C security rules

The help and queries for C security analysis have now moved to the C++ queries space.

Overview of security queries

- Array offset used before range check - Accessing an array offset before checking the range means that the program may attempt to read beyond the end of a buffer
- Authentication bypass by spoofing - Authentication by checking that the peer's address matches a known IP or web address is unsafe as it is vulnerable to spoofing attacks.
- Bad check for overflow of integer addition - Checking for overflow of integer addition by comparing against one of the arguments of the addition does not work when the result of the addition is automatically promoted to a larger type.
- Badly bounded write - Buffer write operations with a length parameter that does not match the size of the destination buffer may overflow.
- Buffer not sufficient for string - A buffer allocated using 'malloc' may not have enough space for a string that is being copied into it. The operation can cause a buffer overrun. Make sure that the buffer contains enough room for the string (including the zero terminator).
- Call to memory access function may overflow buffer - Incorrect use of a function that accesses a memory buffer may read or write data past the end of that buffer.
- Cast between HRESULT and a Boolean type - Casting an HRESULT to/from a Boolean type and then using it in a test expression will yield an incorrect result because success (S_OK) in HRESULT is indicated by a value of 0.
- Cast from char* to wchar_t* - Casting a byte string to a wide-character string is likely to yield a string that is incorrectly terminated or aligned. This can lead to undefined behavior, including buffer overruns.
- CGI script vulnerable to cross-site scripting - Writing user input directly to a web page allows for a cross-site scripting vulnerability.
- Cleartext storage of sensitive information in an SQLite database - Storing sensitive information in a non-encrypted database can expose it to an attacker.
- Cleartext storage of sensitive information in buffer - Storing sensitive information in cleartext can expose it to an attacker.
- Cleartext storage of sensitive information in file - Storing sensitive information in cleartext can expose it to an attacker.
- Comparison of narrow type with wide type in loop condition - Comparisons between types of different widths in a loop condition can cause the loop to behave unexpectedly.
- Copy function using source size - Calling a copy operation with a size derived from the source buffer instead of the destination buffer may result in a buffer overflow.
- Cyclic lock order dependency - Locking mutexes in different orders in different threads can cause deadlock.
- Dangerous use of 'cin' - Using `cin` without specifying the length of the input may be dangerous.
- Exposure of system data to an unauthorized control sphere - Exposing system data or debugging information helps an adversary learn about the system and form an attack plan.
- File created without restricting permissions - Creating a file that is world-writable can allow an attacker to write to the file.
- Global variable may be used before initialization - Using an uninitialized variable may lead to undefined results.
- Inconsistent null check of pointer - A dereferenced pointer is not checked for nullness in this location, but it is checked in other locations. Dereferencing a null pointer leads to undefined results.
- Incorrect 'not' operator usage - Usage of a logical-not (!) operator as an operand for a bit-wise operation. This commonly indicates the usage of an incorrect operator instead of the bit-wise not (~) operator, also known as ones' complement operator.
- Infinite loop with unsatisfiable exit condition - A loop with an unsatisfiable exit condition could prevent the program from terminating, making it vulnerable to a denial of service attack.
- Initialization code not run - Not running initialization code may lead to unexpected behavior.

- Lock may not be released - A lock that is acquired one or more times without a matching number of unlocks may cause a deadlock.
- Memory is never freed - A function always returns before freeing memory that was allocated in the function. Freeing all memory allocated in the function before returning ties the lifetime of the memory blocks to that of the function call, making it easier to avoid and detect memory leaks.
- Memory may not be freed - A function may return before freeing memory that was allocated in the function. Freeing all memory allocated in the function before returning ties the lifetime of the memory blocks to that of the function call, making it easier to avoid and detect memory leaks.
- Mismatching new/free or malloc/delete - An object that was allocated with 'malloc' or 'new' is being freed using a mismatching 'free' or 'delete'.
- Multiplication result converted to larger type - A multiplication result that is converted to a larger type can be a sign that the result can overflow the type converted from.
- Mutex locked twice - Calling the lock method of a mutex twice in succession might cause a deadlock.
- Non-constant format string - Passing a non-constant 'format' string to a printf-like function can lead to a mismatch between the number of arguments defined by the 'format' and the number of arguments actually passed to the function. If the format string ultimately stems from an untrusted source, this can be used for exploits.
- No space for zero terminator - Allocating a buffer using 'malloc' without ensuring that there is always space for the entire string and a zero terminator can cause a buffer overrun.
- Not enough memory allocated for array of pointer type - Calling 'malloc', 'calloc' or 'realloc' without allocating enough memory to contain multiple instances of the type of the pointer may result in a buffer overflow
- Not enough memory allocated for pointer type - Calling 'malloc', 'calloc' or 'realloc' without allocating enough memory to contain an instance of the type of the pointer may result in a buffer overflow
- NULL application name with an unquoted path in call to CreateProcess - Calling a function of the CreateProcess* family of functions, where the path contains spaces, introduces a security vulnerability.
- Open descriptor may not be closed - Failing to close resources in the function that opened them makes it difficult to avoid and detect resource leaks.
- Open descriptor never closed - Functions that always return before closing the socket or file they opened leak resources.
- Open file is not closed - A function always returns before closing a file that was opened in the function. Closing resources in the same function that opened them ties the lifetime of the resource to that of the function call, making it easier to avoid and detect resource leaks.
- Open file may not be closed - A function may return before closing a file that was opened in the function. Closing resources in the same function that opened them ties the lifetime of the resource to that of the function call, making it easier to avoid and detect resource leaks.
- Overflow in uncontrolled allocation size - Allocating memory with a size controlled by an external user can result in integer overflow.
- Pointer offset used before it is checked - Accessing a pointer or array using an offset before checking if the value is positive may result in unexpected behavior.
- Pointer to stack object used as return value - Using a pointer to stack memory after the function has returned gives undefined results.
- Possibly wrong buffer size in string copy - Calling 'strncpy' with the size of the source buffer as the third argument may result in a buffer overflow.
- Potential improper null termination - Using a string that may not be null terminated as an argument to a string function can result in buffer overflow or buffer over-read.
- Potential integer arithmetic overflow - A user-controlled integer arithmetic expression that is not validated can cause overflows.
- Potentially overflowing call to snprintf - Using the return value from snprintf without proper checks can cause overflow.
- Potentially overrunning write - Buffer write operations that do not control the length of data written may overflow.
- Potentially overrunning write with float to string conversion - Buffer write operations that do not control the length of data written may overflow when floating point inputs take extreme values.
- Potentially uninitialized local variable - Reading from a local variable that has not been assigned to will typically yield garbage.

