# Detecting Implicit Conversions in OpenVPN2 Using CodeQL

Case Study

**September 2025**

*Prepared by:* **Paweł Płatek**

# Table of Contents

# Executive Summary

In 2023, Trail of Bits conducted a security review of OpenVPN2. The audit identified many instances of implicit integer conversions that could lead to security vulnerabilities (see finding TOB-OVPN-12). Trail of Bits recommended triaging all the conversions, preferably using a static analysis tool like CodeQL to facilitate the process.

This report is a follow-up on this recommendation. Its aim was to answer the question, **are any of OpenVPN2's approximately 2,500 implicit integer conversions actually exploitable?** While the answer turned out to be "no," this report documents our journey narrowing down these thousands of warnings to just 20 potentially vulnerable cases—and ultimately to determining that none of them represent exploitable bugs. This effort involved writing a new CodeQL query and improving CodeQL's existing analyses until it produced a number of findings manageable for manual review.

In this report, we describe the static analysis approaches we used at each step in refining our query:

| Topic | Findings in OpenVPN2 |
|---|---|
| Modeling of expressions, types, and conversions | 5,725 |
| Constant analysis | 1,017 |
| Data flow analysis (discarded) | 830 |
| Simple range analysis (a.k.a., value range propagation) | 913 |
| Extending simple range analysis | 575 |
| IR-based range analysis | 435 |
| Modeling of codebases | 254 |
| Taint tracking analysis from user-controlled inputs | 20 |

This report is intended for both developers and security researchers. It includes four sections:

- Explanation of the implicit conversion algorithm and systemic review of how implicit conversions may introduce security bugs
- Overview of existing detection capabilities by compilers and CodeQL
- Step-by-step construction of a new CodeQL query

- Recommendations on how to improve the security of C codebases

The final version of the CodeQL query for finding problematic implicit integer conversions can be found in Trail of Bits' `codeql-queries` repository.

# Overview of Implicit Conversion Bugs

C language does not have strict type checking. As a result, the compiler introduces implicit type conversions whenever an *expression is used in a context where a value of a different type is expected*. As this is performed during the compilation phase, such conversions are hidden from developers.

Let us start with an overview on when and how the compiler introduces conversions (or, casts).

## When Implicit Conversions Are Introduced

For our purposes, we are interested in these three types of implicit integer conversions:

1. Conversions as if by assignment (or, conversions of one integer type to another): This type of conversion may occur during variable initialization, variable assignment, function calls, and function returns.

```
int func(short arg) {
    long long z = arg; // short to long long
    return z; // long long to int
}

void main() {
    unsigned int arg = -1;  // int to unsigned int
    func(arg); // unsigned int to short
}
```

*Figure 1: Example conversions as if by assignment*

2. *Integer promotions* (or, casting small types to `int`): This type of conversion happens during the first stage of types reconciliation (the third type, described below), as well as for an operand of unary arithmetic and bitwise operators (+, -, ~), and for both operands of shift operations (<<, >>).

```
void main() {
    uint8_t val = 0x13;
    uint8_t val2 = (~val) >> 3;  // integer promotion (followed by int to uint8_t)
}
```

*Figure 2: `val` is first promoted to `int`, then bitwise operations are performed, then `val` is converted back to `uint8_t`.*

3. Usual arithmetic conversions (or, types reconciliation). This type of conversion occurs on integer operands in binary (bitwise) arithmetic (+, -, *, &, ^, ...), relational operations (==, >, ...), or conditional operations (? :).

```
int main() {
    unsigned short val = 0x13;
    int val2 = 0x37;
    return val + val2;  // unsigned short and int reconciled
}
```

*Figure 3: Example of usual arithmetic conversions*

In summary, this is what happens during implicit conversions:

1. *Conversions as if by assignment*: The integer conversion algorithm (see below) is applied to the operand.
2. *Integer promotions*: The operand is optionally converted to `int` or `unsigned int`.
3. *Usual arithmetic conversions*: Both operands undergo integer promotion. Then, the integer conversion algorithm is optionally applied to one or both operands.

The integer conversion algorithm takes as input source and destination types and a source value (the operand). It works as follows:

- If types are compatible, the value is not tampered with.
- If the destination type is unsigned, modulo arithmetic is performed.
- If the destination type is signed, then what happens next depends on the implementation (see the documentation for GCC, Clang, and IBM z/OS). Usually, it is modulo arithmetic.

The algorithm for deciding which operands should undergo integer conversion in the usual arithmetic conversion case is a bit complicated; take your time to parse it:

5) Otherwise, both operands are integers. Both operands undergo integer promotions; then, after integer promotion, one of the following cases applies:
- If the types are the same, that type is the common type.
- Else, the types are different:
  - If the types have the same signedness (both signed or both unsigned), the operand whose type has the lesser *conversion rank*[1] is implicitly converted[2] to the other type.
  - Else, the operands have different signedness:
    - If the unsigned type has *conversion rank* greater than or equal to the rank of the signed type, then the operand with the signed type is implicitly converted to the unsigned type.
    - Else, the unsigned type has *conversion rank* less than the signed type:
      - If the signed type can represent all values of the unsigned type, then the operand with the unsigned type is implicitly converted to the signed type.
      - Else, both operands undergo implicit conversion to the unsigned type counterpart of the signed operand's type.

