



This is a living document. Download the most recent version at www.sans-ssi.org. Please send suggestions and comments to spa@sans.org

GSSP (GIAC Secure Software Programmer)

C Secure Coding Tasks, Skills and Knowledge

www.sans.org

April 2007

This document enumerates common C coding tasks and identifies rules, recommendations, and guidelines for accomplishing these tasks securely.

Task 1: C programs must interact securely with their environment. Programs must accept inputs from the environment and properly validate and process these inputs including command line arguments and environment variables. Programs must also be able to invoke external programs in a secure fashion.

01.1.1 Input Validation. The programmer must securely process inputs from all aspects of the environment, then correctly decode, canonicalize, and validate those inputs.

Understand and apply secure coding rules:

1. All input sources must be identified. Data from all untrusted sources must be fully specified and the data validated against these specifications. Ensure that all inputs are of the expected data type and are syntactically and semantically valid.
2. Use the strongest form of input validation possible for your program, for example, indirect selection from a menu of choices.
3. Assume that your client can be spoofed or modified in a client/server or three-tiered architecture.
4. Reject all input that is too short or too long for the program.
5. Set and enforce reasonable range restrictions. Reject all numeric input that is below the minimum valid value or above the maximum valid value.
6. Do not use functions that input character data and automatically convert the data if these functions do not properly handle all valid inputs.
7. Treat all environmental variables with suspicion. Do not make assumptions about the size, value, or existence of an environment variable.
8. Convert all input into canonical forms, and validate input after the input has been converted (including appropriate decoding). This applies to pathnames and filenames as well as URLs and other input strings that contain character sequences that can be subsequently converted to other sequences.
9. Ensure that each input is only decoded once.

Skill in:

1. Preventing vulnerabilities resulting from buffer overflows, path traversal, canonicalization errors, and missing XML validation.
2. Preventing spoofing attacks.

01.1.2 Data sanitization – The programmer must correctly sanitize, encode, and pass data to subsystems and external processes in a way that prevents attackers from directly influencing their behavior.

Understand and apply secure coding rules:

1. Prevent command injection attacks caused by passing user-controlled input to

complex subsystems such as command interpreters. Where user input is required for such an invocation, fully specify allowable forms of strings to be passed to the complex subsystem, and sanitize these strings against these specifications prior to invocation.
2. Escape all metacharacters, using built-in functions when available.
3. When interfacing with a relational database, reject all user input that contains SQL metacharacters that are not found in your white list. Escape any SQL metacharacters that your application needs to accept, such as single quotes in names. Always prepare SQL statements.
Skill in:
1. Preventing OS command injection, cross-site scripting (XSS), SQL injection and CRLF injection attacks.
01.1.3 Invocation – The programmer must invoke external programs by maintaining full control over the environment in which the program runs, ensuring that the correct program is invoked, and only using the expected arguments.
Understand and apply secure coding rules:
1. Sanitize the environment before invoking external programs. Consider purging the environment of all environmental variables except those that your program requires, and have these variables default to safe values.
2. Allow for the possibility of multiple environment variables with the same name, or that these variables might contain untrusted values.
3. Ensure that search paths cannot contain any locations that are not under your control or otherwise trusted.
4. Ensure that you have full control over all arguments to the component that you are invoking, stripping out argument separators and quotes.
5. Avoid invoking a general purpose command interpreter to execute a program when the functionality of the command interpreter is not required. This can be accomplished, for example, by using <code>exec()</code> instead of <code>system()</code> .
6. If invocation occurs through externally accessible resources such as named pipes, these resources should verify the integrity and origin of incoming requests.
7. Do not to spawn processes with excessive privileges.
8. Do not pass confidential data to program as command arguments as a user running the <code>ps</code> command may be able to view this data
Skill in:
1. Preventing vulnerabilities resulting from untrusted search paths and origin validation errors.
2. Preventing argument Injection attacks.
Task 2: C programmers must be able to manage dynamically allocated resources such as dynamically allocated memory.
01.2.1 Memory Management –The programmer must properly allocate, manage, and deallocate memory.
Understand and apply secure coding rules:
1. Only free memory allocated dynamically.
2. Avoid freeing memory multiple times by allocating and deallocating memory in the same module, at the same level of abstraction; and by techniques such as reference counting.
3. Do not access freed memory.
4. Avoid allocating zero bytes of memory.
5. Ensure that size arguments to memory allocation functions are valid.
6. Free dynamically allocated memory exactly once.
7. Set pointers to deallocated memory to null immediately after freeing the

