C++ queries

This document contains the help for all the C++ built-in queries for the most recent release of the QL tools.

- 'new' object freed with 'delete[]' — An object that was allocated with 'new' is being freed using 'delete []'. Behavior in such cases is undefined and should be avoided. Use 'delete' instead.
- 'new[]' array freed with 'delete' — An array allocated with 'new[]' is being freed using 'delete'. Behavior in such cases is undefined and should be avoided. Use 'delete[]' when freeing arrays allocated with 'new[]'.
- Abstract namespaces — Finds namespaces that have an abstractness greater than 0.20.
- Accidental rethrow — When there is nothing to rethrow, attempting to rethrow an exception will terminate the program.
- Ambiguously signed bit-field member — Bit fields with integral types should have explicit signedness only. For example, use `unsigned int` rather than `int`. It is implementation specific whether an `int`-typed bit field is signed, so there could be unexpected sign extension or overflow.
- Array argument size mismatch — Finds function calls where the size of an array being passed is smaller than the array size of the declared parameter. This could lead to accesses to memory locations beyond the parameter's array bounds.
- 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
- Assignment where comparison was intended — The '=' operator may have been used accidentally, where '==' was intended.
- 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.
- Average cyclomatic complexity of files — The average cyclomatic complexity of the functions in a file.
- Avoid floats in for loops — Floating point variables should not be used as loop counters. For loops are best suited to simple increments and termination conditions; while loops are preferable for more complex uses.
- AV Rule 189 — The goto statement shall not be used.
- Bad check for oddness — Using "x % 2 == 1" to check whether x is odd does not work for negative numbers.
- 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.
- Block with too many statements — Blocks with too many consecutive statements are candidates for refactoring. Only complex statements are counted here (eg. for, while, switch ...). The top-level logic will be clearer if each complex statement is extracted to a function.
- Boolean value in bitwise operation — A Boolean value (i.e. something that has been coerced to have a value of either 0 or 1) is used in a bitwise operation. This commonly indicates missing parentheses or mistyping logical operators as bitwise operators.
- Branching condition always evaluates to same value — The condition of the branching statement always evaluates to the same value. This means that only one branch will ever be executed.
- 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 a function with one or more incompatible arguments — When the type of a function argument is not compatible with the type of the corresponding parameter, it may lead to unpredictable behavior.
- Call to alloca in a loop — Using alloca in a loop can lead to a stack overflow
- Call to function with extraneous arguments — A function call to a function passed more arguments than there are declared parameters of the function. This may indicate that the code does not follow the author's intent.