1. ↑ Refer to integer promotions below for the rules on ranking.
2. ↑ Refer to "integer conversions" under implicit conversion semantics below.

*Figure 4: Point 5 of the usual arithmetic conversions algorithm section from cppreference.com*

## What Happens to Values

From a security perspective, we are interested in what happens to the actual values of operands after implicit conversions are applied. There are three problematic cases:

- Truncation: When the destination type is smaller than the source and cannot represent the value (i.e., the value is too large to fit into the destination type), the destination value is truncated. In this case, we lose information, which is usually bad.
- Reinterpretation: When the destination type's size is equal to the source type's size, but they have different signedness, the destination value is reinterpreted:
    - Signed to unsigned: Negative values become large positive values.
    - Unsigned to signed: Large values become negative.

  Both cases may lead to unexpected behavior.
- Widening: When a smaller signed type is cast to a bigger unsigned type, negative values wrap around to big positive values, which may be unexpected. Moreover, when a bitwise complement operation (~) is done on a wider type, the results may also be unexpected.

## Problematic Cases

This table shows whether each problematic case is possible when each of the three implicit conversion types occurs:

| | Truncation | Reinterpretation | Widening |
|---|---|---|---|
| As if by assignment | Possible | Possible | Possible |
| Integer promotions | Not possible | Not possible | Possible |
| Usual arithmetic conversions | Not possible | Possible | Possible |

Later in this report, we will discuss how we used CodeQL to detect these "possible" vulnerabilities, but first, we will discuss some exceptions—that is, implicit casts we can consider "safe"—that we can safely ignore (and some we want to detect regardless).

**For conversions as if by assignment**, all three problematic cases may occur. We could ignore some problematic conversions in assignments if the converted value is used only after it is cast back to its initial type (see the example in figure 5). However, experience suggests that such cases are rare, so we will not skip them.

```
short a = -1;
```

```
unsigned long long b = a;
short c = b;
```

*Figure 5: An example of safe implicit casts that we want to detect anyway*

**For integer promotions**, there is one problematic case we want to find: a bitwise complement operation done on a wider type. Such widening is most likely not expected, as the operand's value changes according to the integer's width.

We will skip promotion-related problems that do not involve changes to operands' values. Consider the example in figure 6.

```
unsigned short a = 1;
if (a - 5 < 0) { // promotion to int, unexpected
       puts("called");
}

unsigned short b = 1;
b = b - 5;
if (b < 0) {  // no unexpected promotion
       puts("not called");
}
```

*Figure 6: Example problematic integer promotion*

One may expect the result of `a - 5` to wrap around to an unsigned integer, but this is not the case because of the integer promotion that occurs on this value. In more complex code constructs than this example, problematic promotions may be quite hard to spot.

Moreover, you can unwillingly enter the realm of undefined behavior due to a signed integer overflow after a silent promotion. You can read more about that problem in our blog post "Look out! Divergent representations are everywhere!" and Jeff Hurchalla's blog post "Surprises and Undefined Behavior From Unsigned Integer Promotion."

**During usual arithmetic conversions**, truncation should not be possible. Reinterpretation and widening in addition, subtraction, and multiplication operations is possible, but we can ignore these cases because modulo arithmetic makes these operations behave as expected when it comes to values. Any problems related to these operations would be in the use of their results—but that is covered by conversions as if by assignment.

We can also ignore reinterpreting and widening conversions that are part of equality checks when the upper bound of the other side is smaller than the destination type's maximal value plus the source type's minimal (possibly negative) value. This sounds complicated, but the code snippet in figure 7 succinctly demonstrates this case.

```
int a = getValue();
const unsigned int b = 0xffff3502;
if (a != b) {  // negative a may wrap to b, problematic comparison
```

```
    return;
}

const unsigned int c = 0xcafe;
if (a != c) {  // negative a cannot wrap to c, safe comparison
    return;
}
```

*Figure 7: A negative a value cannot overflow so much that c's upper bound is reached.*

**We are not interested in integer overflows/underflows** in this report; this bug class is a whole different topic. Note that implicit conversions can be used as heuristics for detecting problematic overflows.

```
long mul(int a, int b) {
  return a * b;
}
```

*Figure 8: An `int`-to-`long` conversion can be used to mark the possible overflow in multiplication as problematic.*

Moreover, we are skipping bugs related to implicit conversions of Booleans, floats, bit fields, pointers, qualifiers, and casts during default argument promotions.

Lastly, we do not consider problems with explicit conversions. The assumption is that developers know what they are doing, but may be surprised by implicit, compiler-generated conversions.

For an example of real-world vulnerability of the kind we are after, see CVE-2021-33909.

# Existing Tools for Finding Implicit Conversion Bugs

Now that we understand the implicit conversion bugs we are looking for, let us try to find them in OpenVPN2, using the GCC and Clang compilers and CodeQL with standard queries.

