Defensive Coding

A Guide to Improving Software Security

Florian Weimer

Defensive Coding
A Guide to Improving Software Security
Edition 1.0

Author Florian Weimer *fweimer@redhat.com*

This document provides guidelines for improving software security through secure coding. It covers common programming languages and libraries, and focuses on concrete recommendations.

Part I. Programming Languages

The C Programming Language

1.1. The core language

C provides no memory safety. Most recommendations in this section deal with this aspect of the language.

1.1.1. Undefined behavior

Some C constructs are defined to be undefined by the C standard. This does not only mean that the standard does not describe what happens when the construct is executed. It also allows optimizing compilers such as GCC to assume that this particular construct is never reached. In some cases, this has caused GCC to optimize security checks away. (This is not a flaw in GCC or the C language. But C certainly has some areas which are more difficult to use than others.)

Common sources of undefined behavior are:

- out-of-bounds array accesses
- null pointer dereferences
- overflow in signed integer arithmetic

1.1.2. Recommendations for pointers and array handling

Always keep track of the size of the array you are working with. Often, code is more obviously correct when you keep a pointer past the last element of the array, and calculate the number of remaining elements by substracting the current position from that pointer. The alternative, updating a separate variable every time when the position is advanced, is usually less obviously correct.

Example 1.1, "Array processing in C" shows how to extract Pascal-style strings from a character buffer. The two pointers kept for length checks are `inend` and `outend`. `inp` and `outp` are the respective positions. The number of input bytes is checked using the expression **`len > (size_t) (inend - inp)`**. The cast silences a compiler warning; `inend` is always larger than `inp`.

Example 1.1. Array processing in C

```
ssize_t
extract_strings(const char *in, size_t inlen, char **out, size_t outlen)
{
  const char *inp = in;
  const char *inend = in + inlen;
  char **outp = out;
  char **outend = out + outlen;

  while (inp != inend) {
    size_t len;
    char *s;
    if (outp == outend) {
      errno = ENOSPC;
      goto err;
    }
    len = (unsigned char)*inp;
    ++inp;
    if (len > (size_t)(inend - inp)) {
      errno = EINVAL;
```

```
      goto err;
    }
    s = malloc(len + 1);
    if (s == NULL) {
      goto err;
    }
    memcpy(s, inp, len);
    inp += len;
    s[len] = '\0';
    *outp = s;
    ++outp;
  }
  return outp - out;
err:
  {
    int errno_old = errno;
    while (out != outp) {
      free(*out);
      ++out;
    }
    errno = errno_old;
  }
  return -1;
}
```

It is important that the length checks always have the form **`len > (size_t)(inend - inp)`**, where `len` is a variable of type size_t which denotes the *total* number of bytes which are about to be read or written next. In general, it is not safe to fold multiple such checks into one, as in **`len1 + len2 > (size_t)(inend - inp)`**, because the expression on the left can overflow or wrap around (see *Section 1.1.3, "Recommendations for integer arithmetic"*), and it no longer reflects the number of bytes to be processed.

1.1.3. Recommendations for integer arithmetic

Overflow in signed integer arithmetic is undefined. This means that it is not possible to check for overflow after it happened, see *Example 1.2, "Incorrect overflow detection in C"*.

Example 1.2. Incorrect overflow detection in C

```
void report_overflow(void);

int
add(int a, int b)
{
  int result = a + b;
  if (a < 0 || b < 0) {
    return -1;
  }
  // The compiler can optimize away the following if statement.
  if (result < 0) {
    report_overflow();
  }
  return result;
}
```

The following approaches can be used to check for overflow, without actually causing it.

- Use a wider type to perform the calculation, check that the result is within bounds, and convert the result to the original type. All intermediate results must be checked in this way.

- Perform the calculation in the corresponding unsigned type and use bit fiddling to detect the overflow.
- Compute bounds for acceptable input values which are known to avoid overflow, and reject other values. This is the preferred way for overflow checking on multiplications, see *Example 1.3, "Overflow checking for unsigned multiplication"*.

Example 1.3. Overflow checking for unsigned multiplication

```
unsigned
mul(unsigned a, unsigned b)
{
  if (b && a > ((unsigned)-1) / b) {
    report_overflow();
  }
  return a * b;
}
```

Basic arithmetic operations a commutative, so for bounds checks, there are two different but mathematically equivalent expressions. Sometimes, one of the expressions results in better code because parts of it can be reduced to a constant. This applies to overflow checks for multiplication **`a * b`** involving a constant **`a`**, where the expression is reduced to **`b > C`** for some constant **`C`** determined at compile time. The other expression, **`b && a > ((unsigned)-1) / b`**, is more difficult to optimize at compile time.

When a value is converted to a signed integer, GCC always chooses the result based on 2's complement arithmetic. This GCC extension (which is also implemented by other compilers) helps a lot when implementing overflow checks.

Legacy code should be compiled with the **`-fwrapv`** GCC option. As a result, GCC will provide 2's complement semantics for integer arithmetic, including defined behavior on integer overflow.

1.2. The C standard library

Parts of the C standard library (and the UNIX and GNU extensions) are difficult to use, so you shoud avoid them.

Please check the applicable documentation before using the recommended replacements. Many of these functions allocate buffers using `malloc` which your code must deallocate explicitly using `free`.

1.2.1. Absolutely banned interfaces

The functions listed below must not be used because they are almost always unsafe. Use the indicated replacements instead.

- `gets` # `fgets`
- `getwd` # `getcwd` or `get_current_dir_name`
- `readdir_r` # `readdir`
- `realpath` (with a non-NULL second parameter) # `realpath` with NULL as the second parameter, or `canonicalize_file_name`

The constants listed below must not be used, either. Instead, code must allocate memory dynamically and use interfaces with length checking.

- **`NAME_MAX`** (limit not actually enforced by the kernel)
- **`PATH_MAX`** (limit not actually enforced by the kernel)
- **`_PC_NAME_MAX`** (This limit, returned by the `pathconf` function, is not enforced by the kernel.)
- **`_PC_PATH_MAX`** (This limit, returned by the `pathconf` function, is not enforced by the kernel.)

The following structure members must not be used.

- **`f_namemax`** in **`struct statvfs`** (limit not actually enforced by the kernel, see **`_PC_NAME_MAX`** above)

1.2.2. Functions to avoid

The following string manipulation functions can be used securely in principle, but their use should be avoided because they are difficult to use correctly. Calls to these functions can be replaced with `asprintf` or `vasprintf`. (For non-GNU targets, these functions are available from Gnulib.) In some cases, the `snprintf` function might be a suitable replacement, see *Section 1.2.3, "String Functions With Explicit Length Arguments"*.

- `sprintf`
- `strcat`
- `strcpy`
- `vsprintf`

Use the indicated replacements for the functions below.

- `alloca` # `malloc` and `free` (see *Section 1.3.2, "`alloca` and other forms of stack-based allocation"*)
- `putenv` # explicit `envp` argument in process creation (see *Section 8.1.3, "Specifying the process environment"*)
- `setenv` # explicit `envp` argument in process creation (see *Section 8.1.3, "Specifying the process environment"*)
- `strdupa` # `strdup` and `free` (see *Section 1.3.2, "`alloca` and other forms of stack-based allocation"*)
- `strndupa` # `strndup` and `free` (see *Section 1.3.2, "`alloca` and other forms of stack-based allocation"*)
- `system` # `posix_spawn` or `fork`/`execve`/ (see *Section 8.1.2, "Bypassing the shell"*)
- `unsetenv` # explicit `envp` argument in process creation (see *Section 8.1.3, "Specifying the process environment"*)

1.2.3. String Functions With Explicit Length Arguments

The `snprintf` function provides a way to construct a string in a statically-sized buffer. (If the buffer size is dynamic, use `asprintf` instead.)

```
char fraction[30];
```

```
snprintf(fraction, sizeof(fraction), "%d/%d", numerator, denominator);
```

The second argument to the `snprintf` should always be the size of the buffer in the first argument (which should be a character array). Complex pointer and length arithmetic can introduce errors and nullify the security benefits of `snprintf`. If you need to construct a string iteratively, by repeatedly appending fragments, consider constructing the string on the heap, increasing the buffer with `realloc` as needed. (`snprintf` does not support overlapping the result buffer with argument strings.)

If you use `vsnprintf` (or `snprintf`) with a format string which is not a constant, but a function argument, it is important to annotate the function with a **`format`** function attribute, so that GCC can warn about misuse of your function (see *Example 1.4, "The **`format`** function attribute"*).

Example 1.4. The **`format`** function attribute

```
void log_format(const char *format, ...) __attribute__((format(printf, 1, 2)));

void
log_format(const char *format, ...)
{
  char buf[1000];
  va_list ap;
  va_start(ap, format);
  vsnprintf(buf, sizeof(buf), format, ap);
  va_end(ap);
  log_string(buf);
}
```

There are other functions which operator on NUL-terminated strings and take a length argument which affects the number of bytes written to the destination: `strncpy`, `strncat`, and `stpncpy`. These functions do not ensure that the result string is NUL-terminated. For `strncpy`, NUL termination can be added this way:

```
char buf[10];
strncpy(buf, data, sizeof(buf));
buf[sizeof(buf) - 1] = '\0';
```

Some systems support `strlcpy` and `strlcat` functions which behave this way, but these functions are not part of GNU libc. Using `snprintf` with a suitable format string is a simple (albeit slightly slower) replacement.

1.3. Memory allocators

1.3.1. `malloc` and related functions

The C library interfaces for memory allocation are provided by `malloc`, `free` and `realloc`, and the `calloc` function. In addition to these generic functions, there are derived functions such as `strdup` which perform allocation using `malloc` internally, but do not return untyped heap memory (which could be used for any object).

The C compiler knows about these functions and can use their expected behavior for optimizations. For instance, the compiler assumes that an existing pointer (or a pointer derived from an existing pointer by arithmetic) will not point into the memory area returned by `malloc`.

If the allocation fails, `realloc` does not free the old pointer. Therefore, the idiom **`ptr = realloc(ptr, size);`** is wrong because the memory pointed to by **`ptr`** leaks in case of an error.

1.3.1.1. Use-after-free errors

After `free`, the pointer is invalid. Further pointer dereferences are not allowed (and are usually detected by **valgrind**). Less obvious is that any *use* of the old pointer value is not allowed, either. In particular, comparisons with any other pointer (or the null pointer) are undefined according to the C standard.

The same rules apply to `realloc` if the memory area cannot be enlarged in-place. For instance, the compiler may assume that a comparison between the old and new pointer will always return false, so it is impossible to detect movement this way.

1.3.1.2. Handling memory allocation errors

Recovering from out-of-memory errors is often difficult or even impossible. In these cases, `malloc` and other allocation functions return a null pointer. Dereferencing this pointer lead to a crash. Such dereferences can even be exploitable for code execution if the dereference is combined with an array subscript.

In general, if you cannot check all allocation calls and handle failure, you should abort the program on allocation failure, and not rely on the null pointer dereference to terminate the process. See *Section 9.1, "Recommendations for manually written decoders"* for related memory allocation concerns.

1.3.2. `alloca` and other forms of stack-based allocation

Allocation on the stack is risky because stack overflow checking is implicit. There is a guard page at the end of the memory area reserved for the stack. If the program attempts to read from or write to this guard page, a **`SIGSEGV`** signal is generated and the program typically terminates.

This is sufficient for detecting typical stack overflow situations such as unbounded recursion, but it fails when the stack grows in increments larger than the size of the guard page. In this case, it is possible that the stack pointer ends up pointing into a memory area which has been allocated for a different purposes. Such misbehavior can be exploitable.

A common source for large stack growth are calls to `alloca` and related functions such as `strdupa`. These functions should be avoided because of the lack of error checking. (They can be used safely if the allocated size is less than the page size (typically, 4096 bytes), but this case is relatively rare.) Additionally, relying on `alloca` makes it more difficult to reorgnize the code because it is not allowed to use the pointer after the function calling `alloca` has returned, even if this function has been inlined into its caller.

Similar concerns apply to *variable-length arrays* (VLAs), a feature of the C99 standard which started as a GNU extension. For large objects exceeding the page size, there is no error checking, either.

In both cases, negative or very large sizes can trigger a stack-pointer wraparound, and the stack pointer and end up pointing into caller stack frames, which is fatal and can be exploitable.

If you want to use `alloca` or VLAs for performance reasons, consider using a small on-stack array (less than the page size, large enough to fulfill most requests). If the requested size is small enough, use the on-stack array. Otherwise, call `malloc`. When exiting the function, check if `malloc` had been called, and free the buffer as needed.

1.3.3. Array allocation

When allocating arrays, it is important to check for overflows. The `calloc` function performs such checks.

If `malloc` or `realloc` is used, the size check must be written manually. For instance, to allocate an array of **`n`** elements of type **`T`**, check that the requested size is not greater than **`n / sizeof(T)`**.

1.3.4. Custom memory allocators

Custom memory allocates come in two forms: replacements for `malloc`, and completely different interfaces for memory management. Both approaches can reduce the effectiveness of **valgrind** and similar tools, and the heap corruption detection provided by GNU libc, so they should be avoided.

Memory allocators are difficult to write and contain many performance and security pitfalls.

- When computing array sizes or rounding up allocation requests (to the next allocation granularity, or for alignment purposes), checks for arithmetic overflow are required.
- Size computations for array allocations need overflow checking. See *Section 1.3.3, “Array allocation”*.
- It can be difficult to beat well-tuned general-purpose allocators. In micro-benchmarks, pool allocators can show huge wins, and size-specific pools can reduce internal fragmentation. But often, utilization of individual pools is poor, and

1.3.5. Conservative garbage collection

Garbage collection can be an alternative to explicit memory management using `malloc` and `free`. The Boehm-Dehmers-Weiser allocator can be used from C programs, with minimal type annotations. Performance is competitive with `malloc` on 64-bit architectures, especially for multi-threaded programs. The stop-the-world pauses may be problematic for some real-time applications, though.

However, using a conservative garbage collector may reduce opertunities for code reduce because once one library in a program uses garbage collection, the whole process memory needs to be subject to it, so that no pointers are missed. The Boehm-Dehmers-Weiser collector also reserves certain signals for internal use, so it is not fully transparent to the rest of the program.

The C++ Programming Language

2.1. The core language

C++ includes a large subset of the C language. As far as the C subset is used, the recommendations in *Chapter 1, The C Programming Language* apply.

2.1.1. Array allocation with `operator new[]`

For very large values of **n**, an expression like **`new T[n]`** can return a pointer to a heap region which is too small. In other words, not all array elements are actually backed with heap memory reserved to the array. Current GCC versions generate code that performs a computation of the form **`sizeof(T) * size_t(n) + cookie_size`**, where **`cookie_size`** is currently at most 8. This computation can overflow, and GCC-generated code does not detect this.

The **`std::vector`** template can be used instead an explicit array allocation. (The GCC implementation detects overflow internally.)