- Call to function with fewer arguments than declared parameters — A function call is passing fewer arguments than the number of declared parameters of the function. This may indicate that the code does not follow the author's intent. It is also a vulnerability, since the function is likely to operate on undefined data.
- 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.
- Catching by value — Catching an exception by value will create a copy of the thrown exception, thereby potentially slicing the original exception object.
- CGI script vulnerable to cross-site scripting — Writing user input directly to a web page allows for a cross-site scripting vulnerability.
- Classes with too many fields — Finds classes with many fields; they could probably be refactored by breaking them down into smaller classes, and using composition.
- Classes with too many source dependencies — Finds classes that depend on many other types; they could probably be refactored into smaller classes with fewer dependencies.
- 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.
- Commented-out code — Commented-out code makes the remaining code more difficult to read.
- Comment ratio per function — The ratio of comment lines to the total number of lines in a function.
- 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.
- Comparison result is always the same — When a comparison operation, such as x < y, always returns the same result, it means that the comparison is redundant and may mask a bug because a different check was intended.
- Comparison where assignment was intended — The '==' operator may have been used accidentally, where '=' was intended, resulting in a useless test.
- Comparison with canceling sub-expression — If the same sub-expression is added to both sides of a comparison, and there is no possibility of overflow or rounding, then the sub-expression is redundant and could be removed.
- Compilation time — Measures the amount of time (in milliseconds) spent compiling a C/C++ file, including time spent processing all files included by the pre-processor.
- Complex condition — Boolean expressions that are too deeply nested are hard to read and understand. Consider naming intermediate results as local variables.
- Complex functions — Finds functions which call too many other functions. Splitting these functions would increase maintainability and readability.
- Concrete namespaces — Finds namespaces that have an abstractness equal to 0.
- Constant return type — A 'const' modifier on a function return type is useless and should be removed for clarity.
- Constant return type on member — A 'const' modifier on a member function return type is useless. It is usually a typo or misunderstanding, since the syntax for a 'const' function is 'int foo() const', not 'const int foo()'.
- Constant string literals — A string literal must not be modified, as the result is undefined. To ensure this, only variables of type const char * or const char [] can hold string literals.
- Constructor with default arguments will be used as a copy constructor — Constructors with default arguments should not be signature-compatible with a copy constructor when their default arguments are taken into account.
- Conversion changes sign — Finds conversions from unsigned to signed.
- 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.
- Cyclic namespaces — Shows namespaces that cyclically depend on one another.
- Cyclomatic Complexity — Functions with high cyclomatic complexity. With increasing cyclomatic complexity there need to be more test cases that are necessary to achieve a complete branch coverage when testing this function.
- Cyclomatic complexity of functions — The Cyclomatic complexity (an indication of how many tests are necessary, based on the number of branching statements) per function.
- Dangerous system functions — The library functions abort, exit, getenv and system from library <stdlib.h> should not be used.
- Dangerous use of 'cin' — Using `cin` without specifying the length of the input may be dangerous.
- Dead code due to goto or break statement — A goto or break statement is followed by unreachable code.
- Declaration hides parameter — A local variable hides a parameter. This may be confusing. Consider renaming one of them.
- Declaration hides variable — A local variable hides another local variable from a surrounding scope. This may be confusing. Consider renaming one of the variables.
- Do not expose float representation — The underlying bit representation of floating point numbers should not be used in any way by the programmer. This leads to non-portable and hard to maintain code.
- Dubious NULL check — The address of a field (except the first) will never be NULL, so it is misleading, at best, to check for that case.
- Duplicated lines in files — The number of lines in a file, including code, comment and whitespace lines, which are duplicated in at least one other place.
- Duplicate function — There is another identical implementation of this function. Extract the code to a common file or superclass or delegate to improve sharing.
- Duplicate include guard — Using the same include guard macro in more than one header file may cause unexpected behavior from the compiler.
- Empty branch of conditional — An empty block after a conditional can be a sign of an omission and can decrease maintainability of the code. Such blocks should contain an explanatory comment to aid future maintainers.
- Equality test on floating-point values — Comparing results of floating-point computations with '==' or '!=' is likely to yield surprising results since floating-point computation does not follow the standard rules of algebra.
- Error-prone name of loop variable — The iteration variable of a nested loop should have a descriptive name: short names like i, j, or k can cause confusion except in very simple loops.
- Exception thrown in destructor — Throwing an exception from a destructor may cause immediate program termination.
- 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.
- Expression has no effect — A pure expression whose value is ignored is likely to be the result of a typo.
- Feature envy — A function that uses more functions and variables from another file than functions and variables from its own file. This function might be better placed in the other file, to avoid exposing internals of the file it depends on.
- File created without restricting permissions — Creating a file that is world-writable can allow an attacker to write to the file.
- FIXME comment — Comments containing 'FIXME' indicate that the code has known bugs.
- For loop variable changed in body — Numeric variables being used within a for loop for iteration counting should not be modified in the body of the loop. Reserve for loops for straightforward iterations, and use a while loop instead for more complex cases.
- Function declared in block — Functions should always be declared at file scope. It is confusing to declare a function at block scope, and the visibility of the function is not what would be expected.
- Function is never called — Unused functions may increase object size, decrease readability, and create the possibility of misuse.
- Function length — The average number of lines in functions in each file.
- Functions per file — The total number of functions in each file.