## Compilers

GCC and Clang offer some warnings related to implicit conversions. We can build OpenVPN2 with the relevant warnings enabled with commands below:

```
git clone git@github.com:OpenVPN/openvpn.git && cd openvpn
git checkout v2.6.14
autoreconf -i -v -f
./configure --with-crypto-library=openssl CC="gcc-14" CFLAGS="-Wconversion
-Wsign-compare"
make 2> gcc_warnings.txt
make clean

./configure --with-crypto-library=openssl CC="clang" CFLAGS="-Wconversion
-Wsign-compare"
make 2> clang_warnings.txt

for C in gcc clang; do echo $C; rg '\[-W' "./${C}_warnings.txt" | rev | cut -w -f1 |
rev | sort | uniq -c; done
```

*Figure 9: Commands for compiling OpenVPN2 with conversion warnings*

The following are the results produced by GCC v14.2.0 and Clang v19.1.7:

| Compiler | Warning Type | Count |
|---|---|---|
| GCC | conversion | 476 |
| GCC | sign-conversion | 2,145 |
| GCC | sign-compare | 77 |
| Clang | shorten-64-to-32 | 423 |
| Clang | implicit-int-conversion | 41 |
| Clang | sign-conversion | 1,887 |
| Clang | sign-compare | 71 |

Most of the conversion instances we are interested in should be detected with these flags. One exception is implicit widening during bitwise complement operations, which these compilers do not report.

However, the conversion-related flags produce a lot of false positives as well. **About 2,500 findings** is a lot to review manually. And we have no simple way to improve the results except for modifying the OpenVPN2 code.

## CodeQL

There are basically two ways to approach finding a specific bug pattern with CodeQL: either review the available rules and select those that seem promising, or run CodeQL with all rules enabled and review the results. The latter takes longer but may be more comprehensive, so we will do that here:

```
cat >com.sh <<EOF
#!/bin/bash

autoreconf -i -v -f
./configure --with-crypto-library=openssl
make
EOF
codeql database create codeql.db --language=cpp -c 'bash ./com.sh'

cd ~/codeql-home/codeql-repo/cpp/ql/src/codeql-suites
cat >cpp-real-all.qls <<EOF
- description: All queries for C
- queries: .
- exclude:
    query path:
      - /^external\/.*/
      - /^Metrics\/.*/
      - /^PointsTo\/.*/
      - /^Architecture\/.*/
      - /^Security\/.*DataToExternalAPI.ql$/
EOF
codeql database analyze ./codeql.db -o codeql.sarif --format=sarif-latest --
cpp-real-all
```

*Figure 10: Command for building a CodeQL database and analyzing it with all rules for C*

We made a new query suite to run all the queries from the CodeQL standard repository. Reviewing the results, we can pick queries that seem to be relevant:

- `cpp/conversion-changes-sign`
    - 988 findings
    - Reports only implicit unsigned-to-signed conversions
    - Filters out safe conversions with `const` values
- `cpp/jsf/av-rule-180`
    - 6,750 findings

    - Supports types up to 32 bits only
    - Reports truncation and reinterpretation issues, but not widening
- `cpp/sign-conversion-pointer-arithmetic`
    - 1 finding
    - Detects only unsigned-to-signed conversions of variables that are then used in pointer arithmetic
    - Reports both explicit and implicit conversions

None of these rules detect all the bugs we are after, and they produce too many results for manual triaging. While we can get some inspiration from them, what we need is a new query.

# Writing a New CodeQL Query

In this section, we describe the step-by-step process we took to write a new CodeQL query to detect implicit conversions in OpenVPN2. We started by writing a simple query that detects all implicit conversions, and we added extensions to narrow down its results until we reached a number that we could easily triage.

## Detecting All Conversions

To write a new query, CodeQL needs to be installed locally (we use version 2.22.1) and a directory structure and files need to be created as shown in figure 11 (see also the "Writing custom queries" section of the Trail of Bits Testing Handbook).

```
$ tree codeql-queries/cpp
.
├── src
│   └── security
│       └── UnsafeImplicitConversions
│           ├── UnsafeImplicitConversions.cpp
│           ├── UnsafeImplicitConversions.qhelp
│           └── UnsafeImplicitConversions.ql
└── tests
    └── security
        └── UnsafeImplicitConversions
            ├── UnsafeImplicitConversions.expected
            ├── UnsafeImplicitConversions.qlref
            └── test.cpp
```

*Figure 11: File tree for a new CodeQL query*

The query help (`.qhelp`) and query reference (`.qlref`) files define the purpose and location of the query, respectively; read more about them in the CodeQL documentation. The `test.cpp` and `UnsafeImplicitConversions.expected` files can be used to run a query without rebuilding a database after every change in the tested code. This feature enables new queries to be developed efficiently with the **implement-test-refine methodology**. This feature can be easily used through the Visual Studio Code CodeQL extension.