memory.
8. Check potentially null pointers for the null value before accessing them.
9. Calculate buffer sizes correctly, allowing for the data type differences such as character width (char vs wchar_t) and the presence of terminating null bytes.
10. Keep track of the size of dynamically allocated memory to avoid buffer overflows.
11. Carefully evaluated rarely executed paths and error handling code for memory-related errors.
12. Establish and enforce resource quota to prevent denial-of-service attacks.
13. Do not leave sensitive information exposed in memory longer than required.
a. Clear or overwrite all sensitive information stored in memory that is no longer required, for example, before deallocation.
b. Use volatile pointers to prevent compiler optimizations from removing these memory wipes from the executable code.
c. Completely initialize all memory objects, especially those that might be serialized and transmitted to an untrusted endpoint. Do not realloc() memory containing sensitive information.
Skill in:
1. Preventing vulnerabilities resulting from buffer overflows, memory leaks, freeing the same memory multiple times, and writing to already freed memory.
2. Preventing heap inspection and denial-of-service attacks.
01.2.2 Stack Management - The programmer must understand how the stack is used to support program execution and how to properly secure it.
Understand and apply secure coding rules:
1. Programmers must understand what information is on allocated on the stack, and how to protect it.
2. Programmers must understand how to prevent the stack from unbounded growth.
Skill in:
1. Using the stack securely and efficiently.
Task 3: C programmers must be able to perform input/output from streams and files and interact with the operating system to manage the file system.
01.3.1 Formatted Output –Programmers must carefully control inputs to formatted output functions.
Understand and apply secure coding rules:
1. Do not pass user controlled input into the format parameter of formatted input/output functions such as printf(), fprintf(), and sprintf().
2. Ensure that all localization strings have the expected format specifiers.
3. For well-formed output such as records, ensure that the fields do not contain separator characters.
Skill in:
1. Preventing format string vulnerabilities as well as vulnerabilities resulting from buffer overflows.
2. Preventing memory inspection, information leaks, and denial-of-service attacks.
01.3.2 Stream and File IO — Programmers must create, modify, and delete the correct files and streams with appropriate privileges and permissions.
Understand and apply secure coding rules:
1. Drop process privileges before writing to a file that can be modified by a user. In general, perform all file I/O with the reduced privileges.
2. Canonicalize file names originating from untrusted sources.
3. Create files and directory structures with appropriate access permissions to prevent unintended access.

4. Test to see if the file already exists when creating a file.
5. Prefer functions that do not rely on file names for identification. Identify files using multiple file attributes, for example, by comparing file ownership or creation time.
6. Ensure that file descriptors are not leaked to untrusted processes.
7. Avoid time of check / time of use race conditions.
8. Employ directory restriction operations such as chroot() or jail() to restrict access to files.
9. Use absolute paths where possible for file names to prevent traversal attacks.
10. Avoid the use of functions such as gets() that do not constrain the size of the input.
11. Do not convert the value returned by a character input/output function to char if that value is going to be compared to EOF. Use feof() and ferror() to detect end-of-file and file errors.
12. When reading strings don't assume a newline character is read or even that character data has been read.
Skill in:
1. Preventing vulnerabilities resulting from race conditions (including TOCTOU) and insecure permissions.
2. Preventing path traversal attacks, symbolic link attacks, and hard link attacks.
01.3.3 Temporary Files —The programmer must properly create, protect, and delete temporary files.
Understand and apply secure coding rules:
1. Avoid creating temporary files in shared directories. If you absolutely must create a temporary file in a shared directory use mkstemp() or the TR 24731-1 tmpfile_s() function.
2. Avoid the use of functions such as tmpnam(), tempnam() and mktemp() that cause race conditions and functions like tmpfile() that may generate predictable names
Skill in:
1. Preventing vulnerabilities resulting from race conditions (including TOCTOU) and insecure permissions.
2. Preventing path traversal attacks, symbolic link attacks, and hard link attacks.
Task 4: C programmers must be able to employ specific security mechanisms in addition to the core functionality of the program.
01.4.1 Identification, Authorization & Authentication —The programmer must properly conduct authentication and prevent attackers from bypassing the authentication to gain access to restricted resources.
Understand and apply secure coding rules:
1. Have a basic understanding of the relative strengths of various types of authentication, including two-factor authentication, and their implications on usability.
2. Protect against replay and spoofing attacks.
3. Don't use hard-coded passwords or account names.
4. Use proper hashing to protect passwords.
5. Don't store passwords or keys in plaintext.
6. Employ standardized and known strong protocols and libraries for authentication.
7. Employ authentication commensurate to the value of the data being secured.
8. Where the security value of the data requiring authentication is unknown, allow selection from secure methods of authentication, and set the default to a type of authentication that balances strength with psychological acceptability.

