avoid another round trip to the server, saving time and bandwidth.

4.5.3 Example

A university is writing an ERP (Enterprise Resource Planning) application with a web-based interface to allow university employees to enter time sheet information, bill purchases against accounts, and track the status of various funding sources. The university wishes to ensure (among other security considerations) that malicious or incorrect user input does not result in forbidden changes to ERP data, violations of data integrity, or forbidden access to data by a user.

4.5.4 Applicability

This pattern is applicable to any software that accepts data from an untrusted source. Any data that arrives at a program interface across a security boundary requires validation. General examples of such data include `argv`, environment, sockets, pipes, files, signals, shared memory, and devices. Some input sources specific to web applications are `GET` and `POST` parameters from HTTP forms. Other applications may have other input sources.

4.5.5 Structure

The structure of the Input Validation pattern is fairly simple and only requires identifying and validating each untrusted input as shown in Figure 18.

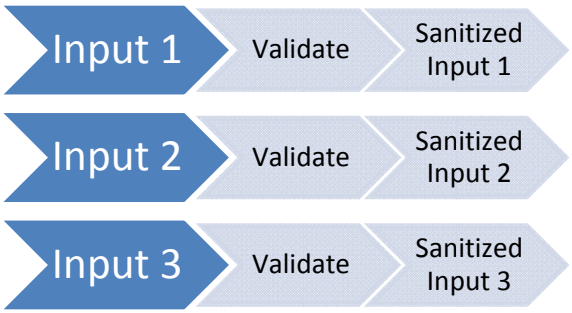

Figure 18: Structure of the Input Validation Pattern

4.5.6 Participants

These are the participants in the Input Validation pattern:

- The system accepting data. The primary participant in this pattern is the system that accepts and validates data.
- External entities providing data. The data provided to the system comes from some external source. Potential data sources include
  - human users
  - files
  - network connections
  - memory shared with other processes
  - database systems

4.5.7 Consequences

The benefits of validating all system input is increased system security (exploits that rely on poor handling of invalid input are prevented) and reliability (the system behaves in a predictable manner when provided with invalid input). The costs of input validation are slower system performance and the additional work required to identify and handle all places where invalid input can occur.

4.5.8 Implementation

The implementation of the Input Validation secure design pattern involves two general design tasks:

- **Specify and validate data.** Data from all untrusted sources must be fully specified and the data validated against these specifications. The system implementation must be designed to handle any range or combination of valid data. Valid data, in this sense, is data that is anticipated by the design and implementation of the system and therefore will not result in the system entering an indeterminate state. For example, if a system accepts two integers as input and multiplies those two values, the system must either (a) validate the input to ensure

that an overflow or other exceptional condition cannot occur as a result of the operation or (b) be prepared to handle the result of the operation in the event of an overflow or other exceptional condition. The specifications must address limits, minimum and maximum values, minimum and maximum lengths, valid content, initialization and re-initialization requirements, and encryption requirements for storage and transmission.

- **Ensure that all input meets the specification.** Use data encapsulation (e.g., classes) to define and encapsulate input. For example, instead of checking each character in a user name input to make sure it is a valid character, define a class that encapsulates all operations on that type of input. Input should be validated as soon as possible. Incorrect input is not always malicious; often it is accidental. Reporting the error as soon as possible often helps correct the problem. When an exception occurs deep in the code it is not always apparent that the cause was an invalid input and which input was out of bounds.

A data dictionary or similar mechanism can be used for specification of all program inputs. Input is usually stored in variables, and some input is eventually stored as persistent data. To validate input, specifications for what is valid input must be developed. A good practice is to define data and variable specifications, not just for all variables that hold user input, but also for all variables that hold data from a persistent store. The need to validate user input is obvious; the need to validate data being read from a persistent store is a defense against the possibility that the persistent store has been tampered with.