- Functions with too many parameters — Finds functions with many parameters; they could probably be refactored by wrapping parameters into a struct.
- Futile conditional — An if-statement with an empty then-branch and no else-branch may indicate that the code is incomplete.
- General statistics — Shows general statistics about the application.
- Global namespace classes — Finds classes that belong to no namespace.
- Global variable may be used before initialization — Using an uninitialized variable may lead to undefined results.
- Global variables — The total number of global variables in each file.
- Hiding identifiers — Identifiers in an inner scope should not use the same name as an identifier in an outer scope, and therefore hide that identifier.
- High afferent coupling namespaces — Finds namespaces that have an afferent coupling greater than 20.
- Implicit downcast from bitfield — A bitfield is implicitly downcast to a smaller integer type. This could lead to loss of upper bits of the bitfield.
- Inappropriate Intimacy — Two files share too much information about each other (accessing many operations or variables in both directions). It would be better to invert some of the dependencies to reduce the coupling between the two files.
- Include header files only — The #include pre-processor directive should only be used to include header files.
- Includes per file — The number of files directly included by this file using `#include`.
- Incoming dependencies per class — The number of classes that depend on a class.
- Incoming dependencies per file — The number of files that depend on a file.
- Inconsistent definition of copy constructor and assignment ('Rule of Two') — Classes that have an explicit copy constructor or copy assignment operator may behave inconsistently if they do not have both.
- Inconsistent direction of for loop — A for-loop iteration expression goes backward with respect of the initialization statement and condition expression.
- 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.
- Inconsistent nullness check — The result value of a function is often checked for nullness, but not always. Since the value is mostly checked, it is likely that the function can return null values in some cases, and omitting the check could crash the program.
- Inconsistent operation on return value — A function is called, and the same operation is usually performed on the return value - for example, free, delete, close etc. However, in some cases it is not performed. These unusual cases may indicate misuse of the API and could cause resource leaks.
- Inconsistent virtual inheritance — A base class shall not be both virtual and non-virtual in the same hierarchy.
- 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.
- Incorrect constructor delegation — A constructor in C++ cannot delegate part of the object initialization to another by calling it. This is likely to leave part of the object uninitialized.
- Indirect includes per file — The number of files included by the pre-processor - either directly by an `#include` directive, or indirectly (by being included by an included file).
- Indirect source includes per file — The number of source files included by the pre-processor - either directly by an `#include` directive, or indirectly (by being included by an included file). This metric excludes included files that aren't part of the main code base (like system headers).
- 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.
- Inheritance depth distribution — Shows the distribution of inheritance depth across all classes.
- Inheritance depth per class — The depth of a class in the inheritance hierarchy.
- Initialization code not run — Not running initialization code may lead to unexpected behavior.
- Irregular enum initialization — In an enumerator list, the = construct should not be used to explicitly initialize members other than the first, unless all items are explicitly initialized. An exception is the pattern to use the last element of an enumerator list to get the number of possible values.

