Secure Coding C++

Basics

Secure coding
There are no «tricks» for secure programming. Use recommended techniques, e.g. those in this cheat sheet.

Coding standard
Have a coding standard and enforce it through automated checks using a static code analysis tool.
Consider the application of an industry standard such as MISRA C++:2008. Many static code analysis tools support the checking of various coding standards.

Secure defaults
Initialize variables using the most secure default value (e.g. default is «check failed»).

Secure flow
Implement a secure program flow (e.g. least privilege principle by default).

Make your own
Don't invent or implement any security-relevant algorithms on your own. Use standards implemented by specialists.

Integer Safety

Type safe C++ constructs
C++ constructs are type safe. Take advantage of the language support and don't use the unsafe C counterparts instead.
Avoid C library functions (or wrap them inside a secure C++ class implementation).

Correct types
Use the right type for the problem:
- bool for true/false checks
- unsigned integers where negative numbers have no meaning, e.g. indices, sizes, loop counters

Floats and doubles
Limit the use of floating-point formats to a minimum. Implicit and explicit type conversions to integer types bare precision risks. The same applies to overloaded operators (self-made cast implementations).

Explicit casts
Use explicit (instead of implicit) casts. The only safe conversion of an itegral value is to a wider type of the same signedness.

Check range
Explicitly or with strong types (e.g. range-checking setter methods). Verify range checking with unit tests.

Off-by-one errors
(Unit) test loops, access-by-index, etc. regarding off-by-one errors.

Strings

Secure string class
Use a class which has been implemented with a focus on security (and initialize its objects correctly).

Unbound copies
Pay special attention to unbound copying (e.g. input from a stream) which is a common string-related error source.

Off-by-one errors
Off-by-one errors often occur at array access or when making assumptions about the return value of a string function.

Null termination errors
Missing null termination is a common string-related error.

Truncation
Avoid data loss through truncation of an input string.

Containers

Arrays
Off-by-one errors are frequent at array allocation and access. Buffer overflow can result from incorrect array access. Prevent these errors by
- checking an integer before using it to index an array
- using a secure array class implementation

Data Protection

Visibility
Store critical data in local variables, not in global ones.

Pointers to sensitive code
Don't expose internal pointers to sensitive code. Instead copy the results to a location which is defined by the caller.

Sensitive data
Don't store sensitive information in a way that makes it easy to find. User names, passwords and cryptographic keys, for instance, should not be stored as plain strings.

Comparing sensitive data
If you have to store sensitive data that has to be compared, store it's hash instead or store it as another «one way function».
To verify the data, compare the stored hash with the hash of the input. An even more secure way to store sensitive data (e.g. passwords) is using salted hash functions.

Clean-up
Clean up sensitive information after use - this is not as simple as it sounds. An optimizing compiler might see that the data is not used anymore and will optimize the clearing code away. Use the "volatile" keyword (type qualifier) to make sure that the clearing code is executed.
Flash memory is not always really cleared. The quickest and surest way to clear persistent memory is to encrypt the content and only delete the key when the information must be cleared. This has the added advantage that it is impossible to tell from the contents which part of the memory is free.

Canaries
Protect the start and the end of the storage range for information that needs integrity protection with «canaries". Check the canaries before reading the information and act upon a mismatch.

Range specification
Use a specification for the validation of stored data (e.g. document the expected range using Doxygen). This allows the detection of instances of tampering with the stored data.

Three Types of Random Numbers

Nonces
Nonces have no meaning and are used once. A persistent counter may be used.

Random numbers
The requirement placed on a «normal» random number generator is the distribution of the numbers. Example usage: schedulers, random delay in protocols. A pseudo random generator (e.g. rand() of the cstdlib) will do in this case.
A seed (e.g. a call to srand()) can be used to initialize the pseudo random generator.

Cryptographically secure random numbers
Cryptographically secure random numbers must be impossible to guess. These random numbers are e.g. used in key generation. Cryptographically secure random numbers must be generated with dedicated hardware.

Error Handling

Error messages
Error messages on external interfaces of the application must not reveal more information than absolutely required. (Apply this especially to the release version. During development it makes debugging harder.)

Error Handling (cont.)

Error codes
If a function returns an error code, this error code must be processed.

Exceptions
There are reasons for and against using C++ exceptions. Secure code can be written with or without this C++ feature.

Design Principles

Input validation
- Verify that inputs contain what is actually expected (value ranges, valid characters…).
- Validate the input in the subsystem where its context is known.
- Use white lists approach.
- Do early checks on encapsulated input data.
- Code must only process data which is within the expected range.

Output sanitization
Ensure that the API always returns data within the documented range.

Least privilege
Minimal access rights for functions, threads, users!

Secure the weakest link
Implement security where it is needed – not where it is easy to implement!

Secure by default
Deliver software in a maximally safe configuration.

Defense in depth
Don't rely on a single protection mechanism.

Don't mix code and data
Data should be stored in non-executable memory. This protects it from the from execution of code which has been injected by buffer overflow. (HW/OS support needed.)

Security by obscurity
Don't rely on «secrets».

Design Patterns

Encryption
Check key length and encryption algorithm periodically against recommendations from authorities.

Single access point
One single access point to an application for all services is easier to protect.

Privileged core
Keep security-critical functionality separate from the rest of the application (separate thread or processor, needs memory management support from HW/OS).

Authentication
Use standard algorithms implemented by specialists.
Consider authentication means other than a password (e.g. certificate, biometric data…).

Session management
Remember that HTTP is stateless. For secure communication, the state of the connection must be known. Use standards to handle this problem, e.g. TLS.

User management
Connect to a trusted remote user management system (if available).

Access control
Must be considered early to integrate well with the design. Choose between MAC/DAC variants with varying characteristics, e.g. roles/ profiles, token, etc.

Concurrency

Single responsibility principle
The SRP states that every class should have a single responsibility. This responsibility should be entirely encapsulated by the class.
Keep your concurrency-related code separate from other code.

Limit shared data
Limit the access of data that may be shared to a minimum. This helps keep the management of such data feasible. One way to avoid sharing data is using a copy of the set of data in question.

Atomic access to shared data
If data must be shared, protect the access of shared data (e.g. with a mutex). This prevents unexpected behaviour and protects this data from leaking to other threads under race conditions.

Independent threads
Try to model data into independent subsets which can be processed by independent threads. This avoids synchronization problems.

Low Level Code

Memory size
Ensure that call stacks (and heap) are dimensioned correctly (big enough).

Memory management
Avoid buffer overflow programming errors by carefully controlling access to the memory, e.g by using more secure function equivalents to memcpy() and memmove(), e.g. memcpy_s() and memmove_s() or a class which has been implemented with a focus on security.

Input streams
Never use gets(). It is impossible to know how many characters gets() will read in advance. Use fgets() instead or a class which has been implemented with a focus on security.
Always do input validation for input streams.

C constructs
Do not use anything that could be accessed out of bound (e.g. C-style arrays, pointers in interfaces…).

Further

Side channels
Be aware of side channels. Side channels leak information about the application via other means, e.g. through timing, EM radiation or power consumption.
Example: Use a time-invariant string compare function for verifying a password. Otherwise an attacker can measure the return time to his tries to identify the password characters one by one.

Clean code
Clean, easy readable, maintainable code is a related to secure coding. Please see our bbv clean coding cheat sheet.

Practices | Smells

V 1.0

Author: Annina Blaas

bbv Software Services AG