

Fedora Security Team

Defensive Coding

A Guide to Improving Software Security



Florian Weimer

Fedora Security Team Defensive Coding

A Guide to Improving Software Security

Edition 1

Author

Florian Weimer

fweimer@redhat.com

Copyright © 2012-2014 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at <http://creativecommons.org/licenses/by-sa/3.0/>. The original authors of this document, and Red Hat, designate the Fedora Project as the "Attribution Party" for purposes of CC-BY-SA. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

For guidelines on the permitted uses of the Fedora trademarks, refer to https://fedoraproject.org/wiki/Legal:Trademark_guidelines.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

All other trademarks are the property of their respective owners.

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

I. Programming Languages	1
1. The C Programming Language	3
1.1. The core language	3
1.1.1. Undefined behavior	3
1.1.2. Recommendations for pointers and array handling	3
1.1.3. Recommendations for integer arithmetic	4
1.1.4. Global variables	6
1.2. The C standard library	6
1.2.1. Absolutely banned interfaces	6
1.2.2. Functions to avoid	7
1.2.3. String Functions With Explicit Length Arguments	8
1.3. Memory allocators	10
1.3.1. malloc and related functions	10
1.3.2. alloca and other forms of stack-based allocation	11
1.3.3. Array allocation	11
1.3.4. Custom memory allocators	11
1.3.5. Conservative garbage collection	12
1.4. Other C-related topics	12
1.4.1. Wrapper functions	12
2. The C++ Programming Language	13
2.1. The core language	13
2.1.1. Array allocation with operator new[]	13
2.1.2. Overloading	13
2.1.3. ABI compatibility and preparing for security updates	13
2.1.4. C++0X and C++11 support	14
2.2. The C++ standard library	14
2.2.1. Functions that are difficult to use	14
2.2.2. String handling with std::string	15
2.2.3. Containers and operator[]	16
2.2.4. Iterators	16
3. The Java Programming Language	17
3.1. The core language	17
3.1.1. Increasing robustness when reading arrays	17
3.1.2. Resource management	18
3.1.3. Finalizers	18
3.1.4. Recovering from exceptions and errors	19
3.2. Low-level features of the virtual machine	20
3.2.1. Reflection and private parts	20
3.2.2. Java Native Interface (JNI)	20
3.2.3. sun.misc.Unsafe	22
3.3. Interacting with the security manager	22
3.3.1. Security manager compatibility	23
3.3.2. Activating the security manager	23
3.3.3. Reducing trust in code	23
3.3.4. Re-gaining privileges	24
4. The Python Programming Language	27
4.1. Dangerous standard library features	27
4.2. Run-time compilation and code generation	27
4.3. Sandboxing	27
5. Shell Programming and bash	29
5.1. Consider alternatives	29

5.2. Shell language features	29
5.2.1. Parameter expansion	29
5.2.2. Double expansion	29
5.2.3. Other obscurities	31
5.3. Invoking external commands	31
5.4. Temporary files	32
5.5. Performing input validation	32
5.6. Guarding shell scripts against changes	33
6. The Go Programming Language	35
6.1. Memory safety	35
6.2. Error handling	35
6.3. Garbage Collector	36
6.4. Marshaling and unmarshaling	36
7. The Vala Programming Language	37
II. Specific Programming Tasks	39
8. Library Design	41
8.1. State management	41
8.1.1. Global state	41
8.1.2. Handles	41
8.2. Object orientation	41
8.3. Callbacks	42
8.4. Process attributes	42
9. File Descriptor Management	43
9.1. Closing descriptors	43
9.1.1. Error handling during descriptor close	43
9.1.2. Closing descriptors and race conditions	43
9.1.3. Lingering state after close	43
9.2. Preventing file descriptor leaks to child processes	44
9.3. Dealing with the select limit	44
10. File system manipulation	47
10.1. Working with files and directories owned by other users	47
10.2. Accessing the file system as a different user	48
10.3. File system limits	48
10.4. File system features	49
10.5. Checking free space	49
11. Temporary files	51
11.1. Obtaining the location of temporary directory	51
11.2. Named temporary files	51
11.3. Temporary files without names	52
11.4. Temporary directories	52
11.5. Compensating for unsafe file creation	52
12. Processes	55
12.1. Safe process creation	55
12.1.1. Obtaining the program path and the command line template	55
12.1.2. Bypassing the shell	55
12.1.3. Specifying the process environment	56
12.1.4. Robust argument list processing	56
12.1.5. Passing secrets to subprocesses	57
12.2. Handling child process termination	57

12.3. SUID/SGID processes	57
12.3.1. Accessing environment variables	58
12.4. Daemons	58
12.5. Semantics of command line arguments	59
12.6. fork as a primitive for parallelism	59
13. Serialization and Deserialization	61
13.1. Recommendations for manually written decoders	61
13.2. Protocol design	61
13.3. Fragmentation	61
13.3.1. Fragment IDs	62
13.4. Library support for deserialization	62
13.5. XML serialization	63
13.5.1. External references	63
13.5.2. Entity expansion	64
13.5.3. XInclude processing	64
13.5.4. Algorithmic complexity of XML validation	64
13.5.5. Using Expat for XML parsing	64
13.5.6. Using Qt for XML parsing	65
13.5.7. Using OpenJDK for XML parsing and validation	67
13.6. Protocol Encoders	69
14. Cryptography	71
14.1. Primitives	71
14.2. Randomness	71
15. RPM packaging	73
15.1. Generating X.509 self-signed certificates during installation	73
15.2. Generating X.509 self-signed certificates before service start	74
III. Implementing Security Features	77
16. Authentication and Authorization	79
16.1. Authenticating servers	79
16.2. Host-based authentication	79
16.3. UNIX domain socket authentication	80
16.4. AF_NETLINK authentication of origin	80
17. Transport Layer Security	81
17.1. Common Pitfalls	81
17.1.1. OpenSSL Pitfalls	82
17.1.2. GNUTLS Pitfalls	83
17.1.3. OpenJDK Pitfalls	83
17.1.4. NSS Pitfalls	84
17.2. TLS Clients	84
17.2.1. Implementation TLS Clients With OpenSSL	85
17.2.2. Implementation TLS Clients With GNUTLS	88
17.2.3. Implementing TLS Clients With OpenJDK	92
17.2.4. Implementing TLS Clients With NSS	96
17.2.5. Implementing TLS Clients With Python	100
A. Revision History	103

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 subtracting 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;
        }
        *outp = s;
        ++outp;
    }
}
```

```
    }
    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. [Example 1.3, “Overflow checking for unsigned addition”](#) shows how to perform an overflow check for unsigned integer addition. For three or more terms, all the intermediate additions have to be checked in this way.

Example 1.3. Overflow checking for unsigned addition

```
void report_overflow(void);

unsigned
add_unsigned(unsigned a, unsigned b)
{
    unsigned sum = a + b;
    if (sum < a) { // or sum < b
        report_overflow();
    }
    return sum;
}
```

- 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 1.4, “Overflow checking for unsigned multiplication”](#).

Example 1.4. 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 are 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.

Sometimes, it is necessary to compare unsigned and signed integer variables. This results in a compiler warning, *comparison between signed and unsigned integer expressions*, because the comparison often gives unexpected results for negative values. When adding a cast, make sure that negative values are covered properly. If the bound is unsigned and the checked quantity is signed, you should cast the checked quantity to an unsigned type as least as wide as either operand type. As a result, negative values will fail the bounds check. (You can still check for negative values separately for clarity, and the compiler will optimize away this redundant check.)

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.

1.1.4. Global variables

Global variables should be avoided because they usually lead to thread safety hazards. In any case, they should be declared **static**, so that access is restricted to a single translation unit.

Global constants are not a problem, but declaring them can be tricky. [Example 1.5](#), “*Declaring a constant array of constant strings*” shows how to declare a constant array of constant strings. The second **const** is needed to make the array constant, and not just the strings. It must be placed after the `*`, and not before it.

Example 1.5. Declaring a constant array of constant strings

```
static const char *const string_list[] = {  
    "first",  
    "second",  
    "third",  
    NULL  
};
```

Sometimes, static variables local to functions are used as a replacement for proper memory management. Unlike non-static local variables, it is possible to return a pointer to static local variables to the caller. But such variables are well-hidden, but effectively global (just as static variables at file scope). It is difficult to add thread safety afterwards if such interfaces are used. Merely dropping the **static** keyword in such cases leads to undefined behavior.

Another source for static local variables is a desire to reduce stack space usage on embedded platforms, where the stack may span only a few hundred bytes. If this is the only reason why the **static** keyword is used, it can just be dropped, unless the object is very large (larger than 128 kilobytes on 32 bit platforms). In the latter case, it is recommended to allocate the object using **malloc**, to obtain proper array checking, for the same reasons outlined in [Section 1.3.2](#), “*alloca and other forms of stack-based allocation*”.

1.2. The C standard library

Parts of the C standard library (and the UNIX and GNU extensions) are difficult to use, so you should 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 12.1.3, “Specifying the process environment”](#))
- `setenv` # explicit `envp` argument in process creation (see [Section 12.1.3, “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 12.1.2, “Bypassing the shell”](#))
- `unsetenv` # explicit `envp` argument in process creation (see [Section 12.1.3, “Specifying the process environment”](#))

1.2.3. String Functions With Explicit Length Arguments

The C run-time library provides string manipulation functions which not just look for NUL characters for string termination, but also honor explicit lengths provided by the caller. However, these functions evolved over a long period of time, and the lengths mean different things depending on the function.

1.2.3.1. `snprintf`

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

```
char fraction[30];
snprintf(fraction, sizeof(fraction), "%d/%d", numerator, denominator);
```

The second argument to the `snprintf` call should always be the size of the buffer in the first argument (which should be a character array). Elaborate pointer and length arithmetic can introduce errors and nullify the security benefits of `snprintf`.

In particular, **`snprintf`** is not well-suited to constructing a string iteratively, by appending to an existing buffer. `snprintf` returns one of two values, **-1** on errors, or the number of characters which *would have been written to the buffer if the buffer were large enough*. This means that adding the result of `snprintf` to the buffer pointer to skip over the characters just written is incorrect and risky. However, as long as the length argument is not zero, the buffer will remain NUL-terminated.