- Lack of cohesion per class (LCOM-CK) — Lack of cohesion for a class as defined by Chidamber and Kemerer.
- Large object passed by value — An object larger than 64 bytes is passed by value to a function. Passing large objects by value unnecessarily use up scarce stack space, increase the cost of calling a function and can be a security risk. Use a const pointer to the object instead.
- Leaky catch — If an exception is allocated on the heap, then it should be deleted when caught.
- Lines of code in files — Measures the number of lines in a file that contain code (rather than lines that only contain comments or are blank)
- Lines of code per class — The number of lines of code in a class.
- Lines of code per function — Measures the number of lines in a function that contain code (rather than lines that only contain comments or are blank).
- Lines of commented-out code in files — The number of lines of commented-out code in a file.
- Lines of comments in files — Measures the number of lines which contain a comment or part of a comment (that is, which are part of a multi-line comment).
- Lines of comments per function — Measures the number of lines in a function that contain a comment or part of a comment (that is, which are part of a multi-line comment).
- Local variable address stored in non-local memory — Storing the address of a local variable in non-local memory can cause a dangling pointer bug if the address is used after the function returns.
- Local variable hides global variable — A local variable or parameter that hides a global variable of the same name. This may be confusing. Consider renaming one of the variables.
- Lock may not be released — A lock that is acquired one or more times without a matching number of unlocks may cause a deadlock.
- Logical expression could be simplified — When a logical expression can be easily simplified, there may be an opportunity to improve readability by doing so, or it may indicate that the code contains a typo.
- Long switch case — A switch statement with too much code in its cases can make the control flow hard to follow. Consider wrapping the code for each case in a function and just using the switch statement to invoke the appropriate function in each case.
- Lossy function result cast — Finds function calls whose result type is a floating point type, and which are casted to an integral type. Includes only expressions with implicit cast and excludes function calls to ceil, floor and round.
- Lossy pointer cast — A pointer type is converted to a smaller integer type. This may lead to loss of information in the variable and is highly non-portable.
- Magic numbers — 'Magic constants' should be avoided: if a nontrivial constant is used repeatedly, it should be encapsulated into a const variable or macro definition.
- Magic strings — 'Magic constants' should be avoided: if a nontrivial constant is used repeatedly, it should be encapsulated into a const variable or macro definition.
- 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'.
- Missing enum case in switch — A switch statement over an enum type is missing a case for some enum constant and does not have a default case. This may cause logic errors.
- Missing header guard — Header files should contain header guards (#defines to prevent the file from being included twice). This prevents errors and inefficiencies caused by repeated inclusion.
- Missing return statement — All functions that are not void should return a value on every exit path.
- Mostly duplicate class — More than 80% of the methods in this class are duplicated in another class. Create a common supertype to improve code sharing.
- Mostly duplicate file — There is another file that shares a lot of the code with this file. Merge the two files to improve maintainability.
- Mostly duplicate function — There is another function that shares a lot of the code with this one. Extract the code to a common file/superclass or delegate to improve sharing.

- Mostly similar file — There is another file that shares a lot of the code with this file. Notice that names of variables and types may have been changed. Merge the two files to improve maintainability.
- 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.
- Namespaces far from main line — Finds namespaces that do not have a good balance between abstractness and stability.
- Negation of unsigned value — The unary minus operator should not be applied to unsigned expressions - cast the expression to a signed type to avoid unexpected behavior.
- Nested loops with same variable — When a nested loop uses the same iteration variable as its outer loop, the behavior of the outer loop easily becomes difficult to understand as the inner loop will affect its control flow. It is likely to be a typo.
- Nesting depth — The maximum number of nested statements (for example, `if`, `for`, `while`, etc.). Blocks are not counted.
- 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.
- Non-empty call to function declared without parameters — A call to a function declared without parameters has arguments, which may indicate that the code does not follow the author's intent.
- Non-virtual destructor in base class — All base classes with a virtual function should define a virtual destructor. If an application attempts to delete a derived class object through a base class pointer, the result is undefined if the base class destructor is non-virtual.
- Non-zero value cast to pointer — A constant value other than zero is converted to a pointer type. This is a dangerous practice, since the meaning of the numerical value of pointers is platform dependent.
- No raw arrays in interfaces — Arrays should not be used in interfaces. Arrays degenerate to pointers when passed as parameters. This array decay problem has long been known to be a source of errors. Consider using std::vector or encapsulating the array in an Array class.
- 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
- No trivial switch statements — Using a switch statement when there are fewer than two non-default cases leads to unclear code.
- No virtual destructor — All base classes with a virtual function should define a virtual destructor. If an application attempts to delete a derived class object through a base class pointer, the result is undefined if the base class destructor is non-virtual.
- 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.
- Number of fields per class — The number of fields in a class.
- Number of function calls per function — The number of C/C++ function calls per function.
- Number of functions per class — The number of member functions in a class.
- Number of parameters per function — The number of formal parameters for each function.
- Number of statements per function — The number of C/C++ statements per function.
- Number of tests — The number of test methods defined in a file.
- Number of todo/fixme comments per file — The number of TODO or FIXME comments in a file.
- 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.
- Opposite operator definition — When two operators are opposites, both should be defined and one should be defined in terms of the other.
- Outgoing dependencies per class — The number of classes on which a class depends.
- Outgoing dependencies per file — The number of files that a file depends on.
- Overflow in uncontrolled allocation size — Allocating memory with a size controlled by an external user can result in integer overflow.
- Overloaded assignment does not return 'this' — An assignment operator should return a reference to *this. Both the standard library types and the built-in types behave in this manner.
- Parameters per function — The average number of parameters of functions in each file.
- Percentage of comments — The percentage of lines that contain comments.
- Percentage of complex code per class — The percentage of the code in a class that is part of a complex member function.
- 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.
- Poorly documented large function — Large functions that have no or almost no comments are likely to be too complex to understand and maintain. The larger a function is, the more problematic the lack of comments.
- Possible signed bit-field member — Failing to explicitly assign bit fields to unsigned integer or enumeration types may result in unexpected sign extension or overflow.
- 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.
- Public functions per file — The total number of public (non-static) functions in each file.
- Public global variables — The total number of global variables in each file with external (public) visibility.
- Redefined default parameter — An inherited default parameter shall never be redefined. Default values are bound statically which is confusing when combined with dynamically bound function calls.
- Redundant null check due to previous dereference — Checking a pointer for nullness after dereferencing it is likely to be a sign that either the check can be removed, or it should be moved before the dereference.
- Register variables — The register storage class specifier shall not be used. It is better and more portable to rely on the compiler to allocate registers automatically.