9. Base access decisions on permission rather than exclusion.
10. Understand the role that permissions play in restricting access to resources, the risks associated with privileges, and the need for proper authorization.
11. Always verify the user's authorization for a particular resource. Do not base authorization decisions using data that the user can directly modify.
Skill in:
1. Preventing authentication bypass by alternative name attacks, alternative path/channel attacks, replay attacks, and spoofing attacks.
01.4.2 Privacy and encryption —The programmer must protect data at rest or in transit and correctly perform the appropriate encryption and decryption. The programmer must prevent sensitive information from leaking to untrusted parties. The programmer must understand the relative strengths and weaknesses of commonly used cryptographic schemes, including hashing.
Understand and apply secure coding rules:
1. Understand the role of encryption in protecting privacy.
2. Never invent your own encryption. Only use well-established schemes that have been subjected to exhaustive public evaluation by trained cryptographers.
3. Understand that randomness plays an important role in cryptographic features, as well as generating unique or unpredictable identifiers, such as user IDs, filenames, or session IDs. Ensure the functions you call generate cryptographic randomness, not statistical randomness.
4. When generating random data for cryptographic operations, gather data from as many sources as possible, append the data into one long string and use a cryptographically secure hash algorithm to hash the data to produce the final random value.
5. Where possible, use standard known interfaces to produce cryptographically random data (but do not use the C99 rand() function which produces predictable values).
6. Use sources with high entropy, but be able to handle when entropy is low. Depending on the technology being used, the source could block or produce numbers that are less random than required.
7. For pseudo-random number generators, use seeds that cannot be predicted by an outsider. Information such as process ID or system time are often predictable. Ensure that you are generating a sufficiently broad range of random numbers, to reduce the chances that they can be guessed using brute force attacks.
8. Avoid storing passwords or encryption keys. Instead, store hashes of the password or key so that provided values can be validated without requiring the storage of the original for comparison. Where password or encryption keys are required for recovery, store the keys on isolated media and use key escrow or splitting techniques if the keys are of a particularly sensitive nature.
9. Try to prevent your process from "dumping core" and writing an image to disk.
10. Don't provide feedback from error messages, logs, and other sources that provide implementation details or contain sensitive information such as passwords.
11. Avoid security through obscurity. Assume that any security mechanism that your software employs can be reversed engineered by the adversary.
12. Remove debug symbols from the production version of your software.
13. Make sure that incoming data or communications cannot be tampered with, from the source or in transit.
14. Assume that your client (or server) code can be modified or subverted to send data or perform actions that a legitimate process would not.

15. Know the limitations of typical schemes for protecting information.
16. Understand the privacy implications of data that you process, and protect it appropriately.
17. Before transferring data to another process, strip all sensitive data from areas that are not directly under the user's control, such as temporary fields.
Skill in:
1. Preventing vulnerabilities resulting from insufficient randomness, plaintext storage of sensitive information, weak encryption, reversible one-way hashes, missing required cryptographic steps, key management errors, insufficient entropy in PRNG, predictability, and a small space of random values
01.4.3 Secure designs —The programmer must be able to select or create secure designs and implement these correctly.
Understand and apply secure coding rules:
1. Programmers must understand the terminology and basic concepts behind the most common vulnerabilities and attacks to create secure designs and to help avoid implementation errors that could lead to vulnerabilities.
2. Run at the lowest privilege level possible. Perform privileged tasks early, the drop privileges as soon as possible, and follow the correct procedures in dropping them. Where feasible, split privileged code and other code into separate components, and carefully check access between these components. Consider designs that allow components that interact with untrusted data sources to run with reduced privileges. Identify all possible privilege chains, and ensure that multiple privileges cannot be combined to acquire an even greater privilege than you intend. Ensure that operations that drop privilege succeed and properly handle cases where they do not.
3. Avoid trust boundary violation occurs where a program blurs the line between what is trusted and what is untrusted. The most common way to make this mistake is to allow trusted and untrusted data, or data and control flow information, to commingle in the same data structure.
4. Log security relevant events such as login, logoff, and credential changes.
Skill in:
1. Developing, selecting, and implementing secure designs.
Task 5: C programmers must be able to deal with concurrency issues including issues arriving from concurrent <i>threads</i> , <i>processes</i> , and <i>tasks</i> .
01.5.1 Concurrency —The programmer must prevent execution ordering of concurrent flows that results in undesired behavior such as race conditions or deadlock.
Understand and apply secure coding rules:
1. Multithreaded programs must be designed carefully to avoid race conditions, deadlock, double-locking, forgetting to unlock, and priority failures such as priority inversion, infinite overtaking, and starvation. The existence of these problems can be induced or aggravated by an attacker.
2. Avoid race conditions by using atomic operations to access shared data. Where these are not possible, employ locking to serialize access to such data, whether it is in the form of files or shared memory. Ensure that concurrency resources, such as locks and shared memory, are only accessible to controlled processes and threads.
3. Avoid time-of-check to time-of-use (TOCTOU) defects, where the security of an operation is checked in one operation, then the operation is performed afterward, by using operating system access checks instead of designing your own access control (never use <code>access()</code>) and by performing file operations on open file descriptors instead of filenames (e.g., use <code>fchmod()</code> instead of <code>chmod()</code>).