Example 1.6, “Repeatedly writing to a buffer using `snprintf`” works because **`end - current > 0`** is a loop invariant. After the loop, the result string is in the `buf` variable.

Example 1.6. Repeatedly writing to a buffer using `snprintf`

```
char buf[512];
char *current = buf;
const char *const end = buf + sizeof(buf);
for (struct item *it = data; it->key; ++it) {
    snprintf(current, end - current, "%s%s=%d",
             current == buf ? "" : ", ", it->key, it->value);
    current += strlen(current);
}
```

If you want to avoid the call to `strlen` for performance reasons, you have to check for a negative return value from `snprintf` and also check if the return value is equal to the specified buffer length or larger. Only if neither condition applies, you may advance the pointer to the start of the write buffer by the number return by `snprintf`. However, this optimization is rarely worthwhile.

Note that it is not permitted to use the same buffer both as the destination and as a source argument.

1.2.3.2. `vsnprintf` and format strings

If you use `vsnprintf` (or `vasprintf` or even `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 1.7, “The `format` function attribute”*).

Example 1.7. 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);
}

```

1.2.3.3. strncpy

The `strncpy` function does not ensure that the target buffer is NUL-terminated. A common idiom for ensuring NUL termination is:

```

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

```

Another approach uses the `strncat` function for this purpose:

```

buf[0] = '\0';
strncat(buf, data, sizeof(buf) - 1);

```

1.2.3.4. strncat

The length argument of the `strncat` function specifies the maximum number of characters copied from the source buffer, excluding the terminating NUL character. This means that the required number of bytes in the destination buffer is the length of the original string, plus the length argument in the `strncat` call, plus one. Consequently, this function is rarely appropriate for performing a length-checked string operation, with the notable exception of the `strncpy` emulation described in [Section 1.2.3.3, “`strncpy`”](#).

To implement a length-checked string append, you can use an approach similar to [Example 1.6, “Repeatedly writing to a buffer using `snprintf`”](#):

```

char buf[10];
snprintf(buf, sizeof(buf), "%s", prefix);
snprintf(buf + strlen(buf), sizeof(buf) - strlen(buf), "%s", data);

```

In many cases, including this one, the string concatenation can be avoided by combining everything into a single format string:

```

snprintf(buf, sizeof(buf), "%s%s", prefix, data);

```

But you should must not dynamically construct format strings to avoid concatenation because this would prevent GCC from type-checking the argument lists.

It is not possible to use format strings like `"%s%s"` to implement concatenation, unless you use separate buffers. `snprintf` does not support overlapping source and target strings.

1.2.3.5. `strlcpy` and `strlcat`

Some systems support `strlcpy` and `strlcat` functions which behave this way, but these functions are not part of GNU libc. `strlcpy` is often replaced with `snprintf` with a "%s" format string. See [Section 1.2.3.3, “`strncpy`”](#) for a caveat related to the `snprintf` return value.

To emulate `strlcat`, use the approach described in [Section 1.2.3.4, “`strncat`”](#).

1.2.3.6. ISO C11 Annex K *_s functions

ISO C11 adds another set of length-checking functions, but GNU libc currently does not implement them.

1.2.3.7. Other `strn*` and `stp*` functions

GNU libc contains additional functions with different variants of length checking. Consult the documentation before using them to find out what the length actually means.

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 13.1, “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 reorganize 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 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 `((size_t) -1) / sizeof(T)`. See [Section 1.1.3, “Recommendations for integer arithmetic”](#).

1.3.4. Custom memory allocators

Custom memory allocators 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 external fragmentation increases the overall memory usage.

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 opportunities 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.

1.4. Other C-related topics

1.4.1. Wrapper functions

Some libraries provide wrappers for standard library functions. Common cases include allocation functions such as `xmalloc` which abort the process on allocation failure (instead of returning a **NULL** pointer), or alternatives to relatively recent library additions such as `snprintf` (along with implementations for systems which lack them).

In general, such wrappers are a bad idea, particularly if they are not implemented as inline functions or preprocessor macros. The compiler lacks knowledge of such wrappers outside the translation unit which defines them, which means that some optimizations and security checks are not performed. Adding `__attribute__` annotations to function declarations can remedy this to some extent, but these annotations have to be maintained carefully for feature parity with the standard implementation.

At the minimum, you should apply these attributes:

- If you wrap function which accepts a GCC-recognized format string (for example, a `printf`-style function used for logging), you should add a suitable **format** attribute, as in [Example 1.7, “The *format* function attribute”](#).
- If you wrap a function which carries a **warn_unused_result** attribute and you propagate its return value, your wrapper should be declared with **warn_unused_result** as well.
- Duplicating the buffer length checks based on the `__builtin_object_size` GCC builtin is desirable if the wrapper processes arrays. (This functionality is used by the `-D_FORTIFY_SOURCE=2` checks to guard against static buffer overflows.) However, designing appropriate interfaces and implementing the checks may not be entirely straightforward.

For other attributes (such as **malloc**), careful analysis and comparison with the compiler documentation is required to check if propagating the attribute is appropriate. Incorrectly applied attributes can result in undesired behavioral changes in the compiled code.

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 versions prior to 4.8 generated code which did not detect this. (Fedora 18 was the first release which fixed this in GCC.)

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[]` and the sources will be compiled with older GCC versions, 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 > 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 13.1, “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++](http://techbase.kde.org/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. Functions that are difficult to use

This section collects functions and function templates which are part of the standard library and are difficult to use.

2.2.1.1. Unpaired iterators

Functions which use output operators or iterators which do not come in pairs (denoting ranges) cannot perform iterator range checking. (See [Section 2.2.4, “Iterators”](#)) Function templates which involve output iterators are particularly dangerous:

- `std::copy`

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

- `std::copy_backward`
- `std::copy_if`
- `std::move` (three-argument variant)
- `std::move_backward`
- `std::partition_copy_if`
- `std::remove_copy`
- `std::remove_copy_if`
- `std::replace_copy`
- `std::replace_copy_if`
- `std::swap_ranges`
- `std::transform`

In addition, `std::copy_n`, `std::fill_n` and `std::generate_n` do not perform iterator checking, either, but there is an explicit count which has to be supplied by the caller, as opposed to an implicit length indicator in the form of a pair of forward iterators.

These output-iterator-expecting functions should only be used with unlimited-range output iterators, such as iterators obtained with the `std::back_inserter` function.

Other functions use single input or forward iterators, which can read beyond the end of the input range if the caller is not careful:

- `std::equal`
- `std::is_permutation`
- `std::mismatch`

2.2.2. String handling with `std::string`

The `std::string` class provides a convenient way to handle strings. Unlike C strings, `std::string` objects have an explicit length (and can contain embedded NUL characters), and storage for its characters is managed automatically. This section discusses `std::string`, but these observations also apply to other instances of the `std::basic_string` template.

The pointer returned by the `data()` member function does not necessarily point to a NUL-terminated string. To obtain a C-compatible string pointer, use `c_str()` instead, which adds the NUL terminator.

The pointers returned by the `data()` and `c_str()` functions and iterators are only valid until certain events happen. It is required that the exact `std::string` object still exists (even if it was initially created as a copy of another string object). Pointers and iterators are also invalidated when non-const member functions are called, or functions with a non-const reference parameter. The behavior of the GCC implementation deviates from that required by the C++ standard if multiple threads are present. In general, only the first call to a non-const member function after a structural modification of the string (such as appending a character) is invalidating, but this also applies to member function such as the non-const version of `begin()`, in violation of the C++ standard.

Particular care is necessary when invoking the `c_str()` member function on a temporary object. This is convenient for calling C functions, but the pointer will turn invalid as soon as the temporary object

is destroyed, which generally happens when the outermost expression enclosing the expression on which `c_str()` is called completes evaluation. Passing the result of `c_str()` to a function which does not store or otherwise leak that pointer is safe, though.

Like with `std::vector` and `std::array`, subscribing with `operator[]` does not perform bounds checks. Use the `at(size_type)` member function instead. See [Section 2.2.3, “Containers and operator\[\]”](#). Furthermore, accessing the terminating NUL character using `operator[]` is not possible. (In some implementations, the `c_str()` member function writes the NUL character on demand.)

Never write to the pointers returned by `data()` or `c_str()` after casting away `const`. If you need a C-style writable string, use a `std::vector<char>` object and its `data()` member function. In this case, you have to explicitly add the terminating NUL character.

GCC's implementation of `std::string` is currently based on reference counting. It is expected that a future version will remove the reference counting, due to performance and conformance issues. As a result, code that implicitly assumes sharing by holding to pointers or iterators for too long will break, resulting in run-time crashes or worse. On the other hand, non-const iterator-returning functions will no longer give other threads an opportunity for invalidating existing iterators and pointers because iterator invalidation does not depend on sharing of the internal character array object anymore.

2.2.3. Containers and operator[]

Many sequence 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.

The `front()` and `back()` member functions are undefined if a vector object is empty. You can use `vec.at(0)` and `vec.at(vec.size() - 1)` as checked replacements. For an empty vector, `data()` is defined; it returns an arbitrary pointer, but not necessarily the NULL pointer.

2.2.4. Iterators

Iterators do not perform any bounds checking. Therefore, all functions that work on iterators should accept them in pairs, denoting a range, and make sure that iterators are not moved outside that range. For forward iterators and bidirectional iterators, you need to check for equality before moving the first or last iterator in the range. For random-access iterators, you need to compute the difference before adding or subtracting an offset. It is not possible to perform the operation and check for an invalid operator afterwards.

Output iterators cannot be compared for equality. Therefore, it is impossible to write code that detects that it has been supplied an output area that is too small, and their use should be avoided.

These issues make some of the standard library functions difficult to use correctly, see [Section 2.2.1.1, “Unpaired iterators”](#).

The Java Programming Language

3.1. The core language

Implementations of the Java programming language provide strong memory safety, even in the presence of data races in concurrent code. This prevents a large range of security vulnerabilities from occurring, unless certain low-level features are used; see [Section 3.2, “Low-level features of the virtual machine”](#).

3.1.1. Increasing robustness when reading arrays

External data formats often include arrays, and the data is stored as an integer indicating the number of array elements, followed by this number of elements in the file or protocol data unit. This length specified can be much larger than what is actually available in the data source.

To avoid allocating extremely large amounts of data, you can allocate a small array initially and grow it as you read more data, implementing an exponential growth policy. See the `readBytes(InputStream, int)` function in [Example 3.1, “Incrementally reading a byte array”](#).

Example 3.1. Incrementally reading a byte array