- Potentially unsafe call to strncat - Calling 'strncat' with the size of the destination buffer as the third argument may result in a buffer overflow.
- Potentially unsafe use of strcat - Using 'strcat' without checking the size of the source string may result in a buffer overflow
- Potential use after free - An allocated memory block is used after it has been freed. Behavior in such cases is undefined and can cause memory corruption.
- Returned pointer not checked - Dereferencing an untested value from a function that can return null may lead to undefined behavior.
- Self assignment check - Copy assignment operators should guard against self assignment; otherwise, self assignment is likely to cause memory corruption.
- Setting a DACL to NULL in a SECURITY_DESCRIPTOR - Setting a DACL to NULL in a SECURITY_DESCRIPTOR will result in an unprotected object. If the DACL that belongs to the security descriptor of an object is set to NULL, a null DACL is created. A null DACL grants full access to any user who requests it; normal security checking is not performed with respect to the object.
- Static array access may cause overflow - Exceeding the size of a static array during write or access operations may result in a buffer overflow.
- Suspicious 'sizeof' use - Taking 'sizeof' of an array parameter is often mistakenly thought to yield the size of the underlying array, but it always yields the machine pointer size.
- Suspicious add with sizeof - Explicitly scaled pointer arithmetic expressions can cause buffer overflow conditions if the offset is also implicitly scaled.
- Suspicious call to memset - Use of memset where the size argument is computed as the size of some non-struct type. When initializing a buffer, you should specify its size as <number of elements> * <size of one element> to ensure portability.
- Suspicious pointer scaling - Implicit scaling of pointer arithmetic expressions can cause buffer overflow conditions.
- Suspicious pointer scaling to char - Implicit scaling of pointer arithmetic expressions can cause buffer overflow conditions.
- Suspicious pointer scaling to void - Implicit scaling of pointer arithmetic expressions can cause buffer overflow conditions.
- Time-of-check time-of-use filesystem race condition - Separately checking the state of a file before operating on it may allow an attacker to modify the file between the two operations.
- Too few arguments to formatting function - Calling a printf-like function with too few arguments can be a source of security issues.
- Unbounded write - Buffer write operations that do not control the length of data written may overflow.
- Unchecked return value used as offset - Using a return value as a pointer offset without checking that the value is positive may lead to buffer overruns.
- Unclear validation of array index - Accessing an array without first checking that the index is within the bounds of the array can cause undefined behavior and can also be a security risk.
- Uncontrolled data in arithmetic expression - Arithmetic operations on uncontrolled data that is not validated can cause overflows.
- Uncontrolled data in SQL query - Including user-supplied data in a SQL query without neutralizing special elements can make code vulnerable to SQL Injection.
- Uncontrolled data used in OS command - Using user-supplied data in an OS command, without neutralizing special elements, can make code vulnerable to command injection.
- Uncontrolled data used in path expression - Accessing paths influenced by users can allow an attacker to access unexpected resources.
- Uncontrolled format string - Using externally-controlled format strings in printf-style functions can lead to buffer overflows or data representation problems.
- Uncontrolled format string (through global variable) - Using externally-controlled format strings in printf-style functions can lead to buffer overflows or data representation problems.
- Uncontrolled process operation - Using externally controlled strings in a process operation can allow an attacker to execute malicious commands.
- Unterminated variadic call - Calling a variadic function without a sentinel value may result in a buffer overflow if the function expects a specific value to terminate the argument list.
- Untrusted input for a condition - Using untrusted inputs in a statement that makes a security decision makes code vulnerable to attack.

- Upcast array used in pointer arithmetic - An array with elements of a derived struct type is cast to a pointer to the base type of the struct. If pointer arithmetic or an array dereference is done on the resulting pointer, it will use the width of the base type, leading to misaligned reads.
- Use of a broken or risky cryptographic algorithm - Using broken or weak cryptographic algorithms can allow an attacker to compromise security.
- Use of a version of OpenSSL with Heartbleed - Using an old version of OpenSSL can allow remote attackers to retrieve portions of memory.
- Use of extreme values in arithmetic expression - If a variable is assigned the maximum or minimum value for that variable's type and is then used in an arithmetic expression, this may result in an overflow.
- Use of inherently dangerous function - Using a library function that does not check buffer bounds requires the surrounding program to be very carefully written to avoid buffer overflows.
- Use of potentially dangerous function - Certain standard library functions are dangerous to call.
- User-controlled data in arithmetic expression - Arithmetic operations on user-controlled data that is not validated can cause overflows.
- User-controlled data may not be null terminated - String operations on user-controlled strings can result in buffer overflow or buffer over-read.
- Wrong type of arguments to formatting function - Calling a printf-like function with the wrong type of arguments causes unpredictable behavior.