- Resource not released in destructor — All resources acquired by a class should be released by its destructor. Avoid the use of the 'open / close' pattern, since C++ constructors and destructors provide a safer way to handle resource acquisition and release. Best practice in C++ is to use the 'RAII' technique: constructors allocate resources and destructors free them.
- Response per class — The number of different member functions or constructors that can be executed by a class.
- Return c_str of local std::string — Returning the c_str of a locally allocated std::string could cause the program to crash or behave non-deterministically because the memory is deallocated when the std::string goes out of scope.
- Returned pointer not checked — Dereferencing an untested value from a function that can return null may lead to undefined behavior.
- Returning stack-allocated memory — A function returns a pointer to a stack-allocated region of memory. This memory is deallocated at the end of the function, which may lead the caller to dereference a dangling pointer.
- Return stack-allocated object — A function must not return a pointer or reference to a non-static local object.
- Return value of a function is ignored — A call to a function ignores its return value, but more than 80% of the total number of calls to the function check the return value. Check the return value of functions consistently, especially for functions like 'fread' or the 'scanf' functions that return the status of the operation.
- Rule of three — Classes that have an explicit destructor, copy constructor, or copy assignment operator may behave inconsistently if they do not have all three.
- Self assignment check — Copy assignment operators should guard against self assignment; otherwise, self assignment is likely to cause memory corruption.
- Self comparison — Comparing a variable to itself always produces the same result, unless the purpose is to check for integer overflow or floating point NaN.
- 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.
- Short-circuiting operator applied to flag — A short-circuiting logical operator is applied to what looks like a flag. This may be a typo for a bitwise operator.
- Short global name — Global variables should have descriptive names, to help document their use, avoid namespace pollution and reduce the risk of shadowing with local variables.
- Sign check of bitwise operation — Checking the sign of a bitwise operation often has surprising edge cases.
- Size of API per class — The number of public member functions in a public class.
- Sizeof with side effects — The sizeof operator should not be used on expressions that contain side effects. It is subtle whether the side effects will occur or not.
- Slicing — Assigning a non-reference instance of a derived type to a variable of the base type slices off all members added by the derived class, and can cause an unexpected state.
- Specialization per class — The extent to which a subclass refines the behavior of its superclasses.
- Stable namespaces — Finds namespaces that have an instability lower than 0.2.
- 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.
- Throwing pointers — Exceptions should be objects rather than pointers to objects.
- 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.
- TODO comment — Comments containing 'TODO' indicate that the code may be in an incomplete state.
- Too few arguments to formatting function — Calling a printf-like function with too few arguments can be a source of security issues.
- Too many arguments to formatting function — A printf-like function called with too many arguments will ignore the excess arguments and output less than might have been intended.
- 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 comparison precedence — Using comparisons as operands of other comparisons is unusual in itself, and most readers will require parentheses to be sure of the precedence.
- 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.
- Undisciplined multiple inheritance — Multiple inheritance should only be used in the following restricted form: n interfaces plus m private implementations, plus at most one protected implementation. Multiple inheritance can lead to complicated inheritance hierarchies that are difficult to comprehend and maintain.
- Undocumented API function — Functions used from outside the file they are declared in should be documented, as they are part of a public API. Without comments, modifying such functions is dangerous because callers easily come to rely on their exact implementation.
- Unsigned comparison to zero — An unsigned value is always non-negative, even if it has been assigned a negative number, so the comparison is redundant and may mask a bug because a different check was intended.
- Unstable namespaces — Finds namespaces that have an instability higher than 0.8.
- 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.
- Unused local variable — A local variable that is never called or accessed may be an indication that the code is incomplete or has a typo.
- Unused static function — A static function that is never called or accessed may be an indication that the code is incomplete or has a typo.
- Unused static variable — A static variable that is never accessed may be an indication that the code is incomplete or has a typo.
- 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.
- Usage of macros — The percentage of source lines in each file that contain use of macros.