If there is no alternative to **`operator new[]`**, code which allocates arrays with a variable length must check for overflow manually. For the **`new T[n]`** example, the size check could be **`n || (n > 0 && n > (size_t(-1) - 8) / sizeof(T))`**. (See *Section 1.1.3, "Recommendations for integer arithmetic"*.) If there are additional dimensions (which must be constants according to the C++ standard), these should be included as factors in the divisor.

These countermeasures prevent out-of-bounds writes and potential code execution. Very large memory allocations can still lead to a denial of service. *Section 9.1, "Recommendations for manually written decoders"* contains suggestions for mitigating this problem when processing untrusted data.

See *Section 1.3.3, "Array allocation"* for array allocation advice for C-style memory allocation.

2.1.2. Overloading

Do not overload functions with versions that have different security characteristics. For instance, do not implement a function `strcat` which works on std::string arguments. Similarly, do not name methods after such functions.

2.1.3. ABI compatibility and preparing for security updates

A stable binary interface (ABI) is vastly preferred for security updates. Without a stable ABI, all reverse dependencies need recompiling, which can be a lot of work and could even be impossible in some cases. Ideally, a security update only updates a single dynamic shared object, and is picked up automatically after restarting affected processes.

Outside of extremely performance-critical code, you should ensure that a wide range of changes is possible without breaking ABI. Some very basic guidelines are:

- Avoid inline functions.
- Use the pointer-to-implementation idiom.
- Try to avoid templates. Use them if the increased type safety provides a benefit to the programmer.
- Move security-critical code out of templated code, so that it can be patched in a central place if necessary.

The KDE project publishes a document with more extensive guidelines on ABI-preserving changes to C++ code, *Policies/Binary Compatibility Issues With C++*[1] (*d-pointer* refers to the pointer-to-implementation idiom).

2.1.4. C++0X and C++11 support

GCC offers different language compatibility modes:

- **`-std=c++98`** for the original 1998 C++ standard
- **`-std=c++03`** for the 1998 standard with the changes from the TR1 technical report
- **`-std=c++11`** for the 2011 C++ standard. This option should not be used.
- **`-std=c++0x`** for several different versions of C++11 support in development, depending on the GCC version. This option should not be used.

For each of these flags, there are variants which also enable GNU extensions (mostly language features also found in C99 or C11): **`-std=gnu++98`**, **`-std=gnu++03`**, **`-std=gnu++11`**. Again, **`-std=gnu++11`** should not be used.

If you enable C++11 support, the ABI of the standard C++ library **`libstdc++`** will change in subtle ways. Currently, no C++ libraries are compiled in C++11 mode, so if you compile your code in C++11 mode, it will be incompatible with the rest of the system. Unfortunately, this is also the case if you do not use any C++11 features. Currently, there is no safe way to enable C++11 mode (except for freestanding applications).

The meaning of C++0X mode changed from GCC release to GCC release. Earlier versions were still ABI-compatible with C++98 mode, but in the most recent versions, switching to C++0X mode activates C++11 support, with its compatibility problems.

Some C++11 features (or approximations thereof) are available with TR1 support, that is, with **`-std=c++03`** or **`-std=gnu++03`** and in the **`<tr1/*>`** header files. This includes **`std::tr1::shared_ptr`** (from **`<tr1/memory>`**) and **`std::tr1::function`** (from **`<tr1/functional>`**). For other C++11 features, the Boost C++ library contains replacements.

2.2. The C++ standard library

The C++ standard library includes most of its C counterpart by reference, see *Section 1.2, “The C standard library”*.

2.2.1. Containers and `operator[]`

Many containers similar to **`std::vector`** provide both **`operator[](size_type)`** and a member function **`at(size_type)`**. This applies to **`std::vector`** itself, **`std::array`**, **`std::string`** and other instances of **`std::basic_string`**.

`operator[](size_type)` is not required by the standard to perform bounds checking (and the implementation in GCC does not). In contrast, **`at(size_type)`** must perform such a check. Therefore, in code which is not performance-critical, you should prefer **`at(size_type)`** over **`operator[](size_type)`**, even though it is slightly more verbose.

[1] http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++

The Python Programming Language

Python provides memory safety by default, so low-level security vulnerabilities are rare and typically needs fixing the Python interpreter or standard library itself.

Other sections with Python-specific advice include:

- *Chapter 7, Temporary files*
- *Section 8.1, “Safe process creation”*
- *Chapter 9, Serialization and Deserialization*, in particular *Section 9.3, “Library support for deserialization”*
- *Section 10.2, “Randomness”*

3.1. Dangerous standard library features

Some areas of the standard library, notably the **`ctypes`** module, do not provide memory safety guarantees comparable to the rest of Python. If such functionality is used, the advice in *Section 1.1, “The core language”* should be followed.

3.2. Run-time compilation and code generation

The following Python functions and statements related to code execution should be avoided:

- `compile`
- `eval`
- **`exec`**
- `execfile`

If you need to parse integers or floating point values, use the `int` and `float` functions instead of `eval`. Sandboxing untrusted Python code does not work reliably.

3.3. Sandboxing

The **`rexec`** Python module cannot safely sandbox untrusted code and should not be used. The standard CPython implementation is not suitable for sandboxing.

Part II. Specific Programming Tasks

Library Design

Throught this section, the term *client code* refers to applications and other libraries using the library.

4.1. State management

4.1.1. Global state

Global state should be avoided.

If this is impossible, the global state must be protected with a lock. For C/C++, you can use the `pthread_mutex_lock` and `pthread_mutex_unlock` functions without linking against **`-lpthread`** because the system provides stubs for non-threaded processes.

For compatibility with `fork`, these locks should be acquired and released in helpers registered with `pthread_atfork`. This function is not available without **`-lpthread`**, so you need to use `dlsym` or a weak symbol to obtain its address.

If you need `fork` protection for other reasons, you should store the process ID and compare it to the value returned by `getpid` each time you access the global state. (`getpid` is not implemented as a system call and is fast.) If the value changes, you know that you have to re-create the state object. (This needs to be combined with locking, of course.)

4.1.2. Handles

Library state should be kept behind a curtain. Client code should receive only a handle. In C, the handle can be a pointer to an incomplete **`struct`**. In C++, the handle can be a pointer to an abstract base class, or it can be hidden using the pointer-to-implementation idiom.

The library should provide functions for creating and destroying handles. (In C++, it is possible to use virtual destructors for the latter.) Consistency between creation and destruction of handles is strongly recommended: If the client code created a handle, it is the responsibility of the client code to destroy it. (This is not always possible or convenient, so sometimes, a transfer of ownership has to happen.)

Using handles ensures that it is possible to change the way the library represents state in a way that is transparent to client code. This is important to facilitate security updates and many other code changes.

It is not always necessary to protect state behind a handle with a lock. This depends on the level of thread safety the library provides.

4.2. Object orientation

Classes should be either designed as base classes, or it should be impossible to use them as base classes (like **`final`** classes in Java). Classes which are not designed for inheritance and are used as base classes nevertheless create potential maintenance hazards because it is difficult to predict how client code will react when calls to virtual methods are added, reordered or removed.

Virtual member functions can be used as callbacks. See *Section 4.3, “Callbacks”* for some of the challenges involved.

4.3. Callbacks

Higher-order code is difficult to analyze for humans and computers alike, so it should be avoided. Often, an iterator-based interface (a library function which is called repeatedly by client code and returns a stream of events) leads to a better design which is easier to document and use.

If callbacks are unavoidable, some guidelines for them follow.

In modern C++ code, **`std::function`** objects should be used for callbacks.

In older C++ code and in C code, all callbacks must have an additional closure parameter of type **`void *`**, the value of which can be specified by client code. If possible, the value of the closure parameter should be provided by client code at the same time a specific callback is registered (or specified as a function argument). If a single closure parameter is shared by multiple callbacks, flexibility is greatly reduced, and conflicts between different pieces of client code using the same library object could be unresolvable. In some cases, it makes sense to provide a de-registration callback which can be used to destroy the closure parameter when the callback is no longer used.

Callbacks can throw exceptions or call `longjmp`. If possible, all library objects should remain in a valid state. (All further operations on them can fail, but it should be possible to deallocate them without causing resource leaks.)

The presence of callbacks raises the question if functions provided by the library are *reentrant*. Unless a library was designed for such use, bad things will happen if a callback function uses functions in the same library (particularly if they are invoked on the same objects and manipulate the same state). When the callback is invoked, the library can be in an inconsistent state. Reentrant functions are more difficult to write than thread-safe functions (by definition, simple locking would immediately lead to deadlocks). It is also difficult to decide what to do when destruction of an object which is currently processing a callback is requested.

4.4. Process attributes

Several attributes are global and affect all code in the process, not just the library that manipulates them.

- environment variables (see *Section 8.3.1, “Accessing environment variables”*)
- umask
- user IDs, group IDs and capabilities
- current working directory
- signal handlers, signal masks and signal delivery
- file locks (especially `fcntl` locks behave in surprising ways, not just in a multi-threaded environment)

Library code should avoid manipulating these global process attributes. It should not rely on environment variables, umask, the current working directory and signal masks because these attributes can be inherted from an untrusted source.

In addition, there are obvious process-wide aspects such as the virtual memory layout, the set of open files and dynamic shared objects, but with the exception of shared objects, these can be manipulated in a relatively isolated way.

File Descriptor Management

File descriptors underlie all input/output mechanisms offered by the system. They are used to implementation the **`FILE *`**-based functions found in **`<stdio.h>`**, and all the file and network communication facilities provided by the Python and Java environments are eventually implemented in them.

File descriptors are small, non-negative integers in userspace, and are backed on the kernel side with complicated data structures which can sometimes grow very large.

5.1. Closing descriptors

If a descriptor is no longer used by a program and is not closed explicitly, its number cannot be reused (which is problematic in itself, see *Section 5.3, “Dealing with the `select` limit”*), and the kernel resources are not freed. Therefore, it is important to close all descriptors at the earlierst point in time possible, but not earlier.

5.1.1. Error handling during descriptor close

The `close` system call is always successful in the sense that the passed file descriptor is never valid after the function has been called. However, `close` still can return an error, for example if there was a file system failure. But this error is not very useful because the absence of an error does not mean that all caches have been emptied and previous writes have been made durable. Programs which need such guarantees must open files with **`O_SYNC`** or use **`fsync`** or **`fdatasync`**, and may also have to **`fsync`** the directory containing the file.

5.1.2. Closing descriptors and race conditions

Unlike process IDs, which are recycle only gradually, the kernel always allocates the lowest unused file descriptor when a new descriptor is created. This means that in a multi-threaded program which constantly opens and closes file descriptors, descriptors are reused very quickly. Unless descriptor closing and other operations on the same file descriptor are synchronized (typically, using a mutex), there will be race coniditons and I/O operations will be applied to the wrong file descriptor.

Sometimes, it is necessary to close a file descriptor concurrently, while another thread might be about to use it in a system call. In order to support this, a program needs to create a single special file descriptor, one on which all I/O operations fail. One way to achieve this is to use `socketpair`, close one of the descriptors, and call **`shutdown(fd, SHUTRDWR)`** on the other.

When a descriptor is closed concurrently, the program does not call `close` on the descriptor. Instead it program uses `dup2` to replace the descriptor to be closed with the dummy descriptor created earlier. This way, the kernel will not reuse the descriptor, but it will carry out all other steps associated with calling a descriptor (for instance, if the descriptor refers to a stream socket, the peer will be notified).

This is just a sketch, and many details are missing. Additional data structures are needed to determine when it is safe to really close the descriptor, and proper locking is required for that.

5.1.3. Lingering state after close

By default, closing a stream socket returns immediately, and the kernel will try to send the data in the background. This means that it is impossible to implement accurate accounting of network-related resource utilization from userspace.

The **`SO_LINGER`** socket option alters the behavior of `close`, so that it will return only after the lingering data has been processed, either by sending it to the peer successfully, or by discarding it

after the configured timeout. However, there is no interface which could perform this operation in the background, so a separate userspace thread is needed for each `close` call, causing scalability issues.

Currently, there is no application-level countermeasure which applies universally. Mitigation is possible with **iptables** (the **`connlimit`** match type in particular) and specialized filtering devices for denial-of-service network traffic.

These problems are not related to the **`TIME_WAIT`** state commonly seen in **netstat** output. The kernel automatically expires such sockets if necessary.

5.2. Preventing file descriptor leaks to child processes

Child processes created with `fork` share the initial set of file descriptors with their parent process. By default, file descriptors are also preserved if a new process image is created with `execve` (or any of the other functions such as `system` or `posix_spawn`).

Usually, this behavior is not desirable. There are two ways to turn it off, that is, to prevent new process images from inheriting the file descriptors in the parent process:

- Set the close-on-exec flag on all newly created file descriptors. Traditionally, this flag is controlled by the **`FD_CLOEXEC`** flag, using **`F_GETFD`** and **`F_SETFD`** operations of the `fcntl` function.

  However, in a multi-threaded process, there is a race condition: a subprocess could have been created between the time the descriptor was created and the **`FD_CLOEXEC`** was set. Therefore, many system calls which create descriptors (such as `open` and `openat`) now accept the `O_CLOEXEC` flag (`SOCK_CLOEXEC` for `socket` and `socketpair`), which cause the **`FD_CLOEXEC`** flag to be set for the file descriptor in an atomic fashion. In addition, a few new systems calls were introduced, such as `pipe2` and `dup3`.

  The downside of this approach is that every descriptor needs to receive special treatment at the time of creation, otherwise it is not completely effective.

- After calling `fork`, but before creating a new process image with `execve`, all file descriptors which the child process will not need are closed.

  Traditionally, this was implemented as a loop over file descriptors ranging from **`3`** to **`255`** and later **`1023`**. But this is only an approximatio because it is possible to create file descriptors outside this range easily (see *Section 5.3, "Dealing with the `select` limit"*). Another approach reads **`/proc/self/fd`** and closes the unexpected descriptors listed there, but this approach is much slower.

At present, environments which care about file descriptor leakage implement the second approach. OpenJDK 6 and 7 are among them.

5.3. Dealing with the `select` limit

By default, a user is allowed to open only 1024 files in a single process, but the system administrator can easily change this limit (which is necessary for busy network servers). However, there is another restriction which is more difficult to overcome.

The `select` function only supports a maximum of **`FD_SETSIZE`** file descriptors (that is, the maximum permitted value for a file descriptor is **`FD_SETSIZE - 1`**, usually 1023.) If a process opens many files, descriptors may exceed such limits. It is impossible to query such descriptors using `select`.

If a library which creates many file descriptors is used in the same process as a library which uses `select`, at least one of them needs to be changed. Calls to `select` can be replaced with calls to `poll` or another event handling mechanism.

Alternatively, the library with high descriptor usage can relocate descriptors above the **`FD_SETSIZE`** limit using the following procedure.