We will begin with a query that finds all implicit conversions by adding the code in figure 12 to the `UnsafeImplicitConversions.ql` file.

```
/**
 * @name Find all implicit casts
 * @description Find all implicit casts
 * @kind problem
 * @id tob/cpp/unsafe-implicit-conversions
 * @tags security
 * @problem.severity warning
```

```
 * @precision low
 **/

import cpp

from IntegralConversion cast, IntegralType fromType, IntegralType toType
where
    cast.isImplicit()
    and fromType = cast.getExpr().getExplicitlyConverted().getUnspecifiedType()
    and toType = cast.getUnspecifiedType()
    and fromType != toType
    and not toType instanceof BoolType

select cast, "Implicit cast from " + fromType + " to " + toType
```

*Figure 12: A query for finding all implicit conversions*

The `IntegralConversion` class contains all conversions between integers. We limit the class to only implicit conversions. Because in CodeQL a list of conversions is "attached" to an expression, we further limit the results by discarding implicit conversions happening before an explicit conversion, through a call to the `getExplicitlyConverted` method. The `getUnspecifiedType` predicate resolves `typedef`s and strips specifiers (like `const`), which we are not interested in. Finally, we ignore conversions between the same types and to the Boolean type.

Running the query against OpenVPN2 **yields 7,000 findings**. To narrow down the results, we can limit them to potentially unsafe conversions only (which we covered in the "Problematic Cases" section), using the code in figure 13.

```
and (
    // truncation
    fromType.getSize() > toType.getSize()
    or
    // reinterpretation
    (
        fromType.getSize() = toType.getSize()
        and
        (
            (fromType.isUnsigned() and toType.isSigned())
            or
            (fromType.isSigned() and toType.isUnsigned())
        )
    )
    or
    // widening
    (
        fromType.getSize() < toType.getSize()
        and
        (
            (fromType.isSigned() and toType.isUnsigned())
```

```
            or
            // unsafe promotion
            exists(ComplementExpr complement |
                complement.getOperand().getConversion*() = cast
            )
        )
    )
)

and not (
    // skip conversions in arithmetic operations
    fromType.getSize() <= toType.getSize() // should always hold
    and exists(BinaryArithmeticOperation arithmetic |
        (arithmetic instanceof AddExpr or arithmetic instanceof SubExpr or
arithmetic instanceof MulExpr)
        and arithmetic.getAnOperand().getConversion*() = cast
    )
)
```

*Figure 13: Update that limits the query's results to problematic conversions only*

The query now yields **5,725 findings**. That is still too many to deal with, so we will keep improving it, first by checking for constant values.

## Constant Values

CodeQL has a simple way of checking the value of an expression if the value is constant: the `getValue` predicate. With this predicate, we can filter out conversions that are known to be safe.

First, we add a new condition to the `where` clause.

```
and not isSafeConstant(cast.getExpr(), toType)
```

*Figure 14: A new predicate*

Then, we create the `isSafeConstant` predicate.

```
import semmle.code.cpp.rangeanalysis.RangeAnalysisUtils

predicate isSafeConstant(Expr cast, IntegralType toType) {
    exists(float knownValue |
        knownValue = cast.getValue().toFloat()
        and knownValue <= typeUpperBound(toType)
        and knownValue >= typeLowerBound(toType)
    )
}
```

*Figure 15: The `isSafeConstant` predicate checks the bounds of the given value to limit false positives*

The predicate checks if the provided expression's value is known and if it fits into the type it is being converted into. The `toFloat` method is important here: in CodeQL a number can be represented as either a 64-bit float or 32-bit signed integer; there is no arbitrary precision numeric type. If we used the integer, we would not be able to get values of large integer types like `long long`.

The `typeLowerBound` and `typeUpperBound` predicates come from the `semmle.code.cpp.rangeanalysis.RangeAnalysisUtils` library that we import at the top of the file; these predicates simply compute the maximal and minimal values that the type can hold.

Finally, we add another condition to the `where` clause that ignores safe conversions inside equality operations (the explanation for this decision is provided in the "Problematic Cases" section).

```
and not (
    fromType.getSize() <= toType.getSize() and
    exists(EqualityOperation eq, Expr firstHandSide, Expr secondHandSide |
      firstHandSide = eq.getAnOperand() and
      secondHandSide = eq.getAnOperand() and
      firstHandSide != secondHandSide and
      firstHandSide.getConversion*() = cast and
      secondHandSide.getValue().toFloat() < (typeUpperBound(toType) +
typeLowerBound(fromType))
    )
  )
```

*Figure 16: Exclusion of safe equality checks*

With these changes, our query now yields only **1,017 findings**. But to check the limitations of the `getValue` method, we can run the following test (that is, by running the CodeQL query against it and checking the findings).

```
char* malloc_wrapper(int size) {
    return (char*)malloc(size);
}

static uint64_t const_large_ok_value4;

void test_fp(uint64_t arg, uint64_t arg2) {
    const uint64_t const_large_ok_value = 0x1;
    const uint64_t const_large_ok_value2 = 0x3 + 0x4;
    uint64_t const_large_ok_value3 = 0x2;
    free(malloc_wrapper(const_large_ok_value));  // 1
    free(malloc_wrapper(const_large_ok_value2)); // 2
    free(malloc_wrapper(const_large_ok_value3)); // 3
    free(malloc_wrapper(const_large_ok_value4)); // 4
    free(malloc_wrapper(0xbabe)); // 5
    free(malloc_wrapper(arg)); // 6
```

```
    free(malloc_wrapper(arg2)); // 7
}

int main(int argc, char *argv[]) {
    if (argc < 2)
        return 1;

    const_large_ok_value4 = 0x5;
    test_fp(0xcafe, 1);
    const_large_ok_value4 = 0x6;
    test_fp(strtoul(argv[1], NULL, 16), 1);
    return 0;
}
```

*Figure 17: Test for finding `getValue` limits*

This test shows that only conversions numbered 1, 2, and 5 are not reported. That means the `getValue` method is rather strict and does not try to return a number that is known at compile time but is not hard-coded or explicitly marked as `const`.