General Implementation Process

In more detail, the process for implementing this pattern consists of the following steps:

1. **Identify all input sources.** All sources of input to the system must be identified. An input source is any entity or resource that provides data to the system where the received data is non-deterministic; that is, any source of data where the value of the data is not completely determined by the current internal state of the system and past actions performed by the system. As mentioned previously, potential input sources are the file system, a database system, network traffic read via a socket, input from a pipe, the keyboard, etc.
2. **Identify all reads of input sources.** For each input source, identify every point in the system where data from the input source is initially read. Note that if the system has been designed to be loosely coupled from the input sources and hence has interaction with the input sources isolated to a small number of places in the code base, the identification of reads from input sources will be relatively simple. However, if the system was designed so that interaction with data sources is scattered throughout the code base, identification of all reads from input sources will be difficult.
3. **Define criteria for valid data.** For each of the data reads identified in the previous step, define what it means for data read by the current read to be valid. The definition of validity will depend on the type of data being read and what that particular data will be used for. For example:
    a. **Numeric data**. Numeric data should be checked to make sure that it is within some fixed bounds. It should also be checked to ensure it does not cause overflow or underflow errors in subsequent computations. Additional guidance on the checking of numer-

ic data can be found in the CERT C Secure Coding rules and recommendations [Seacord 2008].

b. **String data**. If the string data is going to be displayed on a web page, it should be sanitized to ensure that it does not contain HTML and client-side script code. If the string data is going to be used in a database query, it should be sanitized to foil SQL injection attacks.

4. **Figure out how to handle invalid data.** For each of the data reads identified in step two, the behavior of the system when given invalid data should be explicitly defined. Responses to invalid input can range from issuing a warning and continuing with default data to re-requesting the data from the input source. Correct handling of invalid data is a highly application-specific matter.

5. **Add code to check for and handle invalid data.** For each of the data reads identified in step two, code should be written to check the validity of the data read and cases of invalid data should be handled.

   There are two common approaches to identifying invalid data: *blacklisting* and *whitelisting*. Blacklisting consists of comparing input data against a set of inputs known to be invalid, commonly known as a blacklist. If it is not on the blacklist, the input may be considered valid. Whitelisting consists of comparing input data against a set of inputs known to be valid, commonly known as a whitelist. If it is not on the whitelist, the input may be considered invalid. Both whitelisting and blacklisting involve a simple implementation, comparing input against the whitelist or blacklist. The main work comes in maintaining the whitelist or blacklist. When either solution is possible, the whitelist is considered a safer choice because new forms of invalid input need to be entered into a blacklist, but a whitelist requires no change upon discovery of new forms of invalid input.

Additional Implementation Information

Some specific ways to implement input validation in a structured method are available in these sources:

- "Input Validation Using the Strategy Pattern" [Gervasio 2007]. This solution uses the Gang of Four Strategy pattern [Gamma 1995] to handle input validation for various classes of inputs. The presented solution is programmed in PHP.
- "Client/Server Input Validation Using MS ATL Server Libraries" [MSDN 2009c]. This provides an example (in C++) under Windows of doing client-server input validation using input validation routines provided by the ATL libraries.
- *Secure Programming Cookbook* by Viega and Messier [Viega 2003]. This book provides functions and programming strategies for performing input validation in C++.
- "Input Validation in Apache Struts Framework" [You 2009]. This article provides a good tutorial on how to perform input validation when programming in Java using the Apache Struts framework. Of general interest in the tutorial is the detailed specification of valid system input.

4.5.9 Sample Code

This sample code is an example of a structured input validation methodology in C++. Note that there are many other ways to implement the Input Validation pattern.

The basic architecture of the example implementation of the Input Validation pattern is to represent a single set of validation criteria as a `validator` class. A `validator` class is a class with a single static `validate()` method that takes a piece of input to validate and returns `true` if the input is valid and `false` if the input is invalid.