- Create the file descriptor **`fd`** as usual, preferably with the **`O_CLOEXEC`** flag.
- Before doing anything else with the descriptor **`fd`**, invoke:

```
int newfd = fcntl(fd, F_DUPFD_CLOEXEC, (long)FD_SETSIZE);
```

- Check that **`newfd`** result is non-negative, otherwise close **`fd`** and report an error, and return.
- Close **`fd`** and continue to use **`newfd`**.

The new descriptor has been allocated above the **`FD_SETSIZE`**. Even though this algorithm is racy in the sense that the **`FD_SETSIZE`** first descriptors could fill up, a very high degree of physical parallelism is required before this becomes a problem.

File system manipulation

In this chapter, we discuss general file system manipulation, with a focus on access files and directories to which an other, potentially untrusted user has write access.

Temporary files are covered in their own chapter, *Chapter 7, Temporary files*.

6.1. Working with files and directories owned by other users

Sometimes, it is necessary to operate on files and directories owned by other (potentially untrusted) users. For example, a system administrator could remove the home directory of a user, or a package manager could update a file in a directory which is owned by an application-specific user. This differs from accessing the file system as a specific user; see *Section 6.2, "Accessing the file system as a different user"*.

Accessing files across trust boundaries faces several challenges, particularly if an entire directory tree is being traversed:

1. Another user might add file names to a writable directory at any time. This can interfere with file creation and the order of names returned by `readdir`.

2. Merely opening and closing a file can have side effects. For instance, an automounter can be triggered, or a tape device rewound. Opening a file on a local file system can block indefinitely, due to mandatory file locking, unless the **`O_NONBLOCK`** flag is specified.

3. Hard links and symbolic links can redirect the effect of file system operations in unexpected ways. The **`O_NOFOLLOW`** and **`AT_SYMLINK_NOFOLLOW`** variants of system calls only affected final path name component.

4. The structure of a directory tree can change. For example, the parent directory of what used to be a subdirectory within the directory tree being processed could suddenly point outside that directory tree.

Files should always be created with the **`O_CREAT`** and **`O_EXCL`** flags, so that creating the file will fail if it already exists. This guards against the unexpected appearance of file names, either due to creation of a new file, or hard-linking of an existing file. In multi-threaded programs, rather than manipulating the umask, create the files with mode **`000`** if possible, and adjust it afterwards with `fchmod`.

To avoid issues related to symbolic links and directory tree restructuring, the "**`at`**" variants of system calls have to be used (that is, functions like `openat`, `fchownat`, `fchmodat`, and `unlinkat`, together with **`O_NOFOLLOW`** or **`AT_SYMLINK_NOFOLLOW`**). Path names passed to these functions must have just a single component (that is, without a slash). When descending, the descriptors of parent directories must be kept open. The missing **`opendirat`** function can be emulated with **`openat`** (with an **`O_DIRECTORY`** flag, to avoid opening special files with side effects), followed by **`fdopendir`**.

If the "**`at`**" functions are not available, it is possible to emulate them by changing the current directory. (Obviously, this only works if the process is not multi-threaded.) `fchdir` has to be used to change the current directory, and the descriptors of the parent directories have to be kept open, just as with the "**`at`**"-based approach. **`chdir("...")`** is unsafe because it might ascend outside the intended directory tree.

This "**`at`**" function emulation is currently required when manipulating extended attributes. In this case, the `lsetxattr` function can be used, with a relative path name consisting of a single component. This also applies to SELinux contexts and the `lsetfilecon` function.

Currently, it is not possible to avoid opening special files *and* changes to files with hard links if the directory containing them is owned by an untrusted user. (Device nodes can be hard-linked, just as regular files.) `fchmodat` and `fchownat` affect files whose link count is greater than one. But opening the files, checking that the link count is one with `fstat`, and using `fchmod` and `fchown` on the file descriptor may have unwanted side effects, due to item 2 above. When creating directories, it is therefore important to change the ownership and permissions only after it has been fully created. Until that point, file names are stable, and no files with unexpected hard links can be introduced.

Similarly, when just reading a directory owned by an untrusted user, it is currently impossible to reliably avoid opening special files.

There is no workaround against the instability of the file list returned by `readdir`. Concurrent modification of the directory can result in a list of files being returned which never actually existed on disk.

Hard links and symbolic links can be safely deleted using `unlinkat` without further checks because deletion only affects the name within the directory tree being processed.

6.2. Accessing the file system as a different user

This section deals with access to the file system as a specific user. This is different from accessing files and directories owned by a different, potentially untrusted user; see *Section 6.2, "Accessing the file system as a different user"*.

One approach is to spawn a child process which runs under the target user and group IDs (both effective and real IDs). Note that this child process can block indefinitely, even when processing regular files only. For example, a special FUSE file system could cause the process to hang in uninterruptible sleep inside a `stat` system call.

An existing process could change its user and group ID using `setfsuid` and `setfsgid`. (These functions are preferred over `seteuid` and `setegid` because they do not allow the impersonated user to send signals to the process.) These functions are not thread safe. In multi-threaded processes, these operations need to be performed in a single-threaded child process. Unexpected blocking may occur as well.

It is not recommended to try to reimplement the kernel permission checks in user space because the required checks are complex. It is also very difficult to avoid race conditions during path name resolution.

6.3. File system limits

For historical reasons, there are preprocessor constants such as **`PATH_MAX`**, **`NAME_MAX`**. However, on most systems, the length of canonical path names (absolute path names with all symbolic links resolved, as returned by `realpath` or `canonicalize_file_name`) can exceed **`PATH_MAX`** bytes, and individual file name components can be longer than **`NAME_MAX`**. This is also true of the **`_PC_PATH_MAX`** and **`_PC_NAME_MAX`** values returned by `pathconf`, and the **`f_namemax`** member of **`struct statvfs`**. Therefore, these constants should not be used. This is also reason why the `readdir_r` should never be used (instead, use `readdir`).

You should not write code in a way that assumes that there is an upper limit on the number of subdirectories of a directory, the number of regular files in a directory, or the link count of an inode.

6.4. File system features

Not all file systems support all features. This makes it very difficult to write general-purpose tools for copying files. For example, a copy operation intending to preserve file permissions will generally fail when copying to a FAT file system.

- Some file systems are case-insensitive. Most should be case-preserving, though.
- Name length limits vary greatly, from eight to thousands of bytes. Path length limits differ as well. Most systems impose an upper bound on path names passed to the kernel, but using relative path names, it is possible to create and access files whose absolute path name is essentially of unbounded length.
- Some file systems do not store names as fairly unrestricted byte sequences, as it has been traditionally the case on GNU systems. This means that some byte sequences (outside the POSIX safe character set) are not valid names. Conversely, names of existing files may not be representable as byte sequences, and the files are thus inaccessible on GNU systems. Some file systems perform Unicode canonicalization on file names. These file systems preserve case, but reading the name of a just-created file using `readdir` might still result in a different byte sequence.
- Permissions and owners are not universally supported (and SUID/SGID bits may not be available). For example, FAT file systems assign ownership based on a mount option, and generally mark all files as executable. Any attempt to change permissions would result in an error.
- Non-regular files (device nodes, FIFOs) are not generally available.
- Only on some file systems, files can have holes, that is, not all of their contents is backed by disk storage.
- `ioctl` support (even fairly generic functionality such as **`FIEMAP`** for discovering physical file layout and holes) is file-system-specific.
- Not all file systems support extended attributes, ACLs and SELinux metadata. Size and naming restriction on extended attributes vary.
- Hard links may not be supported at all (FAT) or only within the same directory (AFS). Symbolic links may not be available, either. Reflinks (hard links with copy-on-write semantics) are still very rare. Recent systems restrict creation of hard links to users which own the target file or have read/write access to it, but older systems do not.
- Renaming (or moving) files using `rename` can fail (even when `stat` indicates that the source and target directories are located on the same file system). This system call should work if the old and new paths are located in the same directory, though.
- Locking semantics vary among file systems. This affects advisory and mandatory locks. For example, some network file systems do not allow deleting files which are opened by any process.
- Resolution of time stamps varies from two seconds to nanoseconds. Not all time stamps are available on all file systems. File creation time (*birth time*) is not exposed over the `stat`/`fstat` interface, even if stored by the file system.

6.5. Checking free space

The `statvfs` and `fstatvfs` functions allow programs to examine the number of available blocks and inodes, through the members **`f_bfree`**, **`f_bavail`**, **`f_ffree`**, and **`f_favail`** of **`struct statvfs`**. Some file systems return fictional values in the **`f_ffree`** and **`f_favail`** fields, so the only

reliable way to discover if the file system still has space for a file is to try to create it. The **`f_bfree`** field should be reasonably accurate, though.

Temporary files

In this chapter, we describe how to create temporary files and directories, how to remove them, and how to work with programs which do not create files in ways that a safe with a shared directory for temporary files. General file system manipulation is treated in a separate chapter, *Chapter 6, File system manipulation*.

Secure creation of temporary files has four different aspects.

- The location of the directory for temporary files must be obtained in a secure manner (that is, untrusted environment variables must be ignored, see *Section 8.3.1, "Accessing environment variables"*).
- A new file must be created. Reusing an existing file must be avoided (the **`/tmp`** race condition). This is tricky because traditionally, system-wide temporary directories shared by all users are used.
- The file must be created in a way that makes it impossible for other users to open it.
- The descriptor for the temporary file should not leak to subprocesses.

All functions mentioned below will take care of these aspects.

Traditionally, temporary files are often used to reduce memory usage of programs. More and more systems use RAM-based file systems such as **`tmpfs`** for storing temporary files, to increase performance and decrease wear on Flash storage. As a result, spooling data to temporary files does not result in any memory savings, and the related complexity can be avoided if the data is kept in process memory.

7.1. Obtaining the location of temporary directory

Some functions below need the location of a directory which stores temporary files. For C/C++ programs, use the following steps to obtain that directory:

- Use `secure_getenv` to obtain the value of the **`TMPDIR`** environment variable. If it is set, convert the path to a fully-resolved absolute path, using **`realpath(path, NULL)`**. Check if the new path refers to a directory and is writeable. In this case, use it as the temporary directory.
- Fall back to **`/tmp`**.

In Python, you can use the `tempfile.tempdir` variable.

Java does not support SUID/SGID programs, so you can use the `java.lang.System.getenv(String)` method to obtain the value of the **`TMPDIR`** environment variable, and follow the two steps described above. (Java's default directory selection does not honor **`TMPDIR`**.)

7.2. Named temporary files

The `mkostemp` function creates a named temporary file. You should specify the **`O_CLOEXEC`** flag to avoid file descriptor leaks to subprocesses. (Applications which do not use multiple threads can also use `mkstemp`, but libraries should use `mkostemp`.) For determining the directory part of the file name pattern, see *Section 7.1, "Obtaining the location of temporary directory"*.

The file is not removed automatically. It is not safe to rename or delete the file before processing, or transform the name in any way (for example, by adding a file extension). If you need multiple temporary files, call `mkostemp` multiple times. Do not create additional file names derived from the

name provided by a previous `mkostemp` call. However, it is safe to close the descriptor returned by `mkostemp` and reopen the file using the generated name.

The Python class **`tempfile.NamedTemporaryFile`** provides similar functionality, except that the file is deleted automatically by default. Note that you may have to use the **`file`** attribute to obtain the actual file object because some programming interfaces cannot deal with file-like objects. The C function `mkostemp` is also available as `tempfile.mkstemp`.

In Java, you can use the `java.io.File.createTempFile(String, String, File)` function, using the temporary file location determined according to *Section 7.1, "Obtaining the location of temporary directory"*. Do not use `java.io.File.deleteOnExit()` to delete temporary files, and do not register a shutdown hook for each temporary file you create. In both cases, the deletion hint cannot be removed from the system if you delete the temporary file prior to termination of the VM, causing a memory leak.

7.3. Temporary files without names

The `tmpfile` function creates a temporary file and immediately deletes it, while keeping the file open. As a result, the file lacks a name and its space is deallocated as soon as the file descriptor is closed (including the implicit close when the process terminates). This avoids cluttering the temporary directory with orphaned files.

Alternatively, if the maximum size of the temporary file is known beforehand, the `fmemopen` function can be used to create a **`FILE *`** object which is backed by memory.

In Python, unnamed temporary files are provided by the **`tempfile.TemporaryFile`** class, and the **`tempfile.SpooledTemporaryFile`** class provides a way to avoid creation of small temporary files.

Java does not support unnamed temporary files.

7.4. Temporary directories

The `mkdtemp` function can be used to create a temporary directory. (For determining the directory part of the file name pattern, see *Section 7.1, "Obtaining the location of temporary directory"*.) The directory is not automatically removed. In Python, this function is available as `tempfile.mkdtemp`. In Java 7, temporary directories can be created using the `java.nio.file.Files.createTempDirectory(Path, String, FileAttribute...)` function.

When creating files in the temporary directory, use automatically generated names, e.g., derived from a sequential counter. Files with externally provided names could be picked up in unexpected contexts, and crafted names could actually point outside of the tempoary directory (due to *directory traversal*).

Removing a directory tree in a completely safe manner is complicated. Unless there are overriding performance concerns, the **rm** program should be used, with the **`-rf`** and **`--`** options.

7.5. Compensating for unsafe file creation

There are two ways to make a function or program which excepts a file name safe for use with temporary files. See *Section 8.1, "Safe process creation"*, for details on subprocess creation.

- Create a temporary directory and place the file there. If possible, run the program in a subprocess which uses the temporary directory as its current directory, with a restricted environment. Use generated names for all files in that temporary directory. (See *Section 7.4, "Temporary directories"*.)

- Create the temporary file and pass the generated file name to the function or program. This only works if the function or program can cope with a zero-length existing file. It is safe only under additional assumptions:
  - The function or program must not create additional files whose name is derived from the specified file name or are otherwise predictable.
  - The function or program must not delete the file before processing it.
  - It must not access any existing files in the same directory.

  It is often difficult to check whether these additional assumptions are matched, therefore this approach is not recommended.

Processes

8.1. Safe process creation

This section describes how to create new child processes in a safe manner. In addition to the concerns addressed below, there is the possibility of file descriptor leaks, see *Section 5.2, "Preventing file descriptor leaks to child processes"*.

8.1.1. Obtaining the program path and the command line template

The name and path to the program being invoked should be hard-coded or controlled by a static configuration file stored at a fixed location (at an file system absolute path). The same applies to the template for generating the command line.

The configured program name should be an absolute path. If it is a relative path, the contents of the PATH must be obtained in s secure manner (see *Section 8.3.1, "Accessing environment variables"*). If the PATH variable is not set or untrusted, the safe default **`/bin:/usr/bin`** must be used.

If too much flexibility is provided here, it may allow invocation of arbitrary programs without proper authorization.

8.1.2. Bypassing the shell

Child processes should be created without involving the system shell.

For C/C++, `system` should not be used. The `posix_spawn` function can be used instead, or a combination `fork` and `execve`. (In some cases, it may be preferable to use `vfork` or the Linux-specific `clone` system call instead of `fork`.)

In Python, the **`subprocess`** module bypasses the shell by default (when the **`shell`** keyword argument is not set to true). `os.system` should not be used.