```
static byte[] readBytes(InputStream in, int length) throws IOException {
    final int startSize = 65536;
    byte[] b = new byte[Math.min(length, startSize)];
    int filled = 0;
    while (true) {
        int remaining = b.length - filled;
        readFully(in, b, filled, remaining);
        if (b.length == length) {
            break;
        }
        filled = b.length;
        if (length - b.length <= b.length) {
            // Allocate final length. Condition avoids overflow.
            b = Arrays.copyOf(b, length);
        } else {
            b = Arrays.copyOf(b, b.length * 2);
        }
    }
    return b;
}

static void readFully(InputStream in, byte[] b, int off, int len)
    throws IOException {
    int startlen = len;
    while (len > 0) {
        int count = in.read(b, off, len);
        if (count < 0) {
            throw new EOFException();
        }
        off += count;
        len -= count;
    }
}
```

When reading data into arrays, hash maps or hash sets, use the default constructor and do not specify a size hint. You can simply add the elements to the collection as you read them.

3.1.2. Resource management

Unlike C++, Java does not offer destructors which can deallocate resources in a predictable fashion. All resource management has to be manual, at the usage site. (Finalizers are generally not usable for resource management, especially in high-performance code; see [Section 3.1.3, “Finalizers”](#).)

The first option is the **try-finally** construct, as shown in [Example 3.2, “Resource management with a try-finally block”](#). The code in the **finally** block should be as short as possible and should not throw any exceptions.

Example 3.2. Resource management with a **try-finally** block

```
InputStream in = new BufferedInputStream(new FileInputStream(path));
try {
    readFile(in);
} finally {
    in.close();
}
```

Note that the resource allocation happens *outside* the **try** block, and that there is no **null** check in the **finally** block. (Both are common artifacts stemming from IDE code templates.)

If the resource object is created freshly and implements the **java.lang.AutoCloseable** interface, the code in [Example 3.3, “Resource management using the try-with-resource construct”](#) can be used instead. The Java compiler will automatically insert the `close()` method call in a synthetic **finally** block.

Example 3.3. Resource management using the **try-with-resource** construct

```
try (InputStream in = new BufferedInputStream(new FileInputStream(path))) {
    readFile(in);
}
```

To be compatible with the **try-with-resource** construct, new classes should name the resource deallocation method `close()`, and implement the **AutoCloseable** interface (the latter breaking backwards compatibility with Java 6). However, using the **try-with-resource** construct with objects that are not freshly allocated is at best awkward, and an explicit **finally** block is usually the better approach.

In general, it is best to design the programming interface in such a way that resource deallocation methods like `close()` cannot throw any (checked or unchecked) exceptions, but this should not be a reason to ignore any actual error conditions.

3.1.3. Finalizers

Finalizers can be used a last-resort approach to free resources which would otherwise leak. Finalization is unpredictable, costly, and there can be a considerable delay between the last reference to an object going away and the execution of the finalizer. Generally, manual resource management is required; see [Section 3.1.2, “Resource management”](#).

Finalizers should be very short and should only deallocate native or other external resources held directly by the object being finalized. In general, they must use synchronization: Finalization necessarily happens on a separate thread because it is inherently concurrent. There can be multiple

finalization threads, and despite each object being finalized at most once, the finalizer must not assume that it has exclusive access to the object being finalized (in the **this** pointer).

Finalizers should not deallocate resources held by other objects, especially if those objects have finalizers on their own. In particular, it is a very bad idea to define a finalizer just to invoke the resource deallocation method of another object, or overwrite some pointer fields.

Finalizers are not guaranteed to run at all. For instance, the virtual machine (or the machine underneath) might crash, preventing their execution.

Objects with finalizers are garbage-collected much later than objects without them, so using finalizers to zero out key material (to reduce its undecrypted lifetime in memory) may have the opposite effect, keeping objects around for much longer and prevent them from being overwritten in the normal course of program execution.

For the same reason, code which allocates objects with finalizers at a high rate will eventually fail (likely with a **java.lang.OutOfMemoryError** exception) because the virtual machine has finite resources for keeping track of objects pending finalization. To deal with that, it may be necessary to recycle objects with finalizers.

The remarks in this section apply to finalizers which are implemented by overriding the `finalize()` method, and to custom finalization using reference queues.

3.1.4. Recovering from exceptions and errors

Java exceptions come in three kinds, all ultimately deriving from **java.lang.Throwable**:

- *Run-time exceptions* do not have to be declared explicitly and can be explicitly thrown from any code, by calling code which throws them, or by triggering an error condition at run time, like division by zero, or an attempt at an out-of-bounds array access. These exceptions derive from the **java.lang.RuntimeException** class (perhaps indirectly).
- *Checked exceptions* have to be declared explicitly by functions that throw or propagate them. They are similar to run-time exceptions in other regards, except that there is no language construct to throw them (except the **throw** statement itself). Checked exceptions are only present at the Java language level and are only enforced at compile time. At run time, the virtual machine does not know about them and permits throwing exceptions from any code. Checked exceptions must derive (perhaps indirectly) from the **java.lang.Exception** class, but not from **java.lang.RuntimeException**.
- *Errors* are exceptions which typically reflect serious error conditions. They can be thrown at any point in the program, and do not have to be declared (unlike checked exceptions). In general, it is not possible to recover from such errors; more on that below, in [Section 3.1.4.1, “The difficulty of catching errors”](#). Error classes derive (perhaps indirectly) from **java.lang.Error**, or from **java.lang.Throwable**, but not from **java.lang.Exception**.

The general expectation is that run-time errors are avoided by careful programming (e.g., not dividing by zero). Checked exceptions are expected to be caught as they happen (e.g., when an input file is unexpectedly missing). Errors are impossible to predict and can happen at any point and reflect that something went wrong beyond all expectations.

3.1.4.1. The difficulty of catching errors

Errors (that is, exceptions which do not (indirectly) derive from **java.lang.Exception**), have the peculiar property that catching them is problematic. There are several reasons for this:

- The error reflects a failed consistency check, for example, **java.lang.AssertionError**.

- The error can happen at any point, resulting in inconsistencies due to half-updated objects. Examples are `java.lang.ThreadDeath`, `java.lang.OutOfMemoryError` and `java.lang.StackOverflowError`.
- The error indicates that virtual machine failed to provide some semantic guarantees by the Java programming language. `java.lang.ExceptionInInitializerError` is an example—it can leave behind a half-initialized class.

In general, if an error is thrown, the virtual machine should be restarted as soon as possible because it is in an inconsistent state. Continuing running as before can have unexpected consequences. However, there are legitimate reasons for catching errors because not doing so leads to even greater problems.

Code should be written in a way that avoids triggering errors. See [Section 3.1.1, “Increasing robustness when reading arrays”](#) for an example.

It is usually necessary to log errors. Otherwise, no trace of the problem might be left anywhere, making it very difficult to diagnose realted failures. Consequently, if you catch `java.lang.Exception` to log and suppress all unexpected exceptions (for example, in a request dispatching loop), you should consider switching to `java.lang.Throwable` instead, to also cover errors.

The other reason mainly applies to such request dispatching loops: If you do not catch errors, the loop stops looping, resulting in a denial of service.

However, if possible, catching errors should be coupled with a way to signal the requirement of a virtual machine restart.

3.2. Low-level features of the virtual machine

3.2.1. Reflection and private parts

The `setAccessible(boolean)` method of the `java.lang.reflect.AccessibleObject` class allows a program to disable language-defined access rules for specific constructors, methods, or fields. Once the access checks are disabled, any code can use the `java.lang.reflect.Constructor`, `java.lang.reflect.Method`, or `java.lang.reflect.Field` object to access the underlying Java entity, without further permission checks. This breaks encapsulation and can undermine the stability of the virtual machine. (In contrast, without using the `setAccessible(boolean)` method, this should not happen because all the language-defined checks still apply.)

This feature should be avoided if possible.

3.2.2. Java Native Interface (JNI)

The Java Native Interface allows calling from Java code functions specifically written for this purpose, usually in C or C++.

The transition between the Java world and the C world is not fully type-checked, and the C code can easily break the Java virtual machine semantics. Therefore, extra care is needed when using this functionality.

To provide a moderate amount of type safety, it is recommended to recreate the class-specific header file using `javah` during the build process, include it in the implementation, and use the `-Wmissing-declarations` option.

Ideally, the required data is directly passed to static JNI methods and returned from them, and the code and the C side does not have to deal with accessing Java fields (or even methods).

When using `GetPrimitiveArrayCritical` or `GetStringCritical`, make sure that you only perform very little processing between the get and release operations. Do not access the file system or the network, and not perform locking, because that might introduce blocking. When processing large strings or arrays, consider splitting the computation into multiple sub-chunks, so that you do not prevent the JVM from reaching a safepoint for extended periods of time.

If necessary, you can use the Java **long** type to store a C pointer in a field of a Java class. On the C side, when casting between the **jlong** value and the pointer on the C side,

You should not try to perform pointer arithmetic on the Java side (that is, you should treat pointer-carrying **long** values as opaque). When passing a slice of an array to the native code, follow the Java convention and pass it as the base array, the integer offset of the start of the slice, and the integer length of the slice. On the native side, check the offset/length combination against the actual array length, and use the offset to compute the pointer to the beginning of the array.

Example 3.4. Array length checking in JNI code