## Constant Analysis

We can try to improve the query further with the `getConstantValue` predicate. To use it, we must import a few libraries, get an intermediate representation (IR) instruction related to the conversion expression, and simply call the predicate on the instruction.

```
private import
semmle.code.cpp.ir.implementation.aliased_ssa.constant.ConstantAnalysis
private import semmle.code.cpp.ir.IR
private import semmle.code.cpp.ir.internal.IntegerConstant
private import semmle.code.cpp.rangeanalysis.RangeAnalysisUtils

predicate isSafeConstant(Expr cast, IntegralType toType) {
    exists(float knownValue, Instruction castIR |
        (
            (
            castIR.getUnconvertedResultExpression() = cast
            and knownValue = getValue(getConstantValue(castIR))
            )
            or knownValue = cast.getValue().toFloat()
        )
        and knownValue <= typeUpperBound(toType)
        and knownValue >= typeLowerBound(toType)
    )
}
```

*Figure 18: The `isSafeConstant` predicate with the `getConstantValue` predicate*

Adding this predicate removes **only three false positives** (compared to the raw `getValue`). An example of one of the falsified findings is shown in figure 19. The implicit

conversion occurs in the call to `malloc`, but the value is constant and in the destination type's range.

```
static void
lzo_compress_init(struct compress_context *compctx)
{
    [skipped]
    compctx->wu.lzo.wmem_size = LZO_WORKSPACE;

    [skipped]
    compctx->wu.lzo.wmem = (lzo_voidp) malloc(compctx->wu.lzo.wmem_size);
    [skipped]
}
```

*Figure 19: A false positive reduced by the constant analysis predicate (openvpn/src/openvpn/lzo.c#98–112)*

Looking at the implementation of CodeQL's constant analysis, we can see that it is oversimplified—intraprocedural only, with limited support for bitwise and unary arithmetic[1]. Moreover, it works on `ints`, and not on `floats`, making it unable to handle large values. These facts should explain the lack of effectiveness.

## Data Flow Analysis

The next extension to the query we will try is data flow analysis. We can use it to check if all possible values that flow to a conversion are known to be safe (that is, are in range of the destination type).

```
import semmle.code.cpp.dataflow.new.DataFlow

module ToCastConfig implements DataFlow::ConfigSig {
    predicate isSource(DataFlow::Node source) { source = source }
    predicate isSink(DataFlow::Node sink) { sink.asExpr().getConversion() instanceof
IntegralConversion }
  }
module ToCastConfigFlow = DataFlow::Global<ToCastConfig>;

module LongestFlowConfig implements DataFlow::ConfigSig {
    predicate isSource(DataFlow::Node source) { source = source }
    predicate isSink(DataFlow::Node sink) { sink = sink }
  }
module LongestFlowConfigFlow = DataFlow::Global<LongestFlowConfig>;

bindingset[knownValue]
predicate safeValue(float knownValue, IntegralType toType) {
    knownValue <= typeUpperBound(toType)
    and knownValue >= typeLowerBound(toType)
```

[1] This was improved by GitHub during the writing of this case study.

```
}

predicate isSafeConstant(Expr cast, IntegralType toType) {
    safeValue(cast.getValue().toFloat(), toType)
    or
    forex(
        // new variables
        DataFlow::Node source, DataFlow::Node sink |

        // formula 1
        ToCastConfigFlow::flow(source, sink)
        and source != sink
        and sink.asExpr() = cast
        // only the longest flow
        and not exists(DataFlow::Node source2 |
            LongestFlowConfigFlow::flow(source2, source)
            and source != source2
        )
        |
        // formula 2
        exists(float knownValue |
            knownValue = source.asExpr().getValue().toFloat()
            and safeValue(knownValue, toType)
        )
    )
}
```

*Figure 20: Our query with data flow analysis added*

The `safeValue` method uses the `bindingset` annotation on the `knownValue` variable; this is required because we do not restrict the set of possible values that the `knownValue` variable can hold to a finite set inside the predicate. With this annotation, we let the CodeQL engine know that `safeValue` will only ever be called with an already limited set of float values.

In adding data flow analysis, we can have the `isSafeConstant` predicate check if either of the following is true:

- The expression being converted has a known value and the value is safe.
- At least one data flow to the conversion exists and all flows to the conversion have known, safe values.

For the second bullet point, we use the `forex` quantifier—a shortcut for the `exists` and `forall` quantifiers. In the `forex` quantifier, we first specify new variables; then (after the `|` character) **the first formula** limits the set of values for the new variables (there must be **at least one** set of values for which the formula holds), and then (after the second pipe character) the **second formula** must hold for **all** value sets that are left after the first formula limits the results.