The Java class java.lang.ProcessBuilder can be used to create subprocesses without interference from the system shell.

Portability notice

On Windows, there is no argument vector, only a single argument string. Each application is responsible for parsing this string into an argument vector. There is considerable variance among the quoting style recognized by applications. Some of them expand shell wildcards, others do not. Extensive application-specific testing is required to make this secure.

Note that some common applications (notably **ssh**) unconditionally introduce the use of a shell, even if invoked directly without a shell. It is difficult to use these applications in a secure manner. In this case, untrusted data should be supplied by other means. For example, standard input could be used, instead of the command line.

8.1.3. Specifying the process environment

Child processes should be created with a minimal set of environment variables. This is absolutely essential if there is a trust transition involved, either when the parent process was created, or during the creation of the child process.

In C/C++, the environment should be constructed as an array of strings and passed as the `envp` argument to `posix_spawn` or `execve`. The functions `setenv`, `unsetenv` and `putenv` should not be used. They are not thread-safe and suffer from memory leaks.

Python programs need to specify a **`dict`** for the the `env` argument of the `subprocess.Popen` constructor. The Java class **`java.lang.ProcessBuilder`** provides a `environment()` method, which returns a map that can be manipulated.

The following list provides guidelines for selecting the set of environment variables passed to the child process.

- `PATH` should be initialized to **`/bin:/usr/bin`**.
- `USER` and `HOME` can be inhereted from the parent process environment, or they can be initialized from the **`pwent`** structure for the user.
- The `DISPLAY` and `XAUTHORITY` variables should be passed to the subprocess if it is an X program. Note that this will typically not work across trust boundaries because `XAUTHORITY` refers to a file with **`0600`** permissions.
- The location-related environment variables `LANG`, `LANGUAGE`, `LC_ADDRESS`, `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_IDENTIFICATION`, `LC_MEASUREMENT`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NAME`, `LC_NUMERIC`, `LC_PAPER`, `LC_TELEPHONE` and `LC_TIME` can be passed to the subprocess if present.
- The called process may need application-specific environment variables, for example for passing passwords. (See *Section 8.1.5, “Passing secrets to subprocesses”*.)
- All other environment variables should be dropped. Names for new environment variables should not be accepted from untrusted sources.

8.1.4. Robust argument list processing

When invoking a program, it is sometimes necessary to include data from untrusted sources. Such data should be check against embedded **`NUL`** characters because the system APIs will sliently truncate argument strings at the first **`NUL`** character.

The following recommendations assume that the program being invoked uses GNU-style option processing using `getopt_long`. This convention is widely used, but it is just that, and individual programs might interpret a command line in a different way.

If the untrusted data has to go into an option, use the **`--option-name=VALUE`** syntax, placing the option and its value into the same command line argument. This avoids any potential confusion if the data starts with **`-`**.

For positional arguments, terminate the option list with a single **`--`** marker after the last option, and include the data at the right position. The **`--`** marker terminates option processing, and the data will not be treated as an option even if it starts with a dash.

8.1.5. Passing secrets to subprocesses

The command line (the name of the program and its argument) of a running process is traditionally available to all local users. The called program can overwrite this information, but only after it has run for a bit of time, during which the information may have been read by other processes. However, on Linux, the process environment is restricted to the user who runs the process. Therefore, if you need a convenient way to pass a password to a child process, use an environment variable, and not a command line argument. (See *Section 8.1.3, "Specifying the process environment"*.)

Portability notice

On some UNIX-like systems (notably Solaris), environment variables can be read by any system user, just like command lines.

If the environment-based approach cannot be used due to portability concerns, the data can be passed on standard input. Some programs (notably **gpg**) use special file descriptors whose numbers are specified on the command line. Temporary files are an option as well, but they might give digital forensics access to sensitive data (such as passphrases) because it is difficult to safely delete them in all cases.

8.2. Handling child process termination

When child processes terminate, the parent process is signalled. A stub of the terminated processes (a *zombie*, shown as **<defunct>** by **ps**) is kept around until the status information is collected (*reaped*) by the parent process. Over the years, several interfaces for this have been invented:

- The parent process calls `wait`, `waitpid`, `waitid`, `wait3` or `wait4`, without specifying a process ID. This will deliver any matching process ID. This approach is typically used from within event loops.
- The parent process calls `waitpid`, `waitid`, or `wait4`, with a specific process ID. Only data for the specific process ID is returned. This is typically used in code which spawns a single subprocess in a synchronous manner.
- The parent process installs a handler for the **SIGCHLD** signal, using `sigaction`, and specifies to the **SA_NOCLDWAIT** flag. This approach could be used by event loops as well.

None of these approaches can be used to wait for child process terminated in a completely thread-safe manner. The parent process might execute an event loop in another thread, which could pick up the termination signal. This means that libraries typically cannot make free use of child processes (for example, to run problematic code with reduced privileges in a separate address space).

At the moment, the parent process should explicitly wait for termination of the child process using `waitpid` or `waitpid`, and hope that the status is not collected by an event loop first.

8.3. SUID/SGID processes

Programs can be marked in the file system to indicate to the kernel that a trust transition should happen if the program is run. The **SUID** file permission bit indicates that an executable should run with the effective user ID equal to the owner of the executable file. Similarly, with the **SGID** bit, the effective group ID is set to the group of the executable file.

Linux supports *fscaps*, which can grant additional capabilities to a process in a finer-grained manner. Additional mechanisms can be provided by loadable security modules.

When such a trust transition has happened, the process runs in a potentially hostile environment. Additional care is necessary not to rely on any untrusted information. These concerns also apply to libraries which can be linked into such processes.

8.3.1. Accessing environment variables

The following steps are required so that a program does not accidentally pick up untrusted data from environment variables.

- Compile your C/C++ sources with **`-D_GNU_SOURCE`**. The Autoconf macro **`AC_GNU_SOURCE`** ensures this.
- Check for the presence of the `secure_getenv` and `__secure_getenv` function. The Autoconf directive **`AC_CHECK_FUNCS([__secure_getenv secure_getenv])`** performs these checks.
- Arrange for a proper definition of the `secure_getenv` function. See *Example 8.1, "Obtaining a definition for `secure_getenv`"*.
- Use `secure_getenv` instead of `getenv` to obtain the value of critical environment variables. `secure_getenv` will pretend the variable has not bee set if the process environment is not trusted.

Critical environment variables are debugging flags, configuration file locations, plug-in and log file locations, and anything else that might be used to bypass security restrictions or cause a privileged process to behave in an unexpected way.

Either the `secure_getenv` function or the `__secure_getenv` is available from GNU libc.

Example 8.1. Obtaining a definition for `secure_getenv`

```
#include <stdlib.h>

#ifndef HAVE_SECURE_GETENV
#  ifdef HAVE__SECURE_GETENV
#    define secure_getenv __secure_getenv
#  else
#    error neither secure_getenv nor __secure_getenv are available
#  endif
#endif
```

8.4. Daemons

Background processes providing system services (*daemons*) need to decouple themselves from the controlling terminal and the parent process environment:

- Fork.
- In the child process, call `setsid`. The parent process can simply exit (using `_exit`, to avoid running clean-up actions twice).
- In the child process, fork again. Processing continues in the child process. Again, the parent process should just exit.

- Replace the descriptors 0, 1, 2 with a descriptor for **`/dev/null`**. Logging should be redirected to **syslog**.

Older instructions for creating daemon processes recommended a call to **`umask(0)`**. This is risky because it often leads to world-writable files and directories, resulting in security vulnerabilities such as arbitrary process termination by untrusted local users, or log file truncation. If the *umask* needs setting, a restrictive value such as **`027`** or **`077`** is recommended.

Other aspects of the process environment may have to changed as well (environment variables, signal handler disposition).

It is increasingly common that server processes do not run as background processes, but as regular foreground process under a supervising master process (such as **systemd**). Server processes should offer a command line option which disables forking and replacement of the standard output and standard error streams. Such an option is also useful for debugging.

8.5. Semantics of command line arguments

After process creation and option processing, it is up to the child process to interpret the arguments. Arguments can be file names, host names, or URLs, and many other things. URLs can refer to the local network, some server on the Internet, or to the local file system. Some applications even accept arbitrary code in arguments (for example, **python** with the **`-c`** option).

Similar concerns apply to environment variables, the contents of the current directory and its subdirectories.

Consequently, careful analysis is required if it is safe to pass untrusted data to another program.

8.6. `fork` as a primitive for parallelism

A call to `fork` which is not immediately followed by a call to `execve` (perhaps after rearranging and closing file descriptors) is typically unsafe, especially from a library which does not control the state of the entire process. Such use of `fork` should be replaced with proper child processes or threads.

Serialization and Deserialization

Protocol decoders and file format parsers are often the most-exposed part of an application because they are exposed with little or no user interaction and before any authentication and security checks are made. They are also difficult to write robustly in languages which are not memory-safe.

9.1. Recommendations for manually written decoders

For C and C++, the advice in *Section 1.1.2, "Recommendations for pointers and array handling"* applies. In addition, avoid non-character pointers directly into input buffers. Pointer misalignment causes crashes on some architectures.

When reading variable-sized objects, do not allocate large amounts of data solely based on the value of a size field. If possible, grow the data structure as more data is read from the source, and stop when no data is available. This helps to avoid denial-of-service attacks where little amounts of input data results in enormous memory allocations during decoding. Alternatively, you can impose reasonable bounds on memory allocations, but some protocols do not permit this.

9.2. Protocol design

Binary formats with explicit length fields are more difficult to parse robustly than those where the length of dynamically-sized elements is derived from sentinel values. A protocol which does not use length fields and can be written in printable ASCII characters simplifies testing and debugging. However, binary protocols with length fields may be more efficient to parse.

9.3. Library support for deserialization

For some languages, generic libraries are available which allow to serialize and deserialize user-defined objects. The deserialization part comes in one of two flavors, depending on the library. The first kind uses type information in the data stream to control which objects are instantiated. The second kind uses type definitions supplied by the programmer. The first one allows arbitrary object instantiation, the second one generally does not.

The following serialization frameworks are in the first category, are known to be unsafe, and must not be used for untrusted data:

- Python's *pickle* and *cPickle* modules
- Perl's *Storable* package
- Java serialization (java.io.ObjectInputStream)
- PHP serialization (`unserialize`)
- Most implementations of YAML

When using a type-directed deserialization format where the types of the deserialized objects are specified by the programmer, make sure that the objects which can be instantiated cannot perform any destructive actions in their destructors, even when the data members have been manipulated.

JSON decoders do not suffer from this problem. But you must not use the `eval` function to parse JSON objects in Javascript; even with the regular expression filter from RFC 4627, there are still information leaks remaining.

9.4. XML serialization

9.4.1. External references

XML documents can contain external references. They can occur in various places.

- In the DTD declaration in the header of an XML document:

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

- In a namespace declaration:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

- In an entity defintion:

```
<!ENTITY sys SYSTEM "http://www.example.com/ent.xml">
<!ENTITY pub PUBLIC "-//Example//Public Entity//EN"
  "http://www.example.com/pub-ent.xml">
```

- In a notation:

```
<!NOTATION not SYSTEM "../not.xml">
```

Originally, these external references were intended as unique identifiers, but by many XML implementations, they are used for locating the data for the referenced element. This causes unwanted network traffic, and may disclose file system contents or otherwise unreachable network resources, so this functionality should be disabled.

Depending on the XML library, external referenced might be processed not just when parsing XML, but also when generating it.

9.4.2. Entity expansion

When external DTD processing is disabled, an internal DTD subset can still contain entity definitions. Entity declarations can reference other entities. Some XML libraries expand entities automatically, and this processing cannot be switched off in some places (such as attribute values or content models). Without limits on the entity nesting level, this expansion results in data which can grow exponentially in length with size of the input. (If there is a limit on the nesting level, the growth is still polynomial, unless further limits are imposed.)

Consequently, the processing internal DTD subsets should be disabled if possible, and only trusted DTDs should be processed. If a particular XML application does not permit such restrictions, then application-specific limits are called for.

9.4.3. XInclude processing

XInclude processing can reference file and network resources and include them into the document, much like external entity references. When parsing untrusted XML documents, XInclude processing should be truned off.

XInclude processing is also fairly complex and may pull in support for the XPointer and XPath specifications, considerably increasing the amount of code required for XML processing.

9.4.4. Algorithmic complexity of XML validation

DTD-based XML validation uses regular expressions for content models. The XML specification requires that content models are deterministic, which means that efficient validation is possible. However, some implementations do not enforce determinism, and require exponential (or just polynomial) amount of space or time for validating some DTD/document combinations.

XML schemas and RELAX NG (via the **`xsd:`** prefix) directly support textual regular expressions which are not required to be deterministic.

9.4.5. Using Expat for XML parsing

By default, Expat does not try to resolve external IDs, so no steps are required to block them. However, internal entity declarations are processed. Installing a callback which stops parsing as soon as such entities are encountered disables them, see *Example 9.1, "Disabling XML entity processing with Expat"*. Expat does not perform any validation, so there are no problems related to that.

Example 9.1. Disabling XML entity processing with Expat

```
// Stop the parser when an entity declaration is encountered.
static void
EntityDeclHandler(void *userData,
    const XML_Char *entityName, int is_parameter_entity,
    const XML_Char *value, int value_length,
    const XML_Char *base, const XML_Char *systemId,
    const XML_Char *publicId, const XML_Char *notationName)
{
  XML_StopParser((XML_Parser)userData, XML_FALSE);
}
```

This handler must be installed when the **`XML_Parser`** object is created (*Example 9.2, "Creating an Expat XML parser"*).

Example 9.2. Creating an Expat XML parser

```
XML_Parser parser = XML_ParserCreate("UTF-8");
if (parser == NULL) {
  fprintf(stderr, "XML_ParserCreate failed\n");
  close(fd);
  exit(1);
}
// EntityDeclHandler needs a reference to the parser to stop
// parsing.
XML_SetUserData(parser, parser);
// Disable entity processing, to inhibit entity expansion.
XML_SetEntityDeclHandler(parser, EntityDeclHandler);
```

It is also possible to reject internal DTD subsets altogeher, using a suitable **`XML_StartDoctypeDeclHandler`** handler installed with `XML_SetDoctypeDeclHandler`.

9.4.6. Using OpenJDK for XML parsing and validation

OpenJDK contains facilities for DOM-based, SAX-based, and StAX-based document parsing. Documents can be validated against DTDs or XML schemas.

The approach taken to deal with entity expansion differs from the general recommendation in *Section 9.4.2, "Entity expansion"*. We enable the the feature flag **`javax.xml.XMLConstants.FEATURE_SECURE_PROCESSING`**, which enforces heuristic restrictions on the number of entity expansions. Note that this flag alone does not prevent resolution of external references (system IDs or public IDs), so it is slightly misnamed.

In the following sections, we use helper classes to prevent external ID resolution.