```
JNIEXPORT jint JNICALL Java_sum
(JNIEnv *jEnv, jclass clazz, jbyteArray buffer, jint offset, jint length)
{
    assert(sizeof(jint) == sizeof(unsigned));
    if (offset < 0 || length < 0) {
        (*jEnv)->ThrowNew(jEnv, arrayIndexOutOfBoundsExceptionClass,
            "negative offset/length");
        return 0;
    }
    unsigned uoffset = offset;
    unsigned ulength = length;
    // This cannot overflow because of the check above.
    unsigned totallength = uoffset + ulength;
    unsigned actuallength = (*jEnv)->GetArrayLength(jEnv, buffer);
    if (totallength > actuallength) {
        (*jEnv)->ThrowNew(jEnv, arrayIndexOutOfBoundsExceptionClass,
            "offset + length too large");
        return 0;
    }
    unsigned char *ptr = (*jEnv)->GetPrimitiveArrayCritical(jEnv, buffer, 0);
    if (ptr == NULL) {
        return 0;
    }
    unsigned long long sum = 0;
    for (unsigned char *p = ptr + uoffset, *end = p + ulength; p != end; ++p) {
        sum += *p;
    }
    (*jEnv)->ReleasePrimitiveArrayCritical(jEnv, buffer, ptr, 0);
    return sum;
}
```

In any case, classes referring to native resources must be declared **final**, and must not be serializable or cloneable. Initialization and mutation of the state used by the native side must be controlled carefully. Otherwise, it might be possible to create an object with inconsistent native state which results in a crash (or worse) when used (or perhaps only finalized) later. If you need both Java inheritance and native resources, you should consider moving the native state to a separate class, and only keep a reference to objects of that class. This way, cloning and serialization issues can be avoided in most cases.

If there are native resources associated with an object, the class should have an explicit resource deallocation method ([Section 3.1.2, “Resource management”](#)) and a finalizer ([Section 3.1.3, “Finalizers”](#)) as a last resort. The need for finalization means that a minimum amount of synchronization is needed. Code on the native side should check that the object is not in a closed/freed state.

Many JNI functions create local references. By default, these persist until the JNI-implemented method returns. If you create many such references (e.g., in a loop), you may have to free them using `DeleteLocalRef`, or start using `PushLocalFrame` and `PopLocalFrame`. Global references must be deallocated with `DeleteGlobalRef`, otherwise there will be a memory leak, just as with `malloc` and `free`.

When throwing exceptions using `Throw` or `ThrowNew`, be aware that these functions return regularly. You have to return control manually to the JVM.

Technically, the `JNIEnv` pointer is not necessarily constant during the lifetime of your JNI module. Storing it in a global variable is therefore incorrect. Particularly if you are dealing with callbacks, you may have to store the pointer in a thread-local variable (defined with `__thread`). It is, however, best to avoid the complexity of calling back into Java code.

Keep in mind that C/C++ and Java are different languages, despite very similar syntax for expressions. The Java memory model is much more strict than the C or C++ memory models, and native code needs more synchronization, usually using JVM facilities or POSIX threads mutexes. Integer overflow in Java is defined, but in C/C++ it is not (for the `jint` and `jlong` types).

3.2.3. `sun.misc.Unsafe`

The `sun.misc.Unsafe` class is unportable and contains many functions explicitly designed to break Java memory safety (for performance and debugging). If possible, avoid using this class.

3.3. Interacting with the security manager

The Java platform is largely implemented in the Java language itself. Therefore, within the same JVM, code runs which is part of the Java installation and which is trusted, but there might also be code which comes from untrusted sources and is restricted by the Java sandbox (to varying degrees). The *security manager* draws a line between fully trusted, partially trusted and untrusted code.

The type safety and accessibility checks provided by the Java language and JVM would be sufficient to implement a sandbox. However, only some Java APIs employ such a capabilities-based approach. (The Java SE library contains many public classes with public constructors which can break any security policy, such as `java.io.FileOutputStream`.) Instead, critical functionality is protected by *stack inspection*: At a security check, the stack is walked from top (most-nested) to bottom. The security check fails if a stack frame for a method is encountered whose class lacks the permission which the security check requires.

This simple approach would not allow untrusted code (which lacks certain permissions) to call into trusted code while the latter retains trust. Such trust transitions are desirable because they enable Java as an implementation language for most parts of the Java platform, including security-relevant code. Therefore, there is a mechanism to mark certain stack frames as trusted ([Section 3.3.4, “Re-gaining privileges”](#)).

In theory, it is possible to run a Java virtual machine with a security manager that acts very differently from this approach, but a lot of code expects behavior very close to the platform default (including many classes which are part of the OpenJDK implementation).

3.3.1. Security manager compatibility

A lot of code can run without any additional permissions at all, with little changes. The following guidelines should help to increase compatibility with a restrictive security manager.

- When retrieving system properties using `System.getProperty(String)` or similar methods, catch **SecurityException** exceptions and treat the property as unset.
- Avoid unnecessary file system or network access.
- Avoid explicit class loading. Access to a suitable class loader might not be available when executing as untrusted code.

If the functionality you are implementing absolutely requires privileged access and this functionality has to be used from untrusted code (hopefully in a restricted and secure manner), see [Section 3.3.4, “Re-gaining privileges”](#).

3.3.2. Activating the security manager

The usual command to launch a Java application, **java**, does not activate the security manager. Therefore, the virtual machine does not enforce any sandboxing restrictions, even if explicitly requested by the code (for example, as described in [Section 3.3.3, “Reducing trust in code”](#)).

The **-Djava.security.manager** option activates the security manager, with the fairly restrictive default policy. With a very permissive policy, most Java code will run unchanged. Assuming the policy in [Example 3.5, “Most permissive OpenJDK policy file”](#) has been saved in a file **grant-all.policy**, this policy can be activated using the option **-Djava.security.policy=grant-all.policy** (in addition to the **-Djava.security.manager** option).

Example 3.5. Most permissive OpenJDK policy file

```
grant {
    permission java.security.AllPermission;
};
```

With this most permissive policy, the security manager is still active, and explicit requests to drop privileges will be honored.

3.3.3. Reducing trust in code

[Example 3.6, “Using the security manager to run code with reduced privileges”](#) shows how to run a piece of code with reduced privileges.

Example 3.6. Using the security manager to run code with reduced privileges

```
Permissions permissions = new Permissions();
ProtectionDomain protectionDomain =
    new ProtectionDomain(null, permissions);
AccessControlContext context = new AccessControlContext(
    new ProtectionDomain[] { protectionDomain });

// This is expected to succeed.
try (FileInputStream in = new FileInputStream(path)) {
    System.out.format("FileInputStream: %s%n", in);
}
```

```
}  
  
AccessController.doPrivileged(new PrivilegedExceptionAction<Void>() {  
    @Override  
    public Void run() throws Exception {  
        // This code runs with reduced privileges and is  
        // expected to fail.  
        try (FileInputStream in = new FileInputStream(path)) {  
            System.out.format("FileInputStream: %s%n", in);  
        }  
        return null;  
    }  
}, context);
```

The example above does not add any additional permissions to the **permissions** object. If such permissions are necessary, code like the following (which grants read permission on all files in the current directory) can be used:

```
permissions.add(new FilePermission(  
    System.getProperty("user.dir") + "/-", "read"));
```



Important

Calls to the `java.security.AccessController.doPrivileged()` methods do not enforce any additional restriction if no security manager has been set. Except for a few special exceptions, the restrictions no longer apply if the `doPrivileged()` has returned, even to objects created by the code which ran with reduced privileges. (This applies to object finalization in particular.)

The example code above does not prevent the called code from calling the `java.security.AccessController.doPrivileged()` methods. This mechanism should be considered an additional safety net, but it still can be used to prevent unexpected behavior of trusted code. As long as the executed code is not dynamic and came with the original application or library, the sandbox is fairly effective.

The **context** argument in [Example 3.6, “Using the security manager to run code with reduced privileges”](#) is extremely important—otherwise, this code would increase privileges instead of reducing them.

For activating the security manager, see [Section 3.3.2, “Activating the security manager”](#). Unfortunately, this affects the virtual machine as a whole, so it is not possible to do this from a library.

3.3.4. Re-gaining privileges

Ordinarily, when trusted code is called from untrusted code, it loses its privileges (because of the untrusted stack frames visible to stack inspection). The `java.security.AccessController.doPrivileged()` family of methods provides a controlled backdoor from untrusted to trusted code.



Important

By design, this feature can undermine the Java security model and the sandbox. It has to be used very carefully. Most sandbox vulnerabilities can be traced back to its misuse.

In essence, the `doPrivileged()` methods cause the stack inspection to end at their call site. Untrusted code further down the call stack becomes invisible to security checks.

The following operations are common and safe to perform with elevated privileges.

- Reading custom system properties with fixed names, especially if the value is not propagated to untrusted code. (File system paths including installation paths, host names and user names are sometimes considered private information and need to be protected.)
- Reading from the file system at fixed paths, either determined at compile time or by a system property. Again, leaking the file contents to the caller can be problematic.
- Accessing network resources under a fixed address, name or URL, derived from a system property or configuration file, information leaks notwithstanding.

Example 3.7, “Using the security manager to run code with increased privileges” shows how to request additional privileges.

Example 3.7. Using the security manager to run code with increased privileges

```
// This is expected to fail.
try {
    System.out.println(System.getProperty("user.home"));
} catch (SecurityException e) {
    e.printStackTrace(System.err);
}
AccessController.doPrivileged(new PrivilegedAction<Void>() {
    public Void run() {
        // This should work.
        System.out.println(System.getProperty("user.home"));
        return null;
    }
});
```

Obviously, this only works if the class containing the call to `doPrivileged()` is marked trusted (usually because it is loaded from a trusted class loader).

When writing code that runs with elevated privileges, make sure that you follow the rules below.

- Make the privileged code as small as possible. Perform as many computations as possible before and after the privileged code section, even if it means that you have to define a new class to pass the data around.
- Make sure that you either control the inputs to the privileged code, or that the inputs are harmless and cannot affect security properties of the privileged code.

- Data that is returned from or written by the privileged code must either be restricted (that is, it cannot be accessed by untrusted code), or must be harmless. Otherwise, privacy leaks or information disclosures which affect security properties can be the result.

If the code calls back into untrusted code at a later stage (or performs other actions under control from the untrusted caller), you must obtain the original security context and restore it before performing the callback, as in [Example 3.8, “Restoring privileges when invoking callbacks”](#). (In this example, it would be much better to move the callback invocation out of the privileged code section, of course.)

Example 3.8. Restoring privileges when invoking callbacks