- 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 dangerous function — Use of a standard library function that does not guard against buffer overflow.
- 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 goto — The goto statement can make the control flow of a function hard to understand, when used for purposes other than error handling.
- 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 integer where enum is preferred — Enumeration types should be used instead of integer types (and constants) to select from a limited series of choices.
- Use of potentially dangerous function — Use of a standard library function that is not thread-safe.
- Use of string copy function in a condition — The return value for strcpy, strncpy, or related string copy functions have no reserved return value to indicate an error. Using them in a condition is likely to be a logic error.
- 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.
- Variable is assigned a value that is never read — Assigning a value to a variable that is not used may indicate an error in the code.
- Variable not initialized before use — Using an uninitialized variable may lead to undefined results.
- Variable used in its own initializer — Loading from a variable in its own initializer may lead to undefined behavior.
- Virtual call from constructor or destructor — Virtual functions should not be invoked from a constructor or destructor of the same class. Confusingly, virtual functions are resolved statically (not dynamically) in constructors and destructors for the same class. The call should be made explicitly static by qualifying it using the scope resolution operator.
- Virtual call in constructor or destructor — Calling a virtual function from a constructor or destructor rarely has the intended effect. It is likely to either cause a bug or confuse readers.
- Wrong type of arguments to formatting function — Calling a printf-like function with the wrong type of arguments causes unpredictable behavior.

About the security queries

There are two query suites for C security analysis: `default` and `all`. For most projects we recommend that you run queries from the `default` suite. The `all` suite contains a few additional rules which perform points-to analysis and may timeout on some projects. These rules test for the following CWEs:

Data sources

Many queries rely on tracking the flow of data from sources that cannot be guaranteed to be safe. Where possible, the source of the potentially unsafe data is reported in the query violation message, indicating why that particular use of the data was considered dangerous. We describe here how some common sources of data are classified, in order to make it clear why these are considered to be potentially dangerous.

User Input

A common data source is data that comes from a user. This must be treated as untrusted unless it is validated, as a malicious user can send unexpected data that can have undesirable effects, such as allowing them to perform a SQL injection attack.

The sources that we consider user input are listed below:

1. **Getting the value of a system environment variable**
   The environment in which a program is run is often under the control of the user who runs the program.
2. **Getting a parameter from an HttpServletRequest**
   Spoofing a request can give the user control even over parameters which are not usually set by the user.
3. **Getting the query string from an HttpServletRequest**
   The query string comes from the URL, which is under the control of the user.
4. **Getting the header from an HttpServletRequest**
   Spoofing a request can give the user control over the header.
5. **Getting the value of a property from a Properties object**
   Properties objects are commonly written to and read from disk, which may be under the control of the user.
6. **Getting the output of a ResultSet**
   Databases may contain user input that has been stored, and as such must be treated as untrusted.
7. **Getting the value of a cookie**
   Cookies are controlled by the client, and so their values must be treated as untrusted.
8. **Getting the hostname of a request using reverse DNS**
   If the user controls their DNS server, then they can return whatever result they wish for a reverse DNS lookup.