Example 9.3. Helper class to prevent DTD external entity resolution in OpenJDK

```
class NoEntityResolver implements EntityResolver {
    @Override
    public InputSource resolveEntity(String publicId, String systemId)
            throws SAXException, IOException {
        // Throwing an exception stops validation.
        throw new IOException(String.format(
                "attempt to resolve \"%s\" \"%s\"", publicId, systemId));
    }
}
```

Example 9.4. Helper class to prevent schema resolution in OpenJDK

```
class NoResourceResolver implements LSResourceResolver {
    @Override
    public LSInput resolveResource(String type, String namespaceURI,
            String publicId, String systemId, String baseURI) {
        // Throwing an exception stops validation.
        throw new RuntimeException(String.format(
                "resolution attempt: type=%s namespace=%s " +
                "publicId=%s systemId=%s baseURI=%s",
                type, namespaceURI, publicId, systemId, baseURI));
    }
}
```

Example 9.5, "Java imports for OpenJDK XML parsing" shows the imports used by the examples.

Example 9.5. Java imports for OpenJDK XML parsing

```
import javax.xml.XMLConstants;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.sax.SAXSource;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;
import javax.xml.validation.Validator;

import org.w3c.dom.Document;
```

```
import org.w3c.dom.ls.LSInput;
import org.w3c.dom.ls.LSResourceResolver;
import org.xml.sax.EntityResolver;
import org.xml.sax.ErrorHandler;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import org.xml.sax.XMLReader;
```

9.4.6.1. DOM-based XML parsing and DTD validation in OpenJDK

This approach produces a **`org.w3c.dom.Document`** object from an input stream. *Example 9.6, "DOM-based XML parsing in OpenJDK"* use the data from the **`java.io.InputStream`** instance in the **`inputStream`** variable.

Example 9.6. DOM-based XML parsing in OpenJDK

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
// Impose restrictions on the complexity of the DTD.
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

// Turn on validation.
// This step can be omitted if validation is not desired.
factory.setValidating(true);

// Parse the document.
DocumentBuilder builder = factory.newDocumentBuilder();
builder.setEntityResolver(new NoEntityResolver());
builder.setErrorHandler(new Errors());
Document document = builder.parse(inputStream);
```

External entity references are prohibited using the **`NoEntityResolver`** class in *Example 9.3, "Helper class to prevent DTD external entity resolution in OpenJDK"*. Because external DTD references are prohibited, DTD validation (if enabled) will only happen against the internal DTD subset embedded in the XML document.

To validate the document against an external DTD, use a **`javax.xml.transform.Transformer`** class to add the DTD reference to the document, and an entity resolver which whitelists this external reference.

9.4.6.2. XML Schema validation in OpenJDK

Example 9.7, "SAX-based validation against an XML schema in OpenJDK" shows how to validate a document against an XML Schema, using a SAX-based approach. The XML data is read from an **`java.io.InputStream`** in the **`inputStream`** variable.

Example 9.7. SAX-based validation against an XML schema in OpenJDK

```
SchemaFactory factory = SchemaFactory.newInstance(
        XMLConstants.W3C_XML_SCHEMA_NS_URI);

// This enables restrictions on the schema and document
// complexity.
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

// This prevents resource resolution by the schema itself.
// If the schema is trusted and references additional files,
```

```
// this line must be omitted, otherwise loading these files
// will fail.
factory.setResourceResolver(new NoResourceResolver());

Schema schema = factory.newSchema(schemaFile);
Validator validator = schema.newValidator();

// This prevents external resource resolution.
validator.setResourceResolver(new NoResourceResolver());

validator.validate(new SAXSource(new InputSource(inputStream)));
```

The **`NoResourceResolver`** class is defined in *Example 9.4, "Helper class to prevent schema resolution in OpenJDK"*.

If you need to validate a document against an XML schema, use the code in *Example 9.6, "DOM-based XML parsing in OpenJDK"* to create the document, but do not enable validation at this point. Then use *Example 9.8, "Validation of a DOM document against an XML schema in OpenJDK"* to perform the schema-based validation on the **`org.w3c.dom.Document`** instance **`document`**.

Example 9.8. Validation of a DOM document against an XML schema in OpenJDK

```
SchemaFactory factory = SchemaFactory.newInstance(
        XMLConstants.W3C_XML_SCHEMA_NS_URI);

// This enables restrictions on schema complexity.
factory.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, true);

// The following line prevents resource resolution
// by the schema itself.
factory.setResourceResolver(new NoResourceResolver());

Schema schema = factory.newSchema(schemaFile);

Validator validator = schema.newValidator();

// This prevents external resource resolution.
validator.setResourceResolver(new NoResourceResolver());
validator.validate(new DOMSource(document));
```

9.5. Protocol Encoders

For protocol encoders, you should write bytes to a buffer which grows as needed, using an exponential sizing policy. Explicit lengths can be patched in later, once they are known. Allocating the required number of bytes upfront typically requires separate code to compute the final size, which must be kept in sync with the actual encoding step, or vulnerabilities may result. In multi-threaded code, parts of the object being deserialized might change, so that the computed size is out of date.

You should avoid copying data directly from a received packet during encoding, disregarding the format. Propagating malformed data could enable attacks on other recipients of that data.

When using C or C++ and copying whole data structures directly into the output, make sure that you do not leak information in padding bytes between fields or at the end of the **`struct`**.

Cryptography

10.1. Primitives

Chosing from the following cryptographic primitives is recommended:

- RSA with 2048 bit keys and OAEP
- AES-128 in CBC mode
- SHA-256
- HMAC-SHA-256
- HMAC-SHA-1

Other cryptographic algorithms can be used if they are required for interoperability with existing software:

- RSA with key sizes larger than 1024 and legacy padding
- AES-192
- AES-256
- 3DES (triple DES, with two or three 56 bit keys)
- RC4 (but very, very strongly discouraged)
- SHA-1
- HMAC-MD5

Important

These primitives are difficult to use in a secure way. Custom implementation of security protocols should be avoided. For protecting confidentiality and integrity of network transmissions, TLS should be used (*Chapter 12, Transport Layer Security*).

10.2. Randomness

The following facilities can be used to generate unpredictable and non-repeating values. When these functions are used without special safeguards, each individual rnadom value should be at least 12 bytes long.

- `PK11_GenerateRandom` in the NSS library (usable for high data rates)
- `RAND_bytes` in the OpenSSL library (usable for high data rates)
- `gnutls_rnd` in GNUTLS, with **`GNUTLS_RND_RANDOM`** as the first argument (usable for high data rates)
- java.security.SecureRandom in Java (usable for high data rates)

- `os.urandom` in Python
- Reading from the **`/dev/urandom`** character device

All these functions should be non-blocking, and they should not wait until physical randomness becomes available. (Some cryptography providers for Java can cause java.security.SecureRandom to block, however.) Those functions which do not obtain all bits directly from **`/dev/urandom`** are suitable for high data rates because they do not deplete the system-wide entropy pool.

Difficult to use API

Both `RAND_bytes` and `PK11_GenerateRandom` have three-state return values (with conflicting meanings). Careful error checking is required. Please review the documentation when using these functions.

Other sources of randomness should be considered predictable.

Generating randomness for cryptographic keys in long-term use may need different steps and is best left to cryptographic libraries.

Part III. Implementing Security Features

Authentication and Authorization

11.1. Authenticating servers

When connecting to a server, a client has to make sure that it is actually talking to the server it expects. There are two different aspects, securing the network path, and making sure that the expected user runs the process on the target host. There are several ways to ensure that:

- The server uses a TLS certificate which is valid according to the web browser public key infrastructure, and the client verifies the certificate and the host name.
- The server uses a TLS certificate which is expectedby the client (perhaps it is stored in a configuration file read by the client). In this case, no host name checking is required.
- On Linux, UNIX domain sockets (of the **`PF_UNIX`** protocol family, sometimes called **`PF_LOCAL`**) are restricted by file system permissions. If the server socket path is not world-writable, the server identity cannot be spoofed by local users.
- Port numbers less than 1024 (*trusted ports*) can only be used by **`root`**, so if a UDP or TCP server is running on the local host and it uses a trusted port, its identity is assured. (Not all operating systems enforce the trusted ports concept, and the network might not be trusted, so it is only useful on the local system.)

TLS (*Chapter 12, Transport Layer Security*) is the recommended way for securing connections over untrusted networks.

If the server port number is 1024 is higher, a local user can impersonate the process by binding to this socket, perhaps after crashing the real server by exploiting a denial-of-service vulnerability.

11.2. Host-based authentication

Host-based authentication uses access control lists (ACLs) to accept or deny requests from clients. Thsis authentication method comes in two flavors: IP-based (or, more generally, address-based) and name-based (with the name coming from DNS or **`/etc/hosts`**). IP-based ACLs often use prefix notation to extend access to entire subnets. Name-based ACLs sometimes use wildcards for adding groups of hosts (from entire DNS subtrees). (In the SSH context, host-based authentication means something completely different and is not covered in this section.)

Host-based authentication trust the network and may not offer sufficient granularity, so it has to be considered a weak form of authentication. On the other hand, IP-based authentication can be made extremely robust and can be applied very early in input processing, so it offers an opportunity for significantly reducing the number of potential attackers for many services.

The names returned by `gethostbyaddr` and `getnameinfo` functions cannot be trusted. (DNS PTR records can be set to arbitrary values, not just names belong to the address owner.) If these names are used for ACL matching, a forward lookup using `gethostbyaddr` or `getaddrinfo` has to be performed. The name is only valid if the original address is found among the results of the forward lookup (*double-reverse lookup*).

An empty ACL should deny all access (deny-by-default). If empty ACLs permits all access, configuring any access list must switch to deny-by-default for all unconfigured protocols, in both name-based and address-based variants.

Similarly, if an address or name is not matched by the list, it should be denied. However, many implementations behave differently, so the actual behavior must be documented properly.

IPv6 addresses can embed IPv4 addresses. There is no universally correct way to deal with this ambiguity. The behavior of the ACL implementation should be documented.

11.3. UNIX domain socket authentication

UNIX domain sockets (with address family **`AF_UNIX`** or **`AF_LOCAL`**) are restricted to the local host and offer a special authentication mechanism: credentials passing.

Nowadays, most systems support the **`SO_PEERCRED`** (Linux) or **`LOCAL_PEERCRED`** (FreeBSD) socket options, or the `getpeereid` (other BSDs, MacOS X). These interfaces provide direct access to the (effective) user ID on the other end of a domain socket connect, without cooperation from the other end.

Historically, credentials passing was implemented using ancillary data in the `sendmsg` and `recvmsg` functions. On some systems, only credentials data that the peer has explicitly sent can be received, and the kernel checks the data for correctness on the sending side. This means that both peers need to deal with ancillary data. Compared to that, the modern interfaces are easier to use. Both sets of interfaces vary considerably among UNIX-like systems, unfortunately.

If you want to authenticate based on supplementary groups, you should obtain the user ID using one of these methods, and look up the list of supplementary groups using `getpwuid` (or `getpwuid_r`) and `getgrouplist`. Using the PID and information from **`/proc/PID/status`** is prone to race conditions and insecure.

11.4. `AF_NETLINK` authentication of origin

Netlink messages are used as a high-performance data transfer mechanism between the kernel and the userspace. Traditionally, they are used to exchange information related to the network statck, such as routing table entries.

When processing Netlink messages from the kernel, it is important to check that these messages actually originate from the kernel, by checking that the port ID (or PID) field **`nl_pid`** in the **`sockaddr_nl`** structure is **`0`**. (This structure can be obtained using `recvfrom` or `recvmsg`, it is different from the **`nlmsghdr`** structure.) The kernel does not prevent other processes from sending unicast Netlink messages, but the **`nl_pid`** field in the sender's socket address will be non-zero in such cases.

Applications should not use **`AF_NETLINK`** sockets as an IPC mechanism among processes, but prefer UNIX domain sockets for this tasks.

Transport Layer Security

Transport Layer Security (TLS, formerly Secure Sockets Layer/SSL) is the recommended way to to protect integrity and confidentiality while data is transferred over an untrusted network connection, and to identify the endpoint.

12.1. Common Pitfalls

TLS implementations are difficult to use, and most of them lack a clean API design. The following sections contain implementation-specific advice, and some generic pitfalls are mentioned below.

- Most TLS implementations have questionable default TLS cipher suites. Most of them enable anonymous Diffie-Hellman key exchange (but we generally want servers to authenticate themselves). Many do not disable ciphers which are subject to brute-force attacks because of restricted key lengths. Some even disable all variants of AES in the default configuration.

  When overriding the cipher suite defaults, it is recommended to disable all cipher suites which are not present on a whitelist, instead of simply enabling a list of cipher suites. This way, if an algorithm is disabled by default in the TLS implementation in a future security update, the application will not re-enable it.

- The name which is used in certificate validation must match the name provided by the user or configuration file. No host name canonicalization or IP address lookup must be performed.

- The TLS handshake has very poor performance if the TCP Nagle algorithm is active. You should switch on the **`TCP_NODELAY`** socket option (at least for the duration of the handshake), or use the Linux-specific **`TCP_CORK`** option.

  Example 12.1. Deactivating the TCP Nagle algorithm

  ```
  const int val = 1;
  int ret = setsockopt(sockfd, IPPROTO_TCP, TCP_NODELAY, &val, sizeof(val));
  if (ret < 0) {
    perror("setsockopt(TCP_NODELAY)");
    exit(1);
  }
  ```

- Implementing proper session resumption decreases handshake overhead considerably. This is important if the upper-layer protocol uses short-lived connections (like most application of HTTPS).

- Both client and server should work towards an orderly connection shutdown, that is send **`close_notify`** alerts and respond to them. This is especially important if the upper-layer protocol does not provide means to detect connection truncation (like some uses of HTTP).

- When implementing a server using event-driven programming, it is important to handle the TLS handshake properly because it includes multiple network round-trips which can block when an ordinary TCP `accept` would not. Otherwise, a client which fails to complete the TLS handshake for some reason will prevent the server from handling input from other clients.

- Unlike regular file descriptors, TLS connections cannot be passed between processes. Some TLS implementations add additional restrictions, and TLS connections generally cannot be used across `fork` function calls (see *Section 8.6, “`fork` as a primitive for parallelism”*).

12.1.1. OpenSSL Pitfalls

Some OpenSSL function use *tri-state return values*. Correct error checking is extremely important. Several functions return **`int`** values with the following meaning:

- The value **`1`** indicates success (for example, a successful signature verification).
- The value **`0`** indicates semantic failure (for example, a signature verification which was unsuccessful because the signing certificate was self-signed).
- The value **`-1`** indicates a low-level error in the system, such as failure to allocate memory using `malloc`.

Treating such tri-state return values as booleans can lead to security vulnerabilities. Note that some OpenSSL functions return boolean results or yet another set of status indicators. Each function needs to be checked individually.

Recovering precise error information is difficult. *Example 12.2, "Obtaining OpenSSL error codes"* shows how to obtain a more precise error code after a function call on an **SSL** object has failed. However, there are still cases where no detailed error information is available (e.g., if `SSL_shutdown` fails due to a connection teardown by the other end).

Example 12.2. Obtaining OpenSSL error codes