The following `validator` class checks to see if an integer falls within a defined range.

```
template <int lower, int upper> class InRange {

public:

  static bool validate(int item) {
    return ((item >= lower) && (item <= upper));
  }
};
```

The following `validator` class checks to see whether two integers will *not* overflow if multiplied together [Seacord 2008].

```
class NoOverflowOnMult {

public:

  static bool validate(int o1, int o2) {

    // This validation method only works if the size of a long long is
    // greater than double the size of an integer.
    assert(sizeof(long long) >= 2 * sizeof(int));

    signed long long tmp = (signed long long)o1 * (signed long long)o2;

    // If the product cannot be represented as a 32-bit integer,
    // there is overflow.
    return !((tmp > INT_MAX) || (tmp < INT_MIN));
  }
};
```

The following `validator` class checks to see if a string holds a valid name where a valid name contains only alphanumeric characters, contains exactly one space, and is less than a defined number of characters long.

```
template<int maxNameLen> class GoodName {

public:

  static bool validate(char *str) {

    // The name should contain no digits and exactly 1 space.
```

```
    unsigned int pos = 0;
    bool sawSpace = false;
    while ((pos < maxNameLen) && (str[pos] != '\0')) {
      // Are we looking at a space in the string we are checking?
      if (str[pos] == ' ') {
        // Is this the 2nd space in the string?
        if (sawSpace) {
          // The name has more than 1 space. It is not a valid name.
          return false;
        }
        // Track that we have seen 1 space.
        sawSpace = true;
      }
      else {
        // Is the current character an alphabetic character?
        if (!isalpha(str[pos])) {
          // The name contains at least 1 non-alphabetic character. It
          // is not a valid name.
          return false;
        }
      }
      // Advance to the next character.
      pos++;
    }

    // A valid name string is less than maxNameLen characters.
    if (str[pos] != '\0') {
      return false;
    }

    // If we get here the name is valid.
    return true;
  }
};
```

The `main()` program provides some examples of how to use the `validator` classes.

```
int main(int argc, const char* argv[]) {
  if (InRange<1,10>::validate(5)) {
    cout << "5 is valid input\n";
  }

  if (!InRange<1,10>::validate(15)) {
    cout << "15 is NOT valid input\n";
  }

  if (NoOverflowOnMult::validate(12, 33)) {
    cout << "12*33 will not overflow\n";
  }

  if (!(NoOverflowOnMult::validate(INT_MAX, 33))) {
    cout << "INT_MAX*33 WILL overflow\n";
  }
```

```
  if (GoodName<100>::validate("Corey Duffle")) {
    cout << "'Corey Duffle' is a valid name.\n";
  }

  if (!GoodName<100>::validate("Sir Chumley the 5th")) {
    cout << "'Sir Chumley the 5th' is NOT a valid name.\n";
  }
}
```

4.5.10 Example Resolved

The university has identified three sources of input to their (very simple) ERP system:

1. a database system
2. GET parameters from HTML forms
3. POST parameters from HTML forms

The university has written a utility library to read GET/POST parameters that allows the developers to easily specify a validity checking routine to use when reading GET/POST parameters. The developers are using a simple static analysis tool to ensure that all reads of GET/POST parameters occur only through the utility library. They have instituted formal code reviews of the input validation checking routines to ensure that all input validation criteria are implemented correctly.

The university is using a third-party database abstraction library that sanitizes all provided strings that are to be used in the creation of SQL queries and provides some basic sanity checking of the results of SQL queries.

4.5.11 Known Uses

Many web frameworks and languages and general programming libraries provide support for performing input validation and sanitization. Frameworks with known input validation support include

- Ruby on Rails
- Java Struts
- Pylons
- Django

The Secure XML-RPC Server Library uses the Input Validation secure design pattern.

4.6 Resource Acquisition Is Initialization (RAII)

4.6.1 Intent

The intent of the RAII pattern is to ensure that system resources are properly allocated and deallocated under all possible program execution paths. RAII ensures that program resources are properly handled by performing resource allocation and deallocation in an object's constructor and destructor, removing the need for external users of an object to handle the allocation and deallocation of the object's resources.