```
interface Callback<T> {
    T call(boolean flag);
}

class CallbackInvoker<T> {
    private final AccessControlContext context;
    Callback<T> callback;

    CallbackInvoker(Callback<T> callback) {
        context = AccessController.getContext();
        this.callback = callback;
    }

    public T invoke() {
        // Obtain increased privileges.
        return AccessController.doPrivileged(new PrivilegedAction<T>() {
            @Override
            public T run() {
                // This operation would fail without
                // additional privileges.
                final boolean flag = Boolean.getBoolean("some.property");

                // Restore the original privileges.
                return AccessController.doPrivileged(
                    new PrivilegedAction<T>() {
                        @Override
                        public T run() {
                            return callback.call(flag);
                        }
                    }, context);
            }
        });
    }
}
```


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 11, Temporary files](#)
- [Section 12.1, “Safe process creation”](#)
- [Chapter 13, Serialization and Deserialization](#), in particular [Section 13.4, “Library support for deserialization”](#)
- [Section 14.2, “Randomness”](#)

4.1. 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.

4.2. 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.

4.3. Sandboxing

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

Shell Programming and bash

This chapter contains advice about shell programming, specifically in **bash**. Most of the advice will apply to scripts written for other shells because extensions such as integer or array variables have been implemented there as well, with comparable syntax.

5.1. Consider alternatives

Once a shell script is so complex that advice in this chapter applies, it is time to step back and consider the question: Is there a more suitable implementation language available?

For example, Python with its **subprocess** module can be used to write scripts which are almost as concise as shell scripts when it comes to invoking external programs, and Python offers richer data structures, with less arcane syntax and more consistent behavior.

5.2. Shell language features

The following sections cover subtleties concerning the shell programming languages. They have been written with the **bash** shell in mind, but some of these features apply to other shells as well.

Some of the features described may seem like implementation defects, but these features have been replicated across multiple independent implementations, so they now have to be considered part of the shell programming language.

5.2.1. Parameter expansion

The mechanism by which named shell variables and parameters are expanded is called *parameter expansion*. The most basic syntax is “`$variable`” or “`${variable}`”.

In almost all cases, a parameter expansion should be enclosed in double quotation marks “`...`”.

```
external-program "$arg1" "$arg2"
```

If the double quotation marks are omitted, the value of the variable will be split according to the current value of the IFS variable. This may allow the injection of additional options which are then processed by **external-program**.

Parameter expansion can use special syntax for specific features, such as substituting defaults or performing string or array operations. These constructs should not be used because they can trigger arithmetic evaluation, which can result in code execution. See [Section 5.2.2.1, “Arithmetic evaluation”](#).

5.2.2. Double expansion

Double expansion occurs when, during the expansion of a shell variable, not just the variable is expanded, replacing it by its value, but the *value* of the variable is itself expanded as well. This can trigger arbitrary code execution, unless the value of the variable is verified against a restrictive pattern.

The evaluation process is in fact recursive, so a self-referential expression can cause an out-of-memory condition and a shell crash.

Double expansion may seem like as a defect, but it is implemented by many shells, and has to be considered an integral part of the shell programming language. However, it does make writing robust shell scripts difficult.

Double expansion can be requested explicitly with the **eval** built-in command, or by invoking a subshell with “**bash -c**”. These constructs should not be used.

The following sections give examples of places where implicit double expansion occurs.

5.2.2.1. Arithmetic evaluation

Arithmetic evaluation is a process by which the shell computes the integer value of an expression specified as a string. It is highly problematic for two reasons: It triggers double expansion (see [Section 5.2.2, “Double expansion”](#)), and the language of arithmetic expressions is not self-contained. Some constructs in arithmetic expressions (notably array subscripts) provide a trapdoor from the restricted language of arithmetic expressions to the full shell language, thus paving the way towards arbitrary code execution. Due to double expansion, input which is (indirectly) referenced from an arithmetic expression can trigger execution of arbitrary code, which is potentially harmful.

Arithmetic evaluation is triggered by the follow constructs:

- The *expression* in “**\$((expression))**” is evaluated. This construct is called *arithmetic expansion*.
- “**[\$expression]**” is a deprecated syntax with the same effect.
- The arguments to the **let** shell built-in are evaluated.
- “**((expression))**” is an alternative syntax for “**let expression**”.
- Conditional expressions surrounded by “**[[...]]**” can trigger arithmetic evaluation if certain operators such as **-eq** are used. (The **test** built-in does not perform arithmetic evaluation, even with integer operators such as **-eq**.)

The conditional expression “**[[\$variable =~ regexp]]**” can be used for input validation, assuming that *regexp* is a constant regular expression. See [Section 5.5, “Performing input validation”](#).

- Certain parameter expansions, for example “**\${variable[expression]}**” (array indexing) or “**\${variable:expression}**” (string slicing), trigger arithmetic evaluation of *expression*.
- Assignment to array elements using “**array_variable[subscript]=expression**” triggers evaluation of *subscript*, but not *expression*.
- The expressions in the arithmetic **for** command, “**for ((expression1; expression2; expression3)); do commands; done**” are evaluated. This does not apply to the regular **for** command, “**for variable in list; do commands; done**”.



Important

Depending on the **bash** version, the above list may be incomplete.

If faced with a situation where using such shell features appears necessary, see [Section 5.1, “Consider alternatives”](#).

If it is impossible to avoid shell arithmetic on untrusted inputs, refer to [Section 5.5, “Performing input validation”](#).

5.2.2.2. Type declarations

bash supports explicit type declarations for shell variables:

```
declare -i integer_variable
declare -a array_variable
declare -A assoc_array_variable

typeset -i integer_variable
typeset -a array_variable
typeset -A assoc_array_variable

local -i integer_variable
local -a array_variable
local -A assoc_array_variable

readonly -i integer_variable
readonly -a array_variable
readonly -A assoc_array_variable
```

Variables can also be declared as arrays by assigning them an array expression, as in:

```
array_variable=(1 2 3 4)
```

Some built-ins (such as **mapfile**) can implicitly create array variables.

Such type declarations should not be used because assignment to such variables (independent of the concrete syntax used for the assignment) triggers arithmetic expansion (and thus double expansion) of the right-hand side of the assignment operation. See [Section 5.2.2.1, “Arithmetic evaluation”](#).

Shell scripts which use integer or array variables should be rewritten in another, more suitable language. See [Section 5.1, “Consider alternatives”](#).

5.2.3. Other obscurities

Obscure shell language features should not be used. Examples are:

- Exported functions (**export -f** or **declare -f**).
- Function names which are not valid variable names, such as “**module::function**”.
- The possibility to override built-ins or external commands with shell functions.
- Changing the value of the IFS variable to tokenize strings.

5.3. Invoking external commands

When passing shell variables as single command line arguments, they should always be surrounded by double quotes. See [Section 5.2.1, “Parameter expansion”](#).

Care is required when passing untrusted values as positional parameters to external commands. If the value starts with a hyphen “-”, it may be interpreted by the external command as an option. Depending on the external program, a “- -” argument stops option processing and treats all following arguments as positional parameters. (Double quotes are completely invisible to the command being invoked, so they do not prevent variable values from being interpreted as options.)

Cleaning the environment before invoking child processes is difficult to implement in script. **bash** keeps a hidden list of environment variables which do not correspond to shell variables, and unsetting them from within a **bash** script is not possible. To reset the environment, a script can re-run itself under the “**env -i**” command with an additional parameter which indicates the environment has been cleared and suppresses a further self-execution. Alternatively, individual commands can be executed with “**env -i**”.



Important

Completely isolation from its original execution environment (which is required when the script is executed after a trust transition, e.g., triggered by the SUID mechanism) is impossible to achieve from within the shell script itself. Instead, the invoking process has to clear the process environment (except for few trusted variables) before running the shell script.

Checking for failures in executed external commands is recommended. If no elaborate error recovery is needed, invoking “**set -e**” may be sufficient. This causes the script to stop on the first failed command. However, failures in pipes (“**command1 | command2**”) are only detected for the last command in the pipe, errors in previous commands are ignored. This can be changed by invoking “**set -o pipefail**”. Due to architectural limitations, only the process that spawned the entire pipe can check for failures in individual commands; it is not possible for a process to tell if the process feeding data (or the process consuming data) exited normally or with an error.

See [Section 12.1, “Safe process creation”](#) for additional details on creating child processes.

5.4. Temporary files

Temporary files should be created with the **mktemp** command, and temporary directories with “**mktemp -d**”.

To clean up temporary files and directories, write a clean-up shell function and register it as a trap handler, as shown in [Example 5.1, “Creating and cleaning up temporary files”](#). Using a separate function avoids issues with proper quoting of variables.

Example 5.1. Creating and cleaning up temporary files

```
tmpfile="$(mktemp)"

cleanup () {
    rm -f -- "$tmpfile"
}

trap cleanup 0
```

5.5. Performing input validation

In some cases, input validation cannot be avoided. For example, if arithmetic evaluation is absolutely required, it is imperative to check that input values are, in fact, integers. See [Section 5.2.2.1, “Arithmetic evaluation”](#).

[Example 5.2, “Input validation in bash”](#) shows a construct which can be used to check if a string “**\$value**” is an integer. This construct is specific to **bash** and not portable to POSIX shells.

Example 5.2. Input validation in **bash**

```
if [[ $value =~ ^-?[0-9]+$ ]] ; then
    echo value is an integer
else
    echo "value is not an integer" 1>&2
    exit 1
fi
```

Using **case** statements for input validation is also possible and supported by other (POSIX) shells, but the pattern language is more restrictive, and it can be difficult to write suitable patterns.

The **expr** external command can give misleading results (e.g., if the value being checked contains operators itself) and should not be used.

5.6. Guarding shell scripts against changes

bash only reads a shell script up to the point it is needed for executed the next command. This means that if script is overwritten while it is running, execution can jump to a random part of the script, depending on what is modified in the script and how the file offsets change as a result. (This behavior is needed to support self-extracting shell archives whose script part is followed by a stream of bytes which does not follow the shell language syntax.)

Therefore, long-running scripts should be guarded against concurrent modification by putting as much of the program logic into a **main** function, and invoking the **main** function at the end of the script, using this syntax:

```
main "$@" ; exit $?
```

This construct ensures that **bash** will stop execution after the **main** function, instead of opening the script file and trying to read more commands.

The Go Programming Language

This chapter contains language-specific recommendations for Go.

6.1. Memory safety

Go provides memory safety, but only if the program is not executed in parallel (that is, `GOMAXPROCS` is not larger than `1`). The reason is that interface values and slices consist of multiple words are not updated atomically. Another thread of execution can observe an inconsistent pairing between type information and stored value (for interfaces) or pointer and length (for slices), and such inconsistency can lead to a memory safety violation.

Code which does not run in parallel and does not use the **unsafe** package (or other packages which expose unsafe constructs) is memory-safe. For example, invalid casts and out-of-range subscripting cause panics at run time.

Keep in mind that finalization can introduce parallelism because finalizers are executed concurrently, potentially interleaved with the rest of the program.

6.2. Error handling

Only a few common operations (such as pointer dereference, integer division, array subscripting) trigger exceptions in Go, called *panics*. Most interfaces in the standard library use a separate return value of type **error** to signal error.

Not checking error return values can lead to incorrect operation and data loss (especially in the case of writes, using interfaces such as **io.Writer**).

The correct way to check error return values depends on the function or method being called. In the majority of cases, the first step after calling a function should be an error check against the **nil** value, handling any encountered error. See [Example 6.1, “Regular error handling in Go”](#) for details.

Example 6.1. Regular error handling in Go

```
type Processor interface {
    Process(buf []byte) (message string, err error)
}

type ErrorHandler interface {
    Handle(err error)
}

func RegularError(buf []byte, processor Processor,
    handler ErrorHandler) (message string, err error) {
    message, err = processor.Process(buf)
    if err != nil {
        handler.Handle(err)
        return "", err
    }
    return
}
```

However, with **io.Reader**, **io.ReaderAt** and related interfaces, it is necessary to check for a non-zero number of read bytes first, as shown in [Example 6.2, “Read error handling in Go”](#). If this pattern is not followed, data loss may occur. This is due to the fact that the **io.Reader** interface permits returning both data and an error at the same time.

Example 6.2. Read error handling in Go

```
func IOError(r io.Reader, buf []byte, processor Processor,
    handler ErrorHandler) (message string, err error) {
    n, err := r.Read(buf)
    // First check for available data.
    if n > 0 {
        message, err = processor.Process(buf[0:n])
        // Regular error handling.
        if err != nil {
            handler.Handle(err)
            return "", err
        }
    }
    // Then handle any error.
    if err != nil {
        handler.Handle(err)
        return "", err
    }
    return
}
```

6.3. Garbage Collector

Older Go releases (before Go 1.3) use a conservative garbage collector without blacklisting. This means that data blobs can cause retention of unrelated data structures because the data is conservatively interpreted as pointers. This phenomenon can be triggered accidentally on 32-bit architectures and is more likely to occur if the heap grows larger. On 64-bit architectures, it may be possible to trigger it deliberately—it is unlikely to occur spontaneously.

6.4. Marshaling and unmarshaling

Several packages in the **encoding** hierarchy provide support for serialization and deserialization. The usual caveats apply (see [Chapter 13, Serialization and Deserialization](#)).

As an additional precaution, the `Unmarshal` and `Decode` functions should only be used with fresh values in the **interface{}** argument. This is due to the way defaults for missing values are implemented: During deserialization, missing value do not result in an error, but the original value is preserved. Using a fresh value (with suitable default values if necessary) ensures that data from a previous deserialization operation does not leak into the current one. This is especially relevant when structs are deserialized.

The Vala Programming Language

Vala is a programming language mainly targeted at GNOME developers.

Its syntax is inspired by C# (and thus, indirectly, by Java). But unlike C# and Java, Vala does not attempt to provide memory safety: Vala is compiled to C, and the C code is compiled with GCC using typical compiler flags. Basic operations like integer arithmetic are directly mapped to C constructs. As a result, the recommendations in [Chapter 1, The C Programming Language](#) apply.

In particular, the following Vala language constructs can result in undefined behavior at run time:

- Integer arithmetic, as described in [Section 1.1.3, “Recommendations for integer arithmetic”](#).
- Pointer arithmetic, string subscripting and the **substring** method on strings (the **string** class in the **glib-2.0** package) are not range-checked. It is the responsibility of the calling code to ensure that the arguments being passed are valid. This applies even to cases (like **substring**) where the implementation would have range information to check the validity of indexes. See [Section 1.1.2, “Recommendations for pointers and array handling”](#).
- Similarly, Vala only performs garbage collection (through reference counting) for **GObject** values. For plain C pointers (such as strings), the programmer has to ensure that storage is deallocated once it is no longer needed (to avoid memory leaks), and that storage is not being deallocated while it is still being used (see [Section 1.3.1.1, “Use-after-free errors”](#)).

Part II. Specific Programming Tasks

Library Design

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

8.1. State management

8.1.1. 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.)

8.1.2. 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.

8.2. 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 8.3, “Callbacks”](#) for some of the challenges involved.

8.3. 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.

8.4. Process attributes

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

- environment variables (see [Section 12.3.1, “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 inherited 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 implement 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.

9.1. 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 9.3, “Dealing with the `select` limit](#)”), and the kernel resources are not freed. Therefore, it is important to close all descriptors at the earliest point in time possible, but not earlier.

9.1.1. 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.

9.1.2. Closing descriptors and race conditions

Unlike process IDs, which are recycled 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 conditions 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, SHUT_RDWR)` 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.

9.1.3. 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.

9.2. 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 approximation because it is possible to create file descriptors outside this range easily (see [Section 9.3, “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.

9.3. 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. Replacing the `select` function is the recommended approach.

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 11, Temporary files](#).

10.1. 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 10.2, “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 rewind. 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.

10.2. 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 10.2, “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.

10.3. 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.

10.4. 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.

10.5. 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 are safe with a shared directory for temporary files. General file system manipulation is treated in a separate chapter, [Chapter 10, 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 12.3.1, “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.

11.1. 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**.)

11.2. Named temporary files

The `mkstemp` 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 `mkstemp`.) For determining the directory part of the file name pattern, see [Section 11.1, “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 `mkstemp` multiple times. Do not create additional file names derived from the

name provided by a previous `mkstemp` call. However, it is safe to close the descriptor returned by `mkstemp` 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 `mkstemp` 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 11.1, “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.

11.3. 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.

11.4. 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 11.1, “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 temporary 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.

11.5. Compensating for unsafe file creation

There are two ways to make a function or program which expects a file name safe for use with temporary files. See [Section 12.1, “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 11.4, “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

12.1. 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 9.2, “Preventing file descriptor leaks to child processes”](#).

12.1.1. 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 a secure manner (see [Section 12.3.1, “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.

12.1.2. 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 of `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.

12.1.3. 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 `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 inherited 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 12.1.5, “Passing secrets to subprocesses”](#).)
- All other environment variables should be dropped. Names for new environment variables should not be accepted from untrusted sources.

12.1.4. Robust argument list processing

When invoking a program, it is sometimes necessary to include data from untrusted sources. Such data should be checked against embedded **`NUL`** characters because the system APIs will silently 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.

12.1.5. 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 12.1.3, “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.

12.2. 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 `waitid`, and hope that the status is not collected by an event loop first.

12.3. 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.

12.3.1. 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 12.1, “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 been 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 12.1. 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
```

12.4. 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 be 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 processes 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.

12.5. 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.

12.6. 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.

13.1. 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.

13.2. 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.

In new datagram-oriented protocols, unique numbers such as sequence numbers or identifiers for fragment reassembly (see [Section 13.3, “Fragmentation”](#)) should be at least 64 bits large, and really should not be smaller than 32 bits in size. Protocols should not permit fragments with overlapping contents.

13.3. Fragmentation

Some serialization formats use frames or protocol data units (PDUs) on lower levels which are smaller than the PDUs on higher levels. With such an architecture, higher-level PDUs may have to be *fragmented* into smaller frames during serialization, and frames may need *reassembly* into large PDUs during deserialization.

Serialization formats may use conceptually similar structures for completely different purposes, for example storing multiple layers and color channels in a single image file.

When fragmenting PDUs, establish a reasonable lower bound for the size of individual fragments (as large as possible—limits as low as one or even zero can add substantial overhead). Avoid fragmentation if at all possible, and try to obtain the maximum acceptable fragment length from a trusted data source.

When implementing reassembly, consider the following aspects.

- Avoid allocating significant amount of resources without proper authentication. Allocate memory for the unfragmented PDU as more and more fragments are encountered, and not based on the initially advertised unfragmented PDU size, unless there is a sufficiently low limit on the unfragmented PDU size, so that over-allocation cannot lead to performance problems.
- Reassembly queues on top of datagram-oriented transports should be bounded, both in the combined size of the arrived partial PDUs waiting for reassembly, and the total number of partially reassembled fragments. The latter limit helps to reduce the risk of accidental reassembly of

unrelated fragments, as it can happen with small fragment IDs (see [Section 13.3.1, “Fragment IDs”](#)). It also guards to some extent against deliberate injection of fragments, by guessing fragment IDs.

- Carefully keep track of which bytes in the unfragmented PDU have been covered by fragments so far. If message reordering is a concern, the most straightforward data structure for this is an array of bits, with one bit for every byte (or other atomic unit) in the unfragmented PDU. Complete reassembly can be determined by increasing a counter of set bits in the bit array as the bit array is updated, taking overlapping fragments into consideration.
- Reject overlapping fragments (that is, multiple fragments which provide data at the same offset of the PDU being fragmented), unless the protocol explicitly requires accepting overlapping fragments. The bit array used for tracking already arrived bytes can be used for this purpose.
- Check for conflicting values of unfragmented PDU lengths (if this length information is part of every fragment) and reject fragments which are inconsistent.
- Validate fragment lengths and offsets of individual fragments against the unfragmented PDU length (if they are present). Check that the last byte in the fragment does not lie after the end of the unfragmented PDU. Avoid integer overflows in these computations (see [Section 1.1.3, “Recommendations for integer arithmetic”](#)).