```
static void __attribute__((noreturn))
ssl_print_error_and_exit(SSL *ssl, const char *op, int ret)
{
  int subcode = SSL_get_error(ssl, ret);
  switch (subcode) {
  case SSL_ERROR_NONE:
    fprintf(stderr, "error: %s: no error to report\n", op);
    break;
  case SSL_ERROR_WANT_READ:
  case SSL_ERROR_WANT_WRITE:
  case SSL_ERROR_WANT_X509_LOOKUP:
  case SSL_ERROR_WANT_CONNECT:
  case SSL_ERROR_WANT_ACCEPT:
    fprintf(stderr, "error: %s: invalid blocking state %d\n", op, subcode);
    break;
  case SSL_ERROR_SSL:
    fprintf(stderr, "error: %s: TLS layer problem\n", op);
  case SSL_ERROR_SYSCALL:
    fprintf(stderr, "error: %s: system call failed: %s\n", op, strerror(errno));
    break;
  case SSL_ERROR_ZERO_RETURN:
    fprintf(stderr, "error: %s: zero return\n", op);
  }
  exit(1);
}
```

The `OPENSSL_config` function is documented to never fail. In reality, it can terminate the entire process if there is a failure accessing the configuration file. An error message is written to standard error, but which might not be visible if the function is called from a daemon process.

OpenSSL contains two separate ASN.1 DER decoders. One set of decoders operate on BIO handles (the input/output stream abstraction provided by OpenSSL); their decoder function names start with **`d2i_`** and end in **`_fp`** or **`_bio`** (e.g., `d2i_X509_fp` or `d2i_X509_bio`). These decoders must not be used for parsing data from untrusted sources; instead, the variants without the **`_fp`** and **`_bio`** (e.g.,

`d2i_X509`) shall be used. The BIO variants have received considerably less testing and are not very robust.

For the same reason, the OpenSSL command line tools (such as **`openssl x509`**) are generally generally less robust than the actual library code. They use the BIO functions internally, and not the more robust variants.

The command line tools do not always indicate failure in the exit status of the **openssl** process. For instance, a verification failure in **`openssl verify`** result in an exit status of zero.

The OpenSSL server and client applications (**`openssl s_client`** and **`openssl s_server`**) are debugging tools and should *never* be used as generic clients. For instance, the **`s_client`** tool reacts in a surprisign way to lines starting with **`R`** and **`Q`**.

OpenSSL allows application code to access private key material over documented interfaces. This can significantly increase the part of the code base which has to undergo security certification.

12.1.2. GNUTLS Pitfalls

`libgnutls.so.26` links to **`libpthread.so.0`**. Loading the threading library too late causes problems, so the main program should be linked with **`-lpthread`** as well. As a result, it can be difficult to use GNUTLS in a plugin which is loaded with the `dlopen` function. Another side effect is that applications which merely link against GNUTLS (even without actually using it) may incur a substantial overhead because other libraries automatically switch to thread-safe algorithms.

The `gnutls_global_init` function must be called before using any functionality provided by the library. This function is not thread-safe, so external locking is required, but it is not clear which lock should be used. Omitting the synchronization does not just lead to a memory leak, as it is suggested in the GNUTLS documentation, but to undefined behavior because there is no barrier that would enforce memory ordering.

The `gnutls_global_deinit` function does not actually deallocate all resources allocated by `gnutls_global_init`. It is currently not thread-safe. Therefore, it is best to avoid calling it altogether.

The X.509 implementation in GNUTLS is rather lenient. For example, it is possible to create and process X.509 version 1 certificates which carry extensions. These certificates are (correctly) rejected by other implementations.

12.1.3. OpenJDK Pitfalls

The Java cryptographic framework is highly modular. As a result, when you request an object implementing some cryptographic functionality, you cannot be completely sure that you end up with the well-tested, reviewed implementation in OpenJDK.

OpenJDK (in the source code as published by Oracle) and other implementations of the Java platform require that the system administrator has installed so-called *unlimited strength jurisdiction policy files*. Without this step, it is not possible to use the secure algorithms which offer sufficient cryptographic strength. Most downstream redistributors of OpenJDK remove this requirement.

Some versions of OpenJDK use **`/dev/random`** as the randomness source for nonces and other random data which is needed for TLS operation, but does not actually require physical randomness. As a result, TLS applications can block, waiting for more bits to become available in **`/dev/random`**.

12.1.4. NSS Pitfalls

NSS was not designed to be used by other libraries which can be linked into applications without modifying them. There is a lot of global state. There does not seem to be a way to perform required NSS initialization without race conditions.

If the NSPR descriptor is in an unexpected state, the `SSL_ForceHandshake` function can succeed, but no TLS handshake takes place, the peer is not authenticated, and subsequent data is exchanged in the clear.

NSS disables itself if it detects that the process underwent a `fork` after the library has been initialized. This behavior is required by the PKCS#11 API specification.

12.2. TLS Clients

Secure use of TLS in a client generally involves all of the following steps. (Individual instructions for specific TLS implementations follow in the next sections.)

- The client must configure the TLS library to use a set of trusted root certificates. These certificates are provided by the system in **`/etc/ssl/certs`** or files derived from it.
- The client selects sufficiently strong cryptographic primitives and disables insecure ones (such as no-op encryption). Compression and SSL version 2 support must be disabled (including the SSLv2-compatible handshake).
- The client initiates the TLS connection. The Server Name Indication extension should be used if supported by the TLS implementation. Before switching to the encrypted connection state, the contents of all input and output buffers must be discarded.
- The client needs to validate the peer certificate provided by the server, that is, the client must check that there is a cryptographically protected chain from a trusted root certificate to the peer certificate. (Depending on the TLS implementation, a TLS handshake can succeed even if the certificate cannot be validated.)
- The client must check that the configured or user-provided server name matches the peer certificate provided by the server.

It is safe to provide users detailed diagnostics on certificate validation failures. Other causes of handshake failures and, generally speaking, any details on other errors reported by the TLS implementation (particularly exception tracebacks), must not be divulged in ways that make them accessible to potential attackers. Otherwise, it is possible to create decryption oracles.

Important

Depending on the application, revocation checking (against certificate revocations lists or via OCSP) and session resumption are important aspects of production-quality client. These aspects are not yet covered.

12.2.1. Implementation TLS Clients With OpenSSL

In the following code, the error handling is only exploratory. Proper error handling is required for production use, especially in libraries.

The OpenSSL library needs explicit initialization (see *Example 12.3, “OpenSSL library initialization”*).

Example 12.3. OpenSSL library initialization

```
// The following call prints an error message and calls exit() if
// the OpenSSL configuration file is unreadable.
OPENSSL_config(NULL);
// Provide human-readable error messages.
SSL_load_error_strings();
// Register ciphers.
SSL_library_init();
```

After that, a context object has to be created, which acts as a factory for connection objects (*Example 12.4, "OpenSSL client context creation"*). We use an explicit cipher list so that we do not pick up any strange ciphers when OpenSSL is upgraded. The actual version requested in the client hello depends on additional restrictions in the OpenSSL library. If possible, you should follow the example code and use the default list of trusted root certificate authorities provided by the system because you would have to maintain your own set otherwise, which can be cumbersome.

Example 12.4. OpenSSL client context creation

```
// Configure a client connection context.  Send a hendshake for the
// highest supported TLS version, and disable compression.
const SSL_METHOD *const req_method = SSLv23_client_method();
SSL_CTX *const ctx = SSL_CTX_new(req_method);
if (ctx == NULL) {
  ERR_print_errors(bio_err);
  exit(1);
}
SSL_CTX_set_options(ctx, SSL_OP_NO_SSLv2 | SSL_OP_NO_COMPRESSION);

// Adjust the ciphers list based on a whitelist.  First enable all
// ciphers of at least medium strength, to get the list which is
// compiled into OpenSSL.
if (SSL_CTX_set_cipher_list(ctx, "HIGH:MEDIUM") != 1) {
  ERR_print_errors(bio_err);
  exit(1);
}
{
  // Create a dummy SSL session to obtain the cipher list.
  SSL *ssl = SSL_new(ctx);
  if (ssl == NULL) {
    ERR_print_errors(bio_err);
    exit(1);
  }
  STACK_OF(SSL_CIPHER) *active_ciphers = SSL_get_ciphers(ssl);
  if (active_ciphers == NULL) {
    ERR_print_errors(bio_err);
    exit(1);
  }
  // Whitelist of candidate ciphers.
  static const char *const candidates[] =  {
    "AES128-GCM-SHA256", "AES128-SHA256", "AES256-SHA256", // strong ciphers
    "AES128-SHA", "AES256-SHA", // strong ciphers, also in older versions
    "RC4-SHA", "RC4-MD5", // backwards compatibility, supposed to be weak
    "DES-CBC3-SHA", "DES-CBC3-MD5", // more backwards compatibility
    NULL
  };
  // Actually selected ciphers.
  char ciphers[300];
  ciphers[0] = '\0';
```

```
  for (const char *const *c = candidates; *c; ++c) {
    for (int i = 0; i < sk_SSL_CIPHER_num(active_ciphers); ++i) {
 if (strcmp(SSL_CIPHER_get_name(sk_SSL_CIPHER_value(active_ciphers, i)),
     *c) == 0) {
   if (*ciphers) {
     strcat(ciphers, ":");
   }
   strcat(ciphers, *c);
   break;
 }
    }
  }
  SSL_free(ssl);
  // Apply final cipher list.
  if (SSL_CTX_set_cipher_list(ctx, ciphers) != 1) {
    ERR_print_errors(bio_err);
    exit(1);
  }
}

// Load the set of trusted root certificates.
if (!SSL_CTX_set_default_verify_paths(ctx)) {
  ERR_print_errors(bio_err);
  exit(1);
}
```

A single context object can be used to create multiple connection objects. It is safe to use the same **SSL_CTX** object for creating connections concurrently from multiple threads, provided that the **SSL_CTX** object is not modified (e.g., callbacks must not be changed).

After creating the TCP socket and disabling the Nagle algorithm (per *Example 12.1, "Deactivating the TCP Nagle algorithm"*), the actual connection object needs to be created, as show in *Example 12.4, "OpenSSL client context creation"*. If the handshake started by `SSL_connect` fails, the `ssl_print_error_and_exit` function from *Example 12.2, "Obtaining OpenSSL error codes"* is called.

The `certificate_validity_override` function provides an opportunity to override the validity of the certificate in case the OpenSSL check fails. If such functionality is not required, the call can be removed, otherwise, the application developer has to implement it.

The host name passed to the functions `SSL_set_tlsext_host_name` and `X509_check_host` must be the name that was passed to `getaddrinfo` or a similar name resolution function. No host name canonicalization must be performed. The `X509_check_host` function used in the final step for host name matching is currently only implemented in OpenSSL 1.1, which is not released yet. In case host name matching fails, the function `certificate_host_name_override` is called. This function should check user-specific certificate store, to allow a connection even if the host name does not match the certificate. This function has to be provided by the application developer. Note that the override must be keyed by both the certificate *and* the host name.

Example 12.5. Creating a client connection using OpenSSL

```
// Create the connection object.
SSL *ssl = SSL_new(ctx);
if (ssl == NULL) {
  ERR_print_errors(bio_err);
  exit(1);
}
SSL_set_fd(ssl, sockfd);
```

```
// Enable the ServerNameIndication extension
if (!SSL_set_tlsext_host_name(ssl, host)) {
  ERR_print_errors(bio_err);
  exit(1);
}

// Perform the TLS handshake with the server.
ret = SSL_connect(ssl);
if (ret != 1) {
  // Error status can be 0 or negative.
  ssl_print_error_and_exit(ssl, "SSL_connect", ret);
}

// Obtain the server certificate.
X509 *peercert = SSL_get_peer_certificate(ssl);
if (peercert == NULL) {
  fprintf(stderr, "peer certificate missing");
  exit(1);
}

// Check the certificate verification result.  Allow an explicit
// certificate validation override in case verification fails.
int verifystatus = SSL_get_verify_result(ssl);
if (verifystatus != X509_V_OK && !certificate_validity_override(peercert)) {
  fprintf(stderr, "SSL_connect: verify result: %s\n",
     X509_verify_cert_error_string(verifystatus));
  exit(1);
}

// Check if the server certificate matches the host name used to
// establish the connection.
// FIXME: Currently needs OpenSSL 1.1.
if (X509_check_host(peercert, (const unsigned char *)host, strlen(host),
        0) != 1
    && !certificate_host_name_override(peercert, host)) {
  fprintf(stderr, "SSL certificate does not match host name\n");
  exit(1);
}

X509_free(peercert);
```

The connection object can be used for sending and receiving data, as in *Example 12.6, “Using an OpenSSL connection to send and receive data”*. It is also possible to create a **`BIO`** object and use the **`SSL`** object as the underlying transport, using `BIO_set_ssl`.

Example 12.6. Using an OpenSSL connection to send and receive data

```
const char *const req = "GET / HTTP/1.0\r\n\r\n";
if (SSL_write(ssl, req, strlen(req)) < 0) {
  ssl_print_error_and_exit(ssl, "SSL_write", ret);
}
char buf[4096];
ret = SSL_read(ssl, buf, sizeof(buf));
if (ret < 0) {
  ssl_print_error_and_exit(ssl, "SSL_read", ret);
}
```

When it is time to close the connection, the `SSL_shutdown` function needs to be called twice for an orderly, synchronous connection termination (*Example 12.7, “Closing an OpenSSL connection*

in an orderly fashion"). This exchanges **close_notify** alerts with the server. The additional logic is required to deal with an unexpected **close_notify** from the server. Note that is necessary to explicitly close the underlying socket after the connection object has been freed.

Example 12.7. Closing an OpenSSL connection in an orderly fashion

```
// Send the close_notify alert.
ret = SSL_shutdown(ssl);
switch (ret) {
case 1:
  // A close_notify alert has already been received.
  break;
case 0:
  // Wait for the close_notify alert from the peer.
  ret = SSL_shutdown(ssl);
  switch (ret) {
  case 0:
    fprintf(stderr, "info: second SSL_shutdown returned zero\n");
    break;
  case 1:
    break;
  default:
    ssl_print_error_and_exit(ssl, "SSL_shutdown 2", ret);
  }
  break;
default:
  ssl_print_error_and_exit(ssl, "SSL_shutdown 1", ret);
}
SSL_free(ssl);
close(sockfd);
```

Example 12.8, "Closing an OpenSSL connection in an orderly fashion" shows how to deallocate the context object when it is no longer needed because no further TLS connections will be established.

Example 12.8. Closing an OpenSSL connection in an orderly fashion

```
SSL_CTX_free(ctx);
```

12.2.2. Implementation TLS Clients With GNUTLS

This section describes how to implement a TLS client with full certificate validation (but without certificate revocation checking). Note that the error handling in is only exploratory and needs to be replaced before production use.

The GNUTLS library needs explicit initialization:

```
gnutls_global_init();
```

Failing to do so can result in obscure failures in Base64 decoding. See *Section 12.1.2, "GNUTLS Pitfalls"* for additional aspects of initialization.

Before setting up TLS connections, a credentials objects has to be allocated and initialized with the set of trusted root CAs (*Example 12.9, "Initializing a GNUTLS credentials structure"*).

Example 12.9. Initializing a GNUTLS credentials structure

```
// Load the trusted CA certificates.
gnutls_certificate_credentials_t cred = NULL;
int ret = gnutls_certificate_allocate_credentials (&cred);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_certificate_allocate_credentials: %s\n",
     gnutls_strerror(ret));
  exit(1);
}
// gnutls_certificate_set_x509_system_trust needs GNUTLS version 3.0
// or newer, so we hard-code the path to the certificate store
// instead.
static const char ca_bundle[] = "/etc/ssl/certs/ca-bundle.crt";
ret = gnutls_certificate_set_x509_trust_file
  (cred, ca_bundle, GNUTLS_X509_FMT_PEM);
if (ret == 0) {
  fprintf(stderr, "error: no certificates found in: %s\n", ca_bundle);
  exit(1);
}
if (ret < 0) {
  fprintf(stderr, "error: gnutls_certificate_set_x509_trust_files(%s): %s\n",
     ca_bundle, gnutls_strerror(ret));
  exit(1);
}
```

After the last TLS connection has been closed, this credentials object should be freed:

```
gnutls_certificate_free_credentials(cred);
```

During its lifetime, the credentials object can be used to initialize TLS session objects from multiple threads, provided that it is not changed.

Once the TCP connection has been established, the Nagle algorithm should be disabled (see *Example 12.1, “Deactivating the TCP Nagle algorithm”*). After that, the socket can be associated with a new GNUTLS session object. The previously allocated credentials object provides the set of root CAs. The **NORMAL** set of cipher suites and protocols provides a reasonable default. Then the TLS handshake must be initiated. This is shown in *Example 12.10, “Establishing a TLS client connection using GNUTLS”*.

Example 12.10. Establishing a TLS client connection using GNUTLS

```
// Create the session object.
gnutls_session_t session;
ret = gnutls_init(&session, GNUTLS_CLIENT);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_init: %s\n",
     gnutls_strerror(ret));
  exit(1);
}

// Configure the cipher preferences.
const char *errptr = NULL;
ret = gnutls_priority_set_direct(session, "NORMAL", &errptr);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_priority_set_direct: %s\n"
```

```
      "error: at: \"%s\"\n", gnutls_strerror(ret), errptr);
  exit(1);
}

// Install the trusted certificates.
ret = gnutls_credentials_set(session, GNUTLS_CRD_CERTIFICATE, cred);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_credentials_set: %s\n",
     gnutls_strerror(ret));
  exit(1);
}

// Associate the socket with the session object and set the server
// name.
gnutls_transport_set_ptr(session, (gnutls_transport_ptr_t)(uintptr_t)sockfd);
ret = gnutls_server_name_set(session, GNUTLS_NAME_DNS,
          host, strlen(host));
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_server_name_set: %s\n",
     gnutls_strerror(ret));
  exit(1);
}

// Establish the session.
ret = gnutls_handshake(session);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_handshake: %s\n",
     gnutls_strerror(ret));
  exit(1);
}
```

After the handshake has been completed, the server certificate needs to be verified (*Example 12.11, “Verifying a server certificate using GNUTLS”*). In the example, the user-defined `certificate_validity_override` function is called if the verification fails, so that a separate, user-specific trust store can be checked. This function call can be omitted if the functionality is not needed.

Example 12.11. Verifying a server certificate using GNUTLS

```

// Obtain the server certificate chain.  The server certificate
// itself is stored in the first element of the array.
unsigned certslen = 0;
const gnutls_datum_t *const certs =
  gnutls_certificate_get_peers(session, &certslen);
if (certs == NULL || certslen == 0) {
  fprintf(stderr, "error: could not obtain peer certificate\n");
  exit(1);
}

// Validate the certificate chain.
unsigned status = (unsigned)-1;
ret = gnutls_certificate_verify_peers2(session, &status);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_certificate_verify_peers2: %s\n",
     gnutls_strerror(ret));
  exit(1);
}
if (status != 0 && !certificate_validity_override(certs[0])) {
  gnutls_datum_t msg;
#if GNUTLS_VERSION_AT_LEAST_3_1_4
  int type = gnutls_certificate_type_get (session);
  ret = gnutls_certificate_verification_status_print(status, type, &out, 0);
```

```
#else
  ret = -1;
#endif
  if (ret == 0) {
    fprintf(stderr, "error: %s\n", msg.data);
    gnutls_free(msg.data);
    exit(1);
  } else {
    fprintf(stderr, "error: certificate validation failed with code 0x%x\n",
       status);
    exit(1);
  }
}
```

In the next step (*Example 12.12, "Matching the server host name and certificate in a GNUTLS client"*, the certificate must be matched against the host name (note the unusual return value from `gnutls_x509_crt_check_hostname`). Again, an override function `certificate_host_name_override` is called. Note that the override must be keyed to the certificate *and* the host name. The function call can be omitted if the override is not needed.

Example 12.12. Matching the server host name and certificate in a GNUTLS client

```
// Match the peer certificate against the host name.
// We can only obtain a set of DER-encoded certificates from the
// session object, so we have to re-parse the peer certificate into
// a certificate object.
gnutls_x509_crt_t cert;
ret = gnutls_x509_crt_init(&cert);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_x509_crt_init: %s\n",
     gnutls_strerror(ret));
  exit(1);
}
// The peer certificate is the first certificate in the list.
ret = gnutls_x509_crt_import(cert, certs, GNUTLS_X509_FMT_DER);
if (ret != GNUTLS_E_SUCCESS) {
  fprintf(stderr, "error: gnutls_x509_crt_import: %s\n",
     gnutls_strerror(ret));
  exit(1);
}
ret = gnutls_x509_crt_check_hostname(cert, host);
if (ret == 0 && !certificate_host_name_override(certs[0], host)) {
  fprintf(stderr, "error: host name does not match certificate\n");
  exit(1);
}
gnutls_x509_crt_deinit(cert);
```

In newer GNUTLS versions, certificate checking and host name validation can be combined using the `gnutls_certificate_verify_peers3` function.

An established TLS session can be used for sending and receiving data, as in *Example 12.13, "Using a GNUTLS session"*.

Example 12.13. Using a GNUTLS session

```
char buf[4096];
snprintf(buf, sizeof(buf), "GET / HTTP/1.0\r\nHost: %s\r\n\r\n", host);
ret = gnutls_record_send(session, buf, strlen(buf));
```

```
if (ret < 0) {
  fprintf(stderr, "error: gnutls_record_send: %s\n", gnutls_strerror(ret));
  exit(1);
}
ret = gnutls_record_recv(session, buf, sizeof(buf));
if (ret < 0) {
  fprintf(stderr, "error: gnutls_record_recv: %s\n", gnutls_strerror(ret));
  exit(1);
}
```

In order to shut down a connection in an orderly manner, you should call the `gnutls_bye` function. Finally, the session object can be deallocated using `gnutls_deinit` (see *Example 12.14, "Using a GNUTLS session"*).

Example 12.14. Using a GNUTLS session

```
// Initiate an orderly connection shutdown.
ret = gnutls_bye(session, GNUTLS_SHUT_RDWR);
if (ret < 0) {
  fprintf(stderr, "error: gnutls_bye: %s\n", gnutls_strerror(ret));
  exit(1);
}
// Free the session object.
gnutls_deinit(session);
```

12.2.3. Implementing TLS Clients With OpenJDK

The examples below use the following cryptographic-related classes:

```
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.security.cert.CertificateEncodingException;
import java.security.cert.CertificateException;
import java.security.cert.X509Certificate;
import javax.net.ssl.SSLContext;
import javax.net.ssl.SSLParameters;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.TrustManager;
import javax.net.ssl.X509TrustManager;

import sun.security.util.HostnameChecker;
```

If compatibility with OpenJDK 6 is required, it is necessary to use the internal class **`sun.security.util.HostnameChecker`**. (The public OpenJDK API does not provide any support for dissecting the subject distinguished name of an X.509 certificate, so a custom-written DER parser is needed—or we have to use an internal class, which we do below.) In OpenJDK 7, the `setEndpointIdentificationAlgorithm` method was added to the **`javax.net.ssl.SSLParameters`** class, providing an official way to implement host name checking.

TLS connections are established using an **`SSLContext`** instance. With a properly configured OpenJDK installation, the **SunJSSE** provider uses the system-wide set of trusted root certificate authorities, so no further configuration is necessary. For backwards compatibility with OpenJDK 6, the **TLSv1** provider has to be supported as a fall-back option. This is shown in *Example 12.15, "Setting up an **`SSLContext`** for OpenJDK TLS clients"*.

Example 12.15. Setting up an **`SSLContext`** for OpenJDK TLS clients

```
// Create the context.  Specify the SunJSSE provider to avoid
// picking up third-party providers.  Try the TLS 1.2 provider
// first, then fall back to TLS 1.0.
SSLContext ctx;
try {
    ctx = SSLContext.getInstance("TLSv1.2", "SunJSSE");
} catch (NoSuchAlgorithmException e) {
    try {
        ctx = SSLContext.getInstance("TLSv1", "SunJSSE");
    } catch (NoSuchAlgorithmException e1) {
        // The TLS 1.0 provider should always be available.
        throw new AssertionError(e1);
    } catch (NoSuchProviderException e1) {
        throw new AssertionError(e1);
    }
} catch (NoSuchProviderException e) {
    // The SunJSSE provider should always be available.
    throw new AssertionError(e);
}
ctx.init(null, null, null);
```

In addition to the context, a TLS parameter object will be needed which adjusts the cipher suites and protocols (*Example 12.16, "Setting up **`SSLParameters`** for TLS use with OpenJDK"*). Like the context, these parameters can be reused for multiple TLS connections.

Example 12.16. Setting up **`SSLParameters`** for TLS use with OpenJDK

```
// Prepare TLS parameters.  These have to applied to every TLS
// socket before the handshake is triggered.
SSLParameters params = ctx.getDefaultSSLParameters();
// Do not send an SSL-2.0-compatible Client Hello.
ArrayList<String> protocols = new ArrayList<String>(
    Arrays.asList(params.getProtocols()));
protocols.remove("SSLv2Hello");
params.setProtocols(protocols.toArray(new String[protocols.size()]));
// Adjust the supported ciphers.
ArrayList<String> ciphers = new ArrayList<String>(
    Arrays.asList(params.getCipherSuites()));
ciphers.retainAll(Arrays.asList(
    "TLS_RSA_WITH_AES_128_CBC_SHA256",
    "TLS_RSA_WITH_AES_256_CBC_SHA256",
    "TLS_RSA_WITH_AES_256_CBC_SHA",
    "TLS_RSA_WITH_AES_128_CBC_SHA",
    "SSL_RSA_WITH_3DES_EDE_CBC_SHA",
    "SSL_RSA_WITH_RC4_128_SHA1",
    "SSL_RSA_WITH_RC4_128_MD5",
    "TLS_EMPTY_RENEGOTIATION_INFO_SCSV"));
params.setCipherSuites(ciphers.toArray(new String[ciphers.size()]));
```

As initialized above, the parameter object does not yet require host name checking. This has to be enabled separately, and this is only supported by OpenJDK 7 and later:

```
params.setEndpointIdentificationAlgorithm("HTTPS");
```

All application protocols can use the **"HTTPS"** algorithm. (The algorithms have minor differences with regard to wildcard handling, which should not matter in practice.)

Example 12.17, "Establishing a TLS connection with OpenJDK" shows how to establish the connection. Before the handshake is initialized, the protocol and cipher configuration has to be performed, by applying the parameter object **params**. (After this point, changes to **params** will not affect this TLS socket.) As mentioned initially, host name checking requires using an internal API on OpenJDK 6.

Example 12.17. Establishing a TLS connection with OpenJDK

```
// Create the socket and connect it at the TCP layer.
SSLSocket socket = (SSLSocket) ctx.getSocketFactory()
    .createSocket(host, port);

// Disable the Nagle algorithm.
socket.setTcpNoDelay(true);

// Adjust ciphers and protocols.
socket.setSSLParameters(params);

// Perform the handshake.
socket.startHandshake();

// Validate the host name.  The match() method throws
// CertificateException on failure.
X509Certificate peer = (X509Certificate)
    socket.getSession().getPeerCertificates()[0];
// This is the only way to perform host name checking on OpenJDK 6.
HostnameChecker.getInstance(HostnameChecker.TYPE_TLS).match(
    host, peer);
```

Starting with OpenJDK 7, the last lines can be omitted, provided that host name verification has been enabled by calling the `setEndpointIdentificationAlgorithm` method on the **params** object (before it was applied to the socket).

The TLS socket can be used as a regular socket, as shown in *Example 12.18, "Using a TLS client socket in OpenJDK"*.

Example 12.18. Using a TLS client socket in OpenJDK

```
socket.getOutputStream().write("GET / HTTP/1.0\r\n\r\n"
    .getBytes(Charset.forName("UTF-8")));
byte[] buffer = new byte[4096];
int count = socket.getInputStream().read(buffer);
System.out.write(buffer, 0, count);
```

12.2.3.1. Overriding server certificate validation with OpenJDK 6

Overriding certificate validation requires a custom trust manager. With OpenJDK 6, the trust manager lacks information about the TLS session, and to which server the connection is made. Certificate overrides have to be tied to specific servers (host names). Consequently, different **TrustManager** and **SSLContext** objects have to be used for different servers.

In the trust manager shown in *Example 12.19, "A customer trust manager for OpenJDK TLS clients"*, the server certificate is identified by its SHA-256 hash.

Example 12.19. A customer trust manager for OpenJDK TLS clients

```
public class MyTrustManager implements X509TrustManager {
    private final byte[] certHash;

    public MyTrustManager(byte[] certHash) throws Exception {
        this.certHash = certHash;
    }

    @Override
    public void checkClientTrusted(X509Certificate[] chain, String authType)
            throws CertificateException {
        throw new UnsupportedOperationException();
    }

    @Override
    public void checkServerTrusted(X509Certificate[] chain,
            String authType) throws CertificateException {
        byte[] digest = getCertificateDigest(chain[0]);
        String digestHex = formatHex(digest);

        if (Arrays.equals(digest, certHash)) {
            System.err.println("info: accepting certificate: " + digestHex);
        } else {
            throw new CertificateException("certificate rejected: "  +
                    digestHex);
        }
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return new X509Certificate[0];
    }
}
```

This trust manager has to be passed to the **`init`** method of the **`SSLContext`** object, as show in *Example 12.20, “Using a custom TLS trust manager with OpenJDK”*.

Example 12.20. Using a custom TLS trust manager with OpenJDK