4. Do not trust compilers to generate code that is thread-safe. Do not call non-reentrant library functions from within signal handlers.
5. Anticipate asynchronous events, such as an interrupted connection in the middle of data transfer.
Skill in:
1. Preventing vulnerabilities resulting from signal handler race conditions, time-of-check time-of-use (TOCTOU), deadlock, and resource exhaustion.
Task 6: C programmers must understand how to use built-in data types and more complex data types based upon these primitive types.
01.6.1 Null-terminated byte strings (NTBS) — The programmer must properly use null-terminated byte strings and know about safer alternatives.
Understand and apply secure coding rules:
1. Allocate adequate space when copying bounded strings.
2. Guarantee that all NTBS are null-terminated, and that there are no off-by-one errors that cause the null byte to be written to the wrong location, or overwritten by later code.
3. Do not inadvertently truncate strings, particularly while copying.
4. Use safe string libraries, such as CERT managed strings or StrSafe where possible.
5. Validate strings with carefully constructed parsers. Standard string-based formats, such as XML, can be parsed with standard libraries that are often better vetted than custom parsers.
6. Avoid common errors while using null-terminated byte strings
a. Avoid object copies through pointers if the objects are not necessarily of the same types.
b. Do not provide the length of a buffer in characters to a function that accepts the size of the buffer in bytes, and the other way around.
c. Where possible, use only those functions that stick to one convention to avoid confusion.
d. Avoid allowing user input to affect the termination condition of a loop
e. Do not compare strings by pointer value except where the "atom" pattern is being employed for strings.
f. When using standard C string library functions, always allocate sufficient space for the null-termination character.
Skill in:
1. Preventing vulnerabilities resulting from buffer overflows, null-termination errors, and string truncation errors.
01.6.2 Integer —The programmer must correctly manage integers by detecting and avoiding errors related to signedness, overflow, truncation, and range errors.
Understand and apply secure coding rules:
1. Avoid integer operations where the result is different from a result that would be obtained using infinite range integers. In particular, eliminate unexpected values resulting from integer overflow, truncation, and sign errors.
a. Use explicit width types (e.g. uint32_t, int16_t, DWORD, etc..) for code where the precision of a number must be a specific bit width.
b. Use the size_t or rsize_t types when representing the size of an object
c. Use unsigned types for values that should never be negative, such as lengths and most array indices and offsets.
d. Do not make assumptions about whether or not the char data type is signed.
e. Make sure conversions do not result in a loss of data or in misinterpreted data.

f. Use safe integer libraries, such as IntSafe or SafeInt, where available.
g. Ensure that arithmetic operations can not overflow or that the program detects and correctly handles such overflow.
Skill in:
1. Preventing vulnerabilities resulting from integer overflow, integer sign errors, integer truncation, and integer range errors.
01.6.3 Other data types –The programmer must correctly manage other data types such as arrays, floats, pointers, and objects.
Understand and apply secure coding rules:
1. Arrays
a. Be careful using the sizeof operator to determine the size of an array.
b. Guarantee that array indices are within the valid range.
c. Use consistent array notation across all source files.
d. Ensure size arguments for variable length array are in a valid range.
2. Floating point
a. Take granularity into account when comparing floating point values.
b. Do not use floating point variables as loop counters.
c. Prevent domain errors in math functions.
3. Pointers
a. Avoid casting integer values into pointers.
b. Avoid pointer arithmetic on pointers that do not point to parts of the same memory object.
c. Do not allow user inputs to specify function pointers.
4. Structures
a. Validate any offsets that are used within data structures.
b. Do not perform byte-by-byte comparisons between structures.
c. Do not assume the size of a structure is the sum of the of the sizes of its members.
Skill in:
1. Preventing vulnerabilities resulting from array indexing errors and buffer overflows.
Task 7: C programmers must correctly handle all error conditions.
01.7.1 Return Values —The programmer must handle all possible return values from each function call, including error and unusual conditions.
Understand and apply secure coding rules:
1. Check return values for all functions that may return error conditions unless you can demonstrate that the condition under which the function could fail can never happen, or failure is irrelevant to the program's correct, secure, and robust operation. Functions that intuitively can never fail, such as close(), can fail under certain circumstances.
2. If function failure is an unrecoverable error, at least wrap the function in a version that checks the return value and aborts upon error.
3. Always check the return value of malloc() and by nothrow invocations of the new operator. Ensure that you account for all possible return values.
Skill in:
1. Preventing vulnerabilities resulting from unchecked return values.
2. Preventing privilege escalation, symbolic link attacks and hard link attacks.
Task 8: C programmers must code correctly and use effective coding styles (this is a catch-all category).
01.8.1 Coding style — The programmer must follow conventions that make code understandable and maintainable, in a way that minimizes the chances of