Security analysis testing

The C/C++ security analyses are regularly evaluated against the SAMATE Juliet tests maintained by the US National Institute of Standards and Technology (NIST). This ensures that the quality and discrimination of the results is maintained as the queries are updated, for example, for changes to the C++ language, or improvements to the QL library, and enhancements to the code extraction process.

Summary of results

The following table summarizes the results for the latest release of the C/C++ security queries run against the SAMATE Juliet 1.3 tests. In the table, each row represents a weakness, and the columns show the following information:

- TP – count of all true positive results: the code has a known security weakness, and Semmle's analyses correctly identify this defect.
- FP – count of all false positive results: the code has no known security weakness, but Semmle's analyses are over cautious and suggest a potential problem.
- TN – count of true negative results: the code has no known security weakness, and Semmle's analyses correctly pass the code as secure.
- FN – count of all false negative results: the code has a known security weakness, but Semmle's analyses fail to identify this defect.

In an ideal implementation of the analyses, the number of false positives (FP) and false negatives (FN) would be zero, but that is impossible to achieve by static analysis. The figures for FP and FN show where there are limitations in the present implementation.

CWE	TP	FP	TN	FN	Count
CWE-022	3900	0	4800	900	4800
CWE-078	3900	0	4800	900	4800
CWE-114	390	0	576	186	576
CWE-119	2108	0	12336	10228	12336
CWE-134	2460	0	2880	420	2880
CWE-190	3276	113	3847	684	3960
CWE-191	2370	0	2952	582	2952
CWE-197	495	0	864	369	864
CWE-200	54	0	54	0	54
CWE-242	18	0	18	0	18
CWE-327	54	0	54	0	54
CWE-367	36	0	36	0	36
CWE-416	360	0	459	99	459
CWE-457	180	0	948	768	948
CWE-468	37	0	37	0	37
CWE-665	104	0	193	89	193
CWE-676	18	0	18	0	18
CWE-681	18	0	54	36	54
CWE-	1713	462	1340	89	1802

772					
CWE-835	3	0	6	3	6

Interpreting the results

The report CAS Static Analysis Tool Study – Methodology, by the Center for Assured Software of the National Security Agency of the USA defines four different ways to measure success:

- Precision = TP/ (FP+TP)
- Recall = TP/(TP+FN)
- F-Score = 2*(Precision*Recall)/(Precision+Recall)
- Discrimination rate = #discriminated tests / #tests

For each of these metrics, a higher score is better. There is clearly a trade-off between the precision and recall metrics: increasing the level of precision or recall for any analysis reduces the level of the other metric. The F-score is therefore an attempt to quantify the balance of decision between these two metrics.

The following table shows the results of calculating these metrics for the results shown above. These scores compare very favorably with the sample tools tested by the Center for Assured Software.

CWE	Precision	F-score	Recall	Disc. Rate
CWE-022	100%	90%	81%	81%
CWE-078	100%	90%	81%	81%
CWE-114	100%	81%	68%	68%
CWE-119	100%	29%	17%	17%
CWE-134	100%	92%	85%	85%
CWE-190	97%	89%	83%	80%
CWE-191	100%	89%	80%	80%
CWE-197	100%	73%	57%	57%
CWE-200	100%	100%	100%	100%
CWE-242	100%	100%	100%	100%
CWE-	100%	100%	100%	100%

327				
CWE-367	100%	100%	100%	100%
CWE-416	100%	88%	78%	78%
CWE-457	100%	32%	19%	19%
CWE-468	100%	100%	100%	100%
CWE-665	100%	70%	54%	54%
CWE-676	100%	100%	100%	100%
CWE-681	100%	50%	33%	33%
CWE-772	79%	86%	95%	69%
CWE-835	100%	67%	50%	50%

Conclusions

The tests suggest that Semmle has made judicious choices balancing the number of false positive results (an incorrect warning is issued) and false negative results (a true defect was not identified). Where comparative results are available for other tools, the Semmle analyses stand out for their exceptional accuracy.