```
SSLContext ctx;
try {
    ctx = SSLContext.getInstance("TLSv1.2", "SunJSSE");
} catch (NoSuchAlgorithmException e) {
    try {
        ctx = SSLContext.getInstance("TLSv1", "SunJSSE");
    } catch (NoSuchAlgorithmException e1) {
        throw new AssertionError(e1);
    } catch (NoSuchProviderException e1) {
        throw new AssertionError(e1);
    }
} catch (NoSuchProviderException e) {
    throw new AssertionError(e);
}
MyTrustManager tm = new MyTrustManager(certHash);
ctx.init(null, new TrustManager[] {tm}, null);
```

When certificate overrides are in place, host name verification should not be performed because there is no security requirement that the host name in the certificate matches the host name used to establish the connection (and it often will not). However, without host name verification, it is not possible to perform transparent fallback to certification validation using the system certificate store.

The approach described above works with OpenJDK 6 and later versions. Starting with OpenJDK 7, it is possible to use a custom subclass of the **`javax.net.ssl.X509ExtendedTrustManager`** class. The OpenJDK TLS implementation will call the new methods, passing along TLS session information. This can be used to implement certificate overrides as a fallback (if certificate or host name verification fails), and a trust manager object can be used for multiple servers because the server address is available to the trust manager.

12.2.4. Implementing TLS Clients With NSS

The following code shows how to implement a simple TLS client using NSS. Note that the error handling needs replacing before production use.

Using NSS needs several header files, as shown in *Example 12.21, "Include files for NSS"*.

Example 12.21. Include files for NSS

```
// NSPR include files
#include <prerror.h>
#include <prinit.h>

// NSS include files
#include <nss.h>
#include <pk11pub.h>
#include <secmod.h>
#include <ssl.h>
#include <sslproto.h>

// Private API, no other way to turn a POSIX file descriptor into an
// NSPR handle.
NSPR_API(PRFileDesc*) PR_ImportTCPSocket(int);
```

Initializing the NSS library is a complex task (*Example 12.22, "Initializing the NSS library"*). It is not thread-safe. By default, the library is in export mode, and all strong ciphers are disabled. Therefore, after creating the **`NSSInitCContext`** object, we probe all the strong ciphers we want to use, and check if at least one of them is available. If not, we call `NSS_SetDomesticPolicy` to switch to unrestricted policy mode. This function replaces the existing global cipher suite policy, that is why we avoid calling it unless absolutely necessary.

The simplest way to configured the trusted root certificates involves loading the **`libnssckbi.so`** NSS module with a call to the `SECMOD_LoadUserModule` function. The root certificates are compiled into this module. (The PEM module for NSS, **`libnsspem.so`**, offers a way to load trusted CA certificates from a file.)

Example 12.22. Initializing the NSS library

```
PR_Init(PR_USER_THREAD, PR_PRIORITY_NORMAL, 0);
NSSInitContext *const ctx =
  NSS_InitContext("sql:/etc/pki/nssdb", "", "", "", NULL,
      NSS_INIT_READONLY | NSS_INIT_PK11RELOAD);
if (ctx == NULL) {
  const PRErrorCode err = PR_GetError();
```

```
  fprintf(stderr, "error: NSPR error code %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}

// Ciphers to enable.
static const PRUint16 good_ciphers[] = {
  TLS_RSA_WITH_AES_128_CBC_SHA,
  TLS_RSA_WITH_AES_256_CBC_SHA,
  SSL_RSA_WITH_3DES_EDE_CBC_SHA,
  SSL_NULL_WITH_NULL_NULL // sentinel
};

// Check if the current policy allows any strong ciphers.  If it
// doesn't, switch to the "domestic" (unrestricted) policy.  This is
// not thread-safe and has global impact.  Consequently, we only do
// it if absolutely necessary.
int found_good_cipher = 0;
for (const PRUint16 *p = good_ciphers; *p != SSL_NULL_WITH_NULL_NULL;
     ++p) {
  PRInt32 policy;
  if (SSL_CipherPolicyGet(*p, &policy) != SECSuccess) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: policy for cipher %u: error %d: %s\n",
       (unsigned)*p, err, PR_ErrorToName(err));
    exit(1);
  }
  if (policy == SSL_ALLOWED) {
    fprintf(stderr, "info: found cipher %x\n", (unsigned)*p);
    found_good_cipher = 1;
    break;
  }
}
if (!found_good_cipher) {
  if (NSS_SetDomesticPolicy() != SECSuccess) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: NSS_SetDomesticPolicy: error %d: %s\n",
       err, PR_ErrorToName(err));
    exit(1);
  }
}

// Initialize the trusted certificate store.
char module_name[] = "library=libnssckbi.so name=\"Root Certs\"";
SECMODModule *module = SECMOD_LoadUserModule(module_name, NULL, PR_FALSE);
if (module == NULL || !module->loaded) {
  const PRErrorCode err = PR_GetError();
  fprintf(stderr, "error: NSPR error code %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}
```

Some of the effects of the initialization can be reverted with the following function calls:

```
SECMOD_DestroyModule(module);
NSS_ShutdownContext(ctx);
```

After NSS has been initialized, the TLS connection can be created (*Example 12.23, "Creating a TLS connection with NSS"*). The internal `PR_ImportTCPSocket` function is used to turn the POSIX file descriptor **`sockfd`** into an NSPR file descriptor. (This function is de-facto part of the NSS public ABI, so it will not go away.) Creating the TLS-capable file descriptor requires a *model* descriptor, which is configured with the desired set of protocols and ciphers. (The **`good_ciphers`** variable is part of

Example 12.22, "Initializing the NSS library".) We cannot resort to disabling ciphers not on a whitelist because by default, the AES cipher suites are disabled. The model descriptor is not needed anymore after TLS support has been activated for the existing connection descriptor.

The call to `SSL_BadCertHook` can be omitted if no mechanism to override certificate verification is needed. The **`bad_certificate`** function must check both the host name specified for the connection and the certificate before granting the override.

Triggering the actual handshake requires three function calls, `SSL_ResetHandshake`, `SSL_SetURL`, and `SSL_ForceHandshake`. (If `SSL_ResetHandshake` is omitted, `SSL_ForceHandshake` will succeed, but the data will not be encrypted.) During the handshake, the certificate is verified and matched against the host name.

Example 12.23. Creating a TLS connection with NSS

```
// Wrap the POSIX file descriptor.  This is an internal NSPR
// function, but it is very unlikely to change.
PRFileDesc* nspr = PR_ImportTCPSocket(sockfd);
sockfd = -1; // Has been taken over by NSPR.

// Add the SSL layer.
{
  PRFileDesc *model = PR_NewTCPSocket();
  PRFileDesc *newfd = SSL_ImportFD(NULL, model);
  if (newfd == NULL) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: NSPR error code %d: %s\n",
       err, PR_ErrorToName(err));
    exit(1);
  }
  model = newfd;
  newfd = NULL;
  if (SSL_OptionSet(model, SSL_ENABLE_SSL2, PR_FALSE) != SECSuccess) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: set SSL_ENABLE_SSL2 error %d: %s\n",
       err, PR_ErrorToName(err));
    exit(1);
  }
  if (SSL_OptionSet(model, SSL_V2_COMPATIBLE_HELLO, PR_FALSE) != SECSuccess) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: set SSL_V2_COMPATIBLE_HELLO error %d: %s\n",
       err, PR_ErrorToName(err));
    exit(1);
  }
  if (SSL_OptionSet(model, SSL_ENABLE_DEFLATE, PR_FALSE) != SECSuccess) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: set SSL_ENABLE_DEFLATE error %d: %s\n",
       err, PR_ErrorToName(err));
    exit(1);
  }

  // Disable all ciphers (except RC4-based ciphers, for backwards
  // compatibility).
  const PRUint16 *const ciphers = SSL_GetImplementedCiphers();
  for (unsigned i = 0; i < SSL_GetNumImplementedCiphers(); i++) {
    if (ciphers[i] != SSL_RSA_WITH_RC4_128_SHA
   && ciphers[i] != SSL_RSA_WITH_RC4_128_MD5) {
 if (SSL_CipherPrefSet(model, ciphers[i], PR_FALSE) != SECSuccess) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: disable cipher %u: error %d: %s\n",
     (unsigned)ciphers[i], err, PR_ErrorToName(err));
    exit(1);
 }
```

```
    }
  }

  // Enable the strong ciphers.
  for (const PRUint16 *p = good_ciphers; *p != SSL_NULL_WITH_NULL_NULL;
  ++p) {
    if (SSL_CipherPrefSet(model, *p, PR_TRUE) != SECSuccess) {
 const PRErrorCode err = PR_GetError();
 fprintf(stderr, "error: enable cipher %u: error %d: %s\n",
  (unsigned)*p, err, PR_ErrorToName(err));
 exit(1);
    }
  }

  // Allow overriding invalid certificate.
  if (SSL_BadCertHook(model, bad_certificate, (char *)host) != SECSuccess) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: SSL_BadCertHook error %d: %s\n",
       err, PR_ErrorToName(err));
    exit(1);
  }

  newfd = SSL_ImportFD(model, nspr);
  if (newfd == NULL) {
    const PRErrorCode err = PR_GetError();
    fprintf(stderr, "error: SSL_ImportFD error %d: %s\n",
       err, PR_ErrorToName(err));
    exit(1);
  }
  nspr = newfd;
  PR_Close(model);
}

// Perform the handshake.
if (SSL_ResetHandshake(nspr, PR_FALSE) != SECSuccess) {
  const PRErrorCode err = PR_GetError();
  fprintf(stderr, "error: SSL_ResetHandshake error %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}
if (SSL_SetURL(nspr, host) != SECSuccess) {
  const PRErrorCode err = PR_GetError();
  fprintf(stderr, "error: SSL_SetURL error %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}
if (SSL_ForceHandshake(nspr) != SECSuccess) {
  const PRErrorCode err = PR_GetError();
  fprintf(stderr, "error: SSL_ForceHandshake error %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}
```

After the connection has been established, *Example 12.24, “Using NSS for sending and receiving data”* shows how to use the NSPR descriptor to communicate with the server.

Example 12.24. Using NSS for sending and receiving data

```
char buf[4096];
snprintf(buf, sizeof(buf), "GET / HTTP/1.0\r\nHost: %s\r\n\r\n", host);
PRInt32 ret = PR_Write(nspr, buf, strlen(buf));
if (ret < 0) {
  const PRErrorCode err = PR_GetError();
```

```
  fprintf(stderr, "error: PR_Write error %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}
ret = PR_Read(nspr, buf, sizeof(buf));
if (ret < 0) {
  const PRErrorCode err = PR_GetError();
  fprintf(stderr, "error: PR_Read error %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}
```

Example 12.25, "Closing NSS client connections" shows how to close the connection.

Example 12.25. Closing NSS client connections

```
// Send close_notify alert.
if (PR_Shutdown(nspr, PR_SHUTDOWN_BOTH) != PR_SUCCESS) {
  const PRErrorCode err = PR_GetError();
  fprintf(stderr, "error: PR_Read error %d: %s\n",
     err, PR_ErrorToName(err));
  exit(1);
}
// Closes the underlying POSIX file descriptor, too.
PR_Close(nspr);
```

12.2.5. Implementing TLS Clients With Python

The Python distribution provides a TLS implementation in the **ssl** module (actually a wrapper around OpenSSL). The exported interface is somewhat restricted, so that the client code shown below does not fully implement the recommendations in *Section 12.1.1, "OpenSSL Pitfalls"*.

Important

Currently, most Python function which accept **https://** URLs or otherwise implement HTTPS support do not perform certificate validation at all. (For example, this is true for the **httplib** and **xmlrpclib** modules.) If you use HTTPS, you should not use the built-in HTTP clients. The **Curl** class in the **curl** module, as provided by the **python-pycurl** package implements proper certificate validation.

The **ssl** module currently does not perform host name checking on the server certificate. *Example 12.26, "Implementing TLS host name checking Python (without wildcard support)"* shows how to implement certificate matching, using the parsed certificate returned by `getpeercert`.

Example 12.26. Implementing TLS host name checking Python (without wildcard support)

```
def check_host_name(peercert, name):
    """Simple certificate/host name checker.  Returns True if the
    certificate matches, False otherwise.  Does not support
    wildcards."""
    # Check that the peer has supplied a certificate.
    # None/{} is not acceptable.
```

```
    if not peercert:
        return False
    if peercert.has_key("subjectAltName"):
        for typ, val in peercert["subjectAltName"]:
            if typ == "DNS" and val == name:
                return True
    else:
        # Only check the subject DN if there is no subject alternative
        # name.
        cn = None
        for attr, val in peercert["subject"]:
            # Use most-specific (last) commonName attribute.
            if attr == "commonName":
                cn = val
        if cn is not None:
            return cn == name
    return False
```

To turn a regular, connected TCP socket into a TLS-enabled socket, use the `ssl.wrap_socket` function. The function call in *Example 12.27, "Establishing a TLS client connection with Python"* provides additional arguments to override questionable defaults in OpenSSL and in the Python module.

- **`ciphers="HIGH:-aNULL:-eNULL:-PSK:RC4-SHA:RC4-MD5"`** selects relatively strong cipher suites with certificate-based authentication. (The call to `check_host_name` function provides additional protection against anonymous cipher suites.)
- **`ssl_version=ssl.PROTOCOL_TLSv1`** disables SSL 2.0 support. By default, the **`ssl`** module sends an SSL 2.0 client hello, which is rejected by some servers. Ideally, we would request OpenSSL to negotiated the most recent TLS version supported by the server and the client, but the Python module does not allow this.
- **`cert_reqs=ssl.CERT_REQUIRED`** turns on certificate validation.
- **`ca_certs='/etc/ssl/certs/ca-bundle.crt'`** initializes the certificate store with a set of trusted root CAs. Unfortunately, it is necessary to hard-code this path into applications because the default path in OpenSSL is not available through the Python **`ssl`** module.

The **`ssl`** module (and OpenSSL) perform certificate validation, but the certificate must be compared manually against the host name, by calling the `check_host_name` defined above.

Example 12.27. Establishing a TLS client connection with Python

```
sock = ssl.wrap_socket(sock,
                       ciphers="HIGH:-aNULL:-eNULL:-PSK:RC4-SHA:RC4-MD5",
                       ssl_version=ssl.PROTOCOL_TLSv1,
                       cert_reqs=ssl.CERT_REQUIRED,
                       ca_certs='/etc/ssl/certs/ca-bundle.crt')
# getpeercert() triggers the handshake as a side effect.
if not check_host_name(sock.getpeercert(), host):
    raise IOError("peer certificate does not match host name")
```

After the connection has been established, the TLS socket can be used like a regular socket:

```
sock.write("GET / HTTP/1.1\r\nHost: " + host + "\r\n\r\n")
print sock.read()
```

Closing the TLS socket is straightforward as well:

```
sock.close()
```

Appendix A. Revision History

Revision 0-1 **Thu Mar 7 2013** **Eric Christensen** *sparks@redhat.com*

Initial publication.