The first `exists` predicate (with the `LongestFlowConfigFlow` module) is there to limit the `ToCastConfigFlow` flows to only the longest ones. Without it, we would have a lot of flows from intermediary sources to the sink (the conversion), most of which would not have a known value, thereby causing the whole `isSafeConstant` predicate to fail.

Also worth noting is that we must explicitly state that we are not interested in data flow nodes that are both sources and sinks at the same time (through the lines `source != sink` and `source != source2`).

A nice example of how the use of data flow analysis reduces the query's findings is shown in figures 21 and 22.

```
unsigned int crypto_overhead = 0;
crypto_overhead += cipher_kt_tag_size(kt->cipher);
```

*Figure 21: Implicit conversion from `int` to `unsigned int` in OpenVPN2 (openvpn/src/openvpn/crypto.c#687)*

```
int
cipher_kt_tag_size(const char *ciphername)
{
    if (cipher_kt_mode_aead(ciphername))
    {
        return OPENVPN_AEAD_TAG_LENGTH;
    }
    else
    {
        return 0;
    }
}
```

*Figure 22: We know that the only possible return values from `cipher_kt_tag_size` are `0` and `OPENVPN_AEAD_TAG_LENGTH`. (openvpn/src/openvpn/crypto_openssl.c#748–759)*

The `cipher_kt_tag_size` function returns an `int` that is implicitly converted to `unsigned int`. However, with data flow analysis, CodeQL knows that the function can return only two values and, therefore, the implicit conversion is always safe.

Unfortunately, these changes cause the query to produce false negatives: possible bugs are not reported anymore. In the loop's condition in the code in figure 23, the `i` integer is cast to unsigned long. The data flow analysis finds only the initial assignment with a 0 value, but is not aware of the incrementation in the loop. This means that our query is no longer sound.

```
for (int i = 0; i < cipher_list.num; i++)
{
```

```
    if (cipher_kt_insecure(EVP_CIPHER_get0_name(cipher_list.list[i])))
    {
        print_cipher(EVP_CIPHER_get0_name(cipher_list.list[i]));
    }
}
```

*Figure 23: An example false negative (openvpn/src/openvpn/crypto_openssl.c#411–417)*

Use of data flow analysis results in **830 findings**. This represents a decent reduction of false positives, but at the expense of much longer execution time (from 1 to 242 seconds on the OpenVPN2 codebase), in addition to the lost soundness. We can conclude that data flow analysis is a dead end. **Therefore, we will remove the code from this section** and try a different approach.

## Simple Range Analysis

CodeQL implements value range propagation analysis under the name "range analysis." This analysis aims to determine bounds (minimum and maximum values) of variables at a given point in the program. We can use it to eliminate false positives when a conversion can be proven to be safe.

The analysis comes in two flavors:

- Simple analysis (`semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis`)
- Analysis working on a C++ implementation's IR[2] (`experimental.semmle.code.cpp.rangeanalysis.RangeAnalysis`)

The simple range analysis implements a simplified abstract interpretation algorithm, representing integers as lower-upper bound pairs. The main difference from usual abstract interpretation implementations comes from the fact that CodeQL works iteratively on "layers" and must approximate ("widen") bounds in a less precise manner than normally possible. The analysis has a straightforward API: `lowerBound` and `upperBound` predicates that take an expression as the only argument and return a specific number.

Let us refine our query with simple range analysis:

```
private import semmle.code.cpp.rangeanalysis.RangeAnalysisUtils
private import semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis

predicate safeLowerBound(Expr cast, IntegralType toType) {
    exists(float lowerB |
        lowerB = lowerBound(cast)
        and lowerB >= typeLowerBound(toType)
```

[2] Usage of experimental libraries is discouraged. New code should use the shared `RangeAnalysis` library via the bounded predicate.

```
    )
}

predicate safeUpperBound(Expr cast, IntegralType toType) {
    exists(float upperB |
        upperB = upperBound(cast)
        and upperB <= typeUpperBound(toType)
    )
}

predicate safeBounds(Expr cast, IntegralType toType) {
    safeLowerBound(cast, toType) and safeUpperBound(cast, toType)
}


from IntegralConversion cast, IntegralType fromType, IntegralType toType, Expr
castExpr
 where
     [skipped]
     and cast.getExpr() = castExpr

     and (
        // truncation
        (
            fromType.getSize() > toType.getSize()
            and not safeBounds(castExpr, toType)
        )
        or
        // reinterpretation
        (
            fromType.getSize() = toType.getSize()
            and
            (
                (fromType.isUnsigned() and toType.isSigned() and not
safeUpperBound(castExpr, toType))
                or
                (fromType.isSigned() and toType.isUnsigned() and not
safeLowerBound(castExpr, toType))
            )
        )
        or
        // widening
        (
            fromType.getSize() < toType.getSize()
            and
            (
                (fromType.isSigned() and toType.isUnsigned() and not
safeLowerBound(castExpr, toType))
                or
                // unsafe promotion
                exists(ComplementExpr complement |
                    complement.getOperand().getConversion*() = cast
                )
```

```
            )
        )
    )

    [skipped]

    and not (
      fromType.getSize() <= toType.getSize() and
      exists(EqualityOperation eq, Expr firstHandSide, Expr secondHandSide |
        firstHandSide = eq.getAnOperand() and
        secondHandSide = eq.getAnOperand() and
        firstHandSide != secondHandSide and
        firstHandSide.getConversion*() = cast and
        upperBound(secondHandSide) < (typeUpperBound(toType) +
typeLowerBound(fromType))
      )
    )

select cast, "Implicit cast from " + fromType + " to " + toType
```

*Figure 24: Our query updated with simple range analysis*

The `safeBounds` predicate checks if the value being converted fits into the destination type. If that is the case, then we know that the conversion will never trigger problematic behavior. The simple range analysis should be sound, so the query should not miss bugs.