13.3.1. Fragment IDs

If the underlying transport is datagram-oriented (so that PDUs can be reordered, duplicated or be lost, like with UDP), fragment reassembly needs to take into account endpoint addresses of the communication channel, and there has to be some sort of fragment ID which identifies the individual fragments as part of a larger PDU. In addition, the fragmentation protocol will typically involve fragment offsets and fragment lengths, as mentioned above.

If the transport may be subject to blind PDU injection (again, like UDP), the fragment ID must be generated randomly. If the fragment ID is 64 bit or larger (strongly recommended), it can be generated in a completely random fashion for most traffic volumes. If it is less than 64 bits large (so that accidental collisions can happen if a lot of PDUs are transmitted), the fragment ID should be incremented sequentially from a starting value. The starting value should be derived using a HMAC-like construction from the endpoint addresses, using a long-lived random key. This construction ensures that despite the limited range of the ID, accidental collisions are as unlikely as possible. (This will not work reliably with really short fragment IDs, such as the 16 bit IDs used by the Internet Protocol.)

13.4. 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, and wrappers such as *shelve*
- Perl's *Storable* package
- Java serialization (`java.io.ObjectInputStream`), even if encoded in other formats (as with `java.beans.XMLDecoder`)

- 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.

In general, 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. JSON-based formats can still turn out risky if they serve as an encoding form for any of the serialization frameworks listed above.

13.5. XML serialization

13.5.1. 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 definition:

```
<!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.

13.5.2. 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.

13.5.3. 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 turned 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.

13.5.4. 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.

13.5.5. 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 13.1, “Disabling XML entity processing with Expat”](#). Expat does not perform any validation, so there are no problems related to that.

Example 13.1. 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 13.2, “Creating an Expat XML parser”](#)).

Example 13.2. 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 altogether, using a suitable **XML_StartDoctypeDeclHandler** handler installed with **XML_SetDoctypeDeclHandler**.

13.5.6. Using Qt for XML parsing

The XML component of Qt, QtXml, does not resolve external IDs by default, so it is not required to prevent such resolution. Internal entities are processed, though. To change that, a custom **QXmlDeclHandler** and **QXmlSimpleReader** subclasses are needed. It is not possible to use the **QDomDocument::setContent(const QByteArray &)** convenience methods.

Example 13.3, “A QtXml entity handler which blocks entity processing” shows an entity handler which always returns errors, causing parsing to stop when encountering entity declarations.

Example 13.3. A QtXml entity handler which blocks entity processing

```
class NoEntityHandler : public QXmlDeclHandler {
public:
    bool attributeDecl(const QString&, const QString&, const QString&,
                      const QString&, const QString&);
    bool internalEntityDecl(const QString&, const QString&);
    bool externalEntityDecl(const QString&, const QString&,
                           const QString&);
    QString errorString() const;
};

bool
NoEntityHandler::attributeDecl(
    const QString&, const QString&, const QString&, const QString&,
    const QString&)
{
    return false;
}

bool
NoEntityHandler::internalEntityDecl(const QString&, const QString&)
{
    return false;
}

bool
NoEntityHandler::externalEntityDecl(const QString&, const QString&, const
    QString&)
{
    return false;
}
```

```
}

QString
NoEntityHandler::errorString() const
{
    return "XML declaration not permitted";
}
```

This handler is used in the custom **QXmlReader** subclass in [Example 13.4, “A QtXml XML reader which blocks entity processing”](#). Some parts of QtXml will call the `setDeclHandler(QXmlDeclHandler *)` method. Consequently, we prevent overriding our custom handler by providing a definition of this method which does nothing. In the constructor, we activate namespace processing; this part may need adjusting.

Example 13.4. A QtXml XML reader which blocks entity processing

```
class NoEntityReader : public QXmlSimpleReader {
    NoEntityHandler handler;
public:
    NoEntityReader();
    void setDeclHandler(QXmlDeclHandler *);
};

NoEntityReader::NoEntityReader()
{
    QXmlSimpleReader::setDeclHandler(&handler);
    setFeature("http://xml.org/sax/features/namespaces", true);
    setFeature("http://xml.org/sax/features/namespace-prefixes", false);
}

void
NoEntityReader::setDeclHandler(QXmlDeclHandler *)
{
    // Ignore the handler which was passed in.
}
```

Our **NoEntityReader** class can be used with one of the overloaded `QDomDocument::setContent` methods. [Example 13.5, “Parsing an XML document with QDomDocument, without entity expansion”](#) shows how the **buffer** object (of type **QByteArray**) is wrapped as a **QXmlInputSource**. After calling the `setContent` method, you should check the return value and report any error.

Example 13.5. Parsing an XML document with QDomDocument, without entity expansion

```
NoEntityReader reader;
QBuffer buffer(&data);
buffer.open(QIODevice::ReadOnly);
QXmlInputSource source(&buffer);
QDomDocument doc;
QString errorMsg;
int errorLine;
int errorColumn;
bool okay = doc.setContent
    (&source, &reader, &errorMsg, &errorLine, &errorColumn);
```


13.5.7. 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 13.5.2, “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 13.6. 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 13.7. 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 13.8, “Java imports for OpenJDK XML parsing”](#) shows the imports used by the examples.

Example 13.8. 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;
```

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

This approach produces a **org.w3c.dom.Document** object from an input stream. [Example 13.9](#), “*DOM-based XML parsing in OpenJDK*” use the data from the **java.io.InputStream** instance in the **inputStream** variable.

Example 13.9. 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 13.6](#), “*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.

13.5.7.2. XML Schema validation in OpenJDK

[Example 13.10](#), “*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 13.10. 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 13.7, “Helper class to prevent schema resolution in OpenJDK”](#).

If you need to validate a document against an XML schema, use the code in [Example 13.9, “DOM-based XML parsing in OpenJDK”](#) to create the document, but do not enable validation at this point. Then use [Example 13.11, “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 13.11. 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));
```

13.5.7.3. Other XML parsers in OpenJDK

OpenJDK contains additional XML parsing and processing facilities. Some of them are insecure.

The class `java.beans.XMLDecoder` acts as a bridge between the Java object serialization format and XML. It is close to impossible to securely deserialize Java objects in this format from untrusted inputs, so its use is not recommended, as with the Java object serialization format itself. See [Section 13.4, “Library support for deserialization”](#).

13.6. 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

14.1. Primitives

Choosing 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 17, Transport Layer Security](#)).

14.2. Randomness

The following facilities can be used to generate unpredictable and non-repeating values. When these functions are used without special safeguards, each individual random 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.

RPM packaging

This chapter deals with security-related concerns around RPM packaging. It has to be read in conjunction with distribution-specific packaging guidelines.

15.1. Generating X.509 self-signed certificates during installation

Some applications need X.509 certificates for authentication purposes. For example, a single private/public key pair could be used to define cluster membership, enabling authentication and encryption of all intra-cluster communication. (Lack of certification from a CA matters less in such a context.) For such use, generating the key pair at package installation time when preparing system images for use in the cluster is reasonable. For other use cases, it is necessary to generate the key pair before the service is started for the first time, see [Section 15.2, “Generating X.509 self-signed certificates before service start”](#).



Important

The way the key is generated may not be suitable for key material of critical value. (**openssl genrsa** uses, but does not require, entropy from a physical source of randomness, among other things.) Such keys should be stored in a hardware security module if possible, and generated from random bits reserved for this purpose derived from a non-deterministic physical source.

In the spec file, we define two RPM variables which contain the names of the files used to store the private and public key, and the user name for the service:

```
# Name of the user owning the file with the private key
%define tlsuser %{name}
# Name of the directory which contains the key and certificate files
%define tlsdir %{_sysconfdir}/%{name}
%define tlskey %{tlsdir}/%{name}.key
%define tlscert %{tlsdir}/%{name}.cert
```

These variables likely need adjustment based on the needs of the package.

Typically, the file with the private key needs to be owned by the system user which needs to read it, **%{tlsuser}** (not **root**). In order to avoid races, if the *directory* **%{tlsdir}** is owned by the services user, you should use the code in [Example 15.1, “Creating a key pair in a user-owned directory”](#). The invocation of **su** with the **-s /bin/bash** argument is necessary in case the login shell for the user has been disabled.

Example 15.1. Creating a key pair in a user-owned directory

```
%post
if [ $1 -eq 1 ] ; then
  if ! test -e %{tlskey} ; then
    su -s /bin/bash \
      -c "umask 077 && openssl genrsa -out %{tlskey} 2048 2>/dev/null" \
      %{tlsuser}
```

```

fi
if ! test -e %{tlskey} ; then
    cn="Automatically generated certificate for the %{tlsuser} service"
    req_args="-key %{tlskey} -out %{tlskey} -days 7305 -subj \"/CN=$cn/"
    su -s /bin/bash \
        -c "openssl req -new -x509 -extensions usr_cert $req_args" \
        %{tlsuser}
fi
fi

%files
%dir %attr(0755,%{tlsuser},%{tlsuser}) %{tlsdir}
%ghost %attr(0600,%{tlsuser},%{tlsuser}) %config(noreplace) %{tlskey}
%ghost %attr(0644,%{tlsuser},%{tlsuser}) %config(noreplace) %{tlskey}

```

The files containing the key material are marked as ghost configuration files. This ensures that they are tracked in the RPM database as associated with the package, but RPM will not create them when the package is installed and not verify their contents (the **%ghost**), or delete the files when the package is uninstalled (the **%config(noreplace)** part).

If the *directory* **%{tlsdir}** is owned by **root**, use the code in [Example 15.2, “Creating a key pair in a root-owned directory”](#).

Example 15.2. Creating a key pair in a root-owned directory

```

%post
if [ $1 -eq 1 ] ; then
    if ! test -e %{tlskey} ; then
        (umask 077 && openssl genrsa -out %{tlskey} 2048 >>/dev/null)
        chown %{tlsuser} %{tlskey}
    fi
    if ! test -e %{tlskey} ; then
        cn="Automatically generated certificate for the %{tlsuser} service"
        openssl req -new -x509 -extensions usr_cert \
            -key %{tlskey} -out %{tlskey} -days 7305 -subj \"/CN=$cn/"
    fi
fi