inadvertently introducing vulnerabilities and improves the chances of discovering existing vulnerabilities and security flaws.
Understand and apply secure coding rules:
1. Use a consistent coding style and format.
2. Strive for logical completeness.
3. Use comments consistently and in a readable fashion.
4. Identifiers
a. Use visually distinct identifiers to eliminate errors resulting from incorrectly recognizing the spelling of an identifier during the development and review of code.
b. Do not define multiple identifiers that vary only with respect to one or more visually similar characters.
c. Try to make the initial portions of long identifiers unique for easier recognition.
d. Employ consistent naming conventions when naming variables, functions, and other named objects.
e. Try to make names explanatory without being overly long.
f. Adopt a strategy for short names, for example, "i" is always an "int", and "u" is always an "unsigned int".
g. Do not use the same variable name in two scopes where one scope is contained in another.
5. Use parentheses for precedence of operation.
6. Do not employ functions that are known to be dangerous from a security standpoint. Avoid functions that are known to be used erroneously. Either use safe versions of dangerous functions or rewrite the functions entirely using secure design principles. If a dangerous function must be used, then document its limitations and pay careful attention to the code using such functions to ensure that they are used in a manner that will not facilitate a security exploit.
7. Identify dead code during code reviews and remove it. This includes code that cannot be reached for any set of inputs but does not include code that catches "impossible" cases; this type of code is critical for robustness.
8. Remove unused debug code in the software. Alternately, enclose all optional debug code in conditional compilation structures and explicitly test for the use of the debug code through debug level flags.
9. Constants
a. Declare immutable values as rvalues or unmodifiable lvalues.
b. Const-qualify objects methods that do not modify the state of the object.
c. Declare data members private and use const when returning references to private attributes.
d. Do not cast away a const qualification.
10. Preprocessor
a. Use the preprocessor judiciously.
b. Prefer inline functions to macros.
c. Use parentheses around variable names within macros, and macro expansions must always be parenthesized.
11. Avoid implementation-defined behavior and/or document your assumptions.
12. Use standard types of fixed precision (e.g. int32_t) when precision requirements are known.
13. Write code that makes as few assumptions about the underlying system as possible. Where system-dependent operations are required, isolate such operations in system dependent modules.
14. Evaluate the use of security mechanisms provided by your compiler, including

buffer overflow protection feature.
Skill in:
1. Defining and applying coding styles that improve readability and understandability and decrease the chances of introducing new vulnerabilities or improve the chances of discovering existing vulnerabilities.
01.8.2 Code correctly —The programmer must write correct code, even in cases where incorrect code does not impact program operation in normal use, due to the associated risks.
Understand and apply secure coding rules:
1. Avoid errors of omission or addition where necessary characters are omitted, or characters are accidentally included, and the resulting code still compiles cleanly but behaves in an unexpected fashion. For example, do not inadvertently mix assignment and comparison operators within conditional expressions by adding or omitting a "=" character.
2. Don't take the sizeof a pointer to determine the size of a type.
3. Operands to the sizeof operator should not contain side effects.
4. The second operands of the logical AND and OR operators should not contain side effects.
5. Do not cast away a const qualification.
6. Do not depend on order of evaluation between sequence points or reference uninitialized variables.
7. Do not refer to an object outside of its lifetime.
8. All fetches and stores must be within the bounds of an existing object. This is of particular concern when fetching or storing an array element because these operations are largely unchecked.
9. Ensure that all variables are initialized before reading from them.
10. Ensure that your code does not make assumptions about the precision of numeric values unless the variables that store such values can be shown to support such precision.
11. Compile cleanly at high warning levels.
Skill in:
<ul style="list-style-type: none"> Preventing vulnerabilities resulting from initialization errors.

==end==