With simple range analysis, the query produces **913 results**. Reviewing the findings, we can observe many false positives that we could have expected to be eliminated by the analysis. This observation indicates that there is a room for improvement in the simple range analysis itself.

## Extending Simple Range Analysis

CodeQL includes an API for creating experimental extensions to the simple range analysis. The following extensions have been implemented so far:

- The `ConstantBitwiseAndExprRange` class, which adds (limited) support for the & operator
- The `ConstantShiftExprRange` class, which adds support for binary shift operations
- The `StrlenLiteralRangeExpr` class, which adds support for `strlen` calls with constant strings
- The `SubtractSelf` class, which adds support for `x-x` expressions

To enable the extensions, we can add the import shown in figure 25 to the query.

```
private import experimental.semmle.code.cpp.rangeanalysis.ExtendedRangeAnalysis
```

*Figure 25: Import enabling range analysis extensions*

Note that the `ConstantShiftExprRange` class is not in the `ExtendedRangeAnalysis.qll` file. The class seems to be buggy anyway: manually adding it to the file makes the analysis less precise. That is probably because the class breaks the invariant specifying that if an expression fits into the `SimpleRangeAnalysisExpr` class then its `getLowerBounds` and `getUpperBounds` methods return some value.

With the extensions enabled, the query produces **899 findings**.

But none of these extensions add support for the binary `OR` operator, which could further reduce the findings. To create an extension supporting this operator, we will create the following file in the `rangeanalysis/extensions` folder and import it in the `ExtendedRangeAnalysis.qll` file (or we could simply put the content directly in the query file). The implementation is based on chapter 4 of Hank Warren's book *Hacker's Delight*.

```
private import experimental.semmle.code.cpp.rangeanalysis.ExtendedRangeAnalysis
private import semmle.code.cpp.rangeanalysis.RangeAnalysisUtils

predicate getLeftRightOrOperands(Expr orExpr, Expr l, Expr r) {
  l = orExpr.(BitwiseOrExpr).getLeftOperand() and
  r = orExpr.(BitwiseOrExpr).getRightOperand()
  or
  l = orExpr.(AssignOrExpr).getLValue() and
  r = orExpr.(AssignOrExpr).getRValue()
}

private class ConstantBitwiseOrExprRange extends SimpleRangeAnalysisExpr {
  ConstantBitwiseOrExprRange() {
    exists(Expr l, Expr r |
      getLeftRightOrOperands(this, l, r) |
      // No operand can be a negative constant
      not (evaluateConstantExpr(l) < 0 or evaluateConstantExpr(r) < 0)
    )
  }

  Expr getLeftOperand() { getLeftRightOrOperands(this, result, _) }

  Expr getRightOperand() { getLeftRightOrOperands(this, _, result) }

  override float getLowerBounds() {
    // If an operand can have negative values, the lower bound is unconstrained.
    // Otherwise, the upper bound is the sum of upper bounds.
    exists(float lLower, float rLower |
      lLower = getFullyConvertedLowerBounds(this.getLeftOperand()) and
      rLower = getFullyConvertedLowerBounds(this.getRightOperand()) and
      (
```

```
        (lLower < 0 or rLower < 0) and
        result = exprMinVal(this)
        or
        result = lLower.maximum(rLower)
      )
    )
  }

  override float getUpperBounds() {
    // If an operand can have negative values, the upper bound is unconstrained.
    // Otherwise, the upper bound is the greatest upper bound.
    exists(float lLower, float lUpper, float rLower, float rUpper |
      lLower = getFullyConvertedLowerBounds(this.getLeftOperand()) and
      lUpper = getFullyConvertedUpperBounds(this.getLeftOperand()) and
      rLower = getFullyConvertedLowerBounds(this.getRightOperand()) and
      rUpper = getFullyConvertedUpperBounds(this.getRightOperand()) and
      (
        (lLower < 0 or rLower < 0) and
        result = exprMaxVal(this)
        or
        result = rUpper + lUpper
      )
    )
  }

  override predicate dependsOnChild(Expr child) {
    child = this.getLeftOperand() or child = this.getRightOperand()
  }
}
```

*Figure 26: Range analysis extension for OR operator*

Note that while writing this extension, we observed a CodeQL limitation: native CodeQL's float type does not implement bitwise arithmetic methods—that's why we had to approximate bounds, instead of implementing precise calculations as recommended in *Hacker's Delight*.