%files
%dir %attr(0755,root,root) %{tlsdir}
%ghost %attr(0600,%{tlsuser},%{tlsuser}) %config(noreplace) %{tlskey}
%ghost %attr(0644,root,root) %config(noreplace) %{tlskey}

```

In order for this to work, the package which generates the keys must require the **openssl** package. If the user which owns the key file is generated by a different package, the package generating the certificate must specify a **Requires(pre)**: on the package which creates the user. This ensures that the user account will exist when it is needed for the **su** or **chmod** invocation.

15.2. Generating X.509 self-signed certificates before service start

An alternative way to automatically provide an X.509 key pair is to create it just before the service is started for the first time. This ensures that installation images which are created from installed RPM packages receive different key material. Creating the key pair at package installation time (see [Section 15.1, “Generating X.509 self-signed certificates during installation”](#)) would put the key into the image, which may or may not make sense.



Important

The caveats about the way the key is generated in [Section 15.1, “Generating X.509 self-signed certificates during installation”](#) apply to this procedure as well.

Generating key material before service start may happen very early during boot, when the kernel randomness pool has not yet been initialized. Currently, the only way to check for the initialization is to look for the kernel message **random: nonblocking pool is initialized**. In theory, it is also possible to read from `/dev/random` while generating the key material (instead of `/dev/urandom`), but this can block not just during the boot process, but also much later at run time, and generally results in a poor user experience.

Part III. Implementing Security Features

Authentication and Authorization

16.1. 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 expected by 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 17, Transport Layer Security](#)) is the recommended way for securing connections over untrusted networks.

If the server port number is 1024 or 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.

16.2. Host-based authentication

Host-based authentication uses access control lists (ACLs) to accept or deny requests from clients. This 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 trusts 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.

16.3. 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_PEERCREC** (Linux) or **LOCAL_PEERCREC** (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.

16.4. 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 stack, 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 protect integrity and confidentiality while data is transferred over an untrusted network connection, and to identify the endpoint.

17.1. 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 17.1. 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 12.6](#), “*fork as a primitive for parallelism*”).

17.1.1. 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 17.2, “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 17.2. 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 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.

OpenSSL command-line commands, such as **openssl genrsa**, do not ensure that physical entropy is used for key generation—they obtain entropy from **/dev/urandom** and other sources, but not from **/dev/random**. This can result in weak keys if the system lacks a proper entropy source (e.g., a virtual machine with solid state storage). Depending on local policies, keys generated by these OpenSSL tools should not be used in high-value, critical functions.

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.

17.1.2. 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.

17.1.3. 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`.

17.1.4. 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.

17.2. 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.

17.2.1. 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 17.3, “OpenSSL library initialization”](#)).

Example 17.3. 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 17.4, “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 17.4. OpenSSL client context creation

```
// Configure a client connection context. Send a handshake 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 17.1, “Deactivating the TCP Nagle algorithm”](#)), the actual connection object needs to be created, as show in [Example 17.4, “OpenSSL client context creation”](#). If the handshake started by `SSL_connect` fails, the `ssl_print_error_and_exit` function from [Example 17.2, “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 17.5. 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 *peer_cert = SSL_get_peer_certificate(ssl);
if (peer_cert == NULL) {
    fprintf(stderr, "peer certificate missing");
    exit(1);
}

// Check the certificate verification result. Allow an explicit
// certificate validation override in case verification fails.
int verify_status = SSL_get_verify_result(ssl);
if (verify_status != X509_V_OK && !certificate_validity_override(peer_cert)) {
    fprintf(stderr, "SSL_connect: verify result: %s\n",
        X509_verify_cert_error_string(verify_status));
    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(peer_cert, (const unsigned char *)host, strlen(host),
    0) != 1
    && !certificate_host_name_override(peer_cert, host)) {
    fprintf(stderr, "SSL certificate does not match host name\n");
    exit(1);
}

X509_free(peer_cert);

```

The connection object can be used for sending and receiving data, as in [Example 17.6, “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 17.6. 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 17.7, “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 it is necessary to explicitly close the underlying socket after the connection object has been freed.

Example 17.7. 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 17.8, “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 17.8. Closing an OpenSSL connection in an orderly fashion

```
SSL_CTX_free(ctx);
```

17.2.2. 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 17.1.2, “GNUTLS Pitfalls”](#) for additional aspects of initialization.

Before setting up TLS connections, a credentials object has to be allocated and initialized with the set of trusted root CAs ([Example 17.9, “Initializing a GNUTLS credentials structure”](#)).

Example 17.9. 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 17.1, “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 17.10, “Establishing a TLS client connection using GNUTLS”](#).

Example 17.10. 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\"", 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 17.11, “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 17.11. 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 17.12, “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_cert_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 17.12. 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_cert_t cert;
ret = gnutls_x509_cert_init(&cert);
if (ret != GNUTLS_E_SUCCESS) {
    fprintf(stderr, "error: gnutls_x509_cert_init: %s\n",
        gnutls_strerror(ret));
    exit(1);
}
// The peer certificate is the first certificate in the list.
ret = gnutls_x509_cert_import(cert, certs, GNUTLS_X509_FMT_DER);
if (ret != GNUTLS_E_SUCCESS) {
    fprintf(stderr, "error: gnutls_x509_cert_import: %s\n",
        gnutls_strerror(ret));
    exit(1);
}
ret = gnutls_x509_cert_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_cert_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 17.13, “Using a GNUTLS session”](#).

Example 17.13. 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 17.14, “Using a GNUTLS session”](#)).

Example 17.14. 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);
```

17.2.3. 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 17.15, “Setting up an `SSLContext` for OpenJDK TLS clients”](#).

Example 17.15. 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 17.16, “Setting up **SSLParameters** for TLS use with OpenJDK”](#)). Like the context, these parameters can be reused for multiple TLS connections.

Example 17.16. Setting up **SSLParameters** for TLS use with OpenJDK

```
// Prepare TLS parameters. These have to be 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 17.17, “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 17.17. 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 17.18, “Using a TLS client socket in OpenJDK”*.

Example 17.18. 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);
```

17.2.3.1. 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 17.19, “A customer trust manager for OpenJDK TLS clients”*, the server certificate is identified by its SHA-256 hash.

Example 17.19. 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 17.20, “Using a custom TLS trust manager with OpenJDK”](#).

Example 17.20. 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.

17.2.4. Implementing TLS Clients With NSS

The following code shows how to implement a simple TLS client using NSS. These instructions apply to NSS version 3.14 and later. Versions before 3.14 need different initialization code.

Keep in mind that the error handling needs to be improved before the code can be used in production.

Using NSS needs several header files, as shown in [Example 17.21, “Include files for NSS”](#).

Example 17.21. 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 shown in [Example 17.22, “Initializing the NSS library”](#). This initialization procedure overrides global state. We only call `NSS_SetDomesticPolicy` if there are no strong ciphers available, assuming that it has already been called otherwise. This avoids overriding the process-wide cipher suite policy unnecessarily.

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 17.22. 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 PRCError 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, set the cipher suite 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 PRCError 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 PRCError 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 PRCError 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 17.23, “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. 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 17.23. 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 PRCError 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 PRCError 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 PRCError 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 PRCError err = PR_GetError();
        fprintf(stderr, "error: set SSL_ENABLE_DEFLATE error %d: %s\n",
            err, PR_ErrorToName(err));
        exit(1);
    }
}

// Allow overriding invalid certificate.
if (SSL_BadCertHook(model, bad_certificate, (char *)host) != SECSuccess) {
    const PRCError 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 PRCError 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 PRCError 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 PRCError err = PR_GetError();
    fprintf(stderr, "error: SSL_SetURL error %d: %s\n",
        err, PR_ErrorToName(err));
    exit(1);
}
if (SSL_ForceHandshake(nspr) != SECSuccess) {
    const PRCError 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 17.24, “Using NSS for sending and receiving data”](#) shows how to use the NSPR descriptor to communicate with the server.

Example 17.24. 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 PRCError 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 PRCError err = PR_GetError();
    fprintf(stderr, "error: PR_Read error %d: %s\n",
        err, PR_ErrorToName(err));
    exit(1);
}
}

```

[Example 17.25, “Closing NSS client connections”](#) shows how to close the connection.

Example 17.25. Closing NSS client connections

```

// Send close_notify alert.
if (PR_Shutdown(nspr, PR_SHUTDOWN_BOTH) != PR_SUCCESS) {
    const PRCError 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);
```

17.2.5. 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 17.1.1, “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 17.26, “Implementing TLS host name checking Python (without wildcard support)” shows how to implement certificate matching, using the parsed certificate returned by `getpeercert`.

Example 17.26. Implementing TLS host name checking Python (without wildcard support)

```
def check_host_name(peer_cert, 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 peer_cert:
        return False
    if peer_cert.has_key("subjectAltName"):
        for typ, val in peer_cert["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 peer_cert["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 17.27, “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 negotiate 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 17.27. 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 1.3-1 Mon Oct 13 2014

Florian Weimer fweimer@redhat.com

Go: Mention default value handling in deserialization
Shell: New chapter

Revision 1.2-1 Wed Jul 16 2014

Florian Weimer fweimer@redhat.com

C: Corrected the `strncat` example
C: Mention mixed signed/unsigned comparisons
C: Unsigned overflow checking example
C++: **operator new[]** has been fixed in GCC
C++: Additional material on **std::string**, iterators
OpenSSL: Mention **openssl genrsa** entropy issue
Packaging: X.509 key generation
Go, Vala: Add short chapters
Serialization: Notes on fragmentation and reassembly

Revision 1.1-1 Tue Aug 27 2013

Eric Christensen sparks@redhat.com

Add a chapter which covers some Java topics.
Deserialization: Warn about Java's `java.beans.XMLDecoder`.
C: Correct the advice on array allocation ([bug 995595](https://bugzilla.redhat.com/show_bug.cgi?id=995595)¹).
C: Add material on global variables.

Revision 1.0-1 Thu May 09 2013

Eric Christensen sparks@redhat.com

Added more C and C++ examples.
TLS Client NSS: Rely on NSS 3.14 cipher suite defaults.

Revision 0-1 Thu Mar 7 2013

Eric Christensen sparks@redhat.com

Initial publication.

¹ https://bugzilla.redhat.com/show_bug.cgi?id=995595