With OR support, the query produces **886 findings**. In reviewing the findings, it is obvious that the **simple range analysis is not interprocedural** (a.k.a., context-sensitive or polyvariant). Consider these limitations:

- The analysis does not track global variables. For example, the `x_debug_level` `unsigned int` global variable can take only a limited set of small values, but a false positive is reported for the return statement in the `get_debug_level` function.

```
int
get_debug_level(void)
{
    return x_debug_level;
}
```

*Figure 27: A false positive with global variables (openvpn/src/openvpn/error.c#136–140)*

- The analysis does not propagate bounds of functions' return values. For example, a finding in the `get_ip_encap_overhead` function is reported, where an integer result of the call to the `datagram_overhead` function is converted to `unsigned int`. However, the analysis knows that the `datagram_overhead` function can only return small positive values (the `overhead` variable's bounds are correctly determined).

```
static unsigned int
get_ip_encap_overhead(const struct options *options,
                      const struct link_socket_info *lsi)
{
    [skipped]
    return datagram_overhead(af, proto);
}
```

*Figure 28: A false positive with known return values (openvpn/src/openvpn/mss.c#229–252)*

```
static inline int
datagram_overhead(sa_family_t af, int proto)
{
    int overhead = 0;
    overhead += (proto == PROTO_UDP) ? 8 : 20;
    overhead += (af == AF_INET) ? 20 : 40;
    return overhead;
}
```

*Figure 29: This method's return value is bounded by a small, positive number. (openvpn/src/openvpn/socket.h#615–622)*

- The analysis does not bound function arguments. For example, in the `buf_string_match` function, the conversion from `int` to `size_t` in the call to `memcmp` is reported by the query, but if we track all calls to the `buf_string_match` function, we can see that there is only one call with the `PING_STRING_SIZE` constant.

```
static inline bool
buf_string_match(const struct buffer *src, const void *match, int size)
{
    if (size != src->len)
    {
        return false;
    }
    return memcmp(BPTR(src), match, size) == 0;
}
```

*Figure 30: A false positive with known parameter bounds (openvpn/src/openvpn/buffer.h#864–872)*

```
#define PING_STRING_SIZE 16

static inline bool
is_ping_msg(const struct buffer *buf)
{
    return buf_string_match(buf, ping_string, PING_STRING_SIZE);
}
```

*Figure 31: All calls to `buf_string_match` have a constant `size` argument. (openvpn/src/openvpn/ping.h#40–44)*

Interprocedural analyses—of any kind, not only for bound propagation—are known to be hard computer science problems. CodeQL's simple range analysis must have been good enough for its use cases in the intraprocedural mode for the engineers not to improve it with contextual information.

However, for our use case, even a naive extension of the analysis with context information should be beneficial. Let us implement function parameter propagation—the last bullet above. To find the right bounds, we can detect all calls to a function and set the function's parameters' bounds to a sum of bounds of arguments that are passed to calls to that function.

On the implementation side, we will use the `SimpleRangeAnalysisDefinition` interface. In contrast to the `SimpleRangeAnalysisExpr` class that we used to implement support for the `OR` operator, this interface allows us to set bounds for a variable at a "point in the program." That is, we can set bounds for a function's parameter at the first control flow node of the function, and the bounds will be automatically propagated. If we were to use the `SimpleRangeAnalysisExpr` class, we would "override" all locally introduced constraints to the parameter's bounds.

Note that CodeQL considers a "parameter" a variable declared in a function's definition, and an "argument" an expression that is passed to a function call.

```
private import
experimental.semmle.code.cpp.models.interfaces.SimpleRangeAnalysisDefinition

predicate addressIsTaken(Function f) {
  exists(FunctionCall call |
    call.getAnArgument().getFullyConverted() = f.getAnAccess()
  )
  or
  exists(Expr e |
    e.getFullyConverted() = f.getAnAccess() and
    e.getType() instanceof FunctionPointerType
```

```
  )
  or
  exists(Variable v |
    v.getInitializer().getExpr().getFullyConverted() = f.getAnAccess() and
    v.getType() instanceof FunctionPointerType
  )
  or
  exists(ArrayAggregateLiteral arrayInit |
    arrayInit.getAnElementExpr(_).getFullyConverted() = f.getAnAccess()
  )
  or
  exists(ReturnStmt ret |
    ret.getExpr().getFullyConverted() = f.getAnAccess()
  )
}

class ConstrainArgs extends SimpleRangeAnalysisDefinition {
  private Function func;
  private Parameter param;
  private FunctionCall call;

  ConstrainArgs() {
    param.getFunction() = func
    and call.getTarget() = func
    and call.getEnclosingFunction() != func
    and param.getType().getUnspecifiedType() instanceof IntegralType
    and this = param.getFunction().getEntryPoint()
    and not addressIsTaken(func)
  }

  override predicate hasRangeInformationFor(StackVariable v) { v = param }

  override float getLowerBounds(StackVariable v) {
    v = param
    and result = getFullyConvertedLowerBounds(call.getArgument(param.getIndex()))
  }

  override float getUpperBounds(StackVariable v) {
    v = param
    and result = getFullyConvertedUpperBounds(call.getArgument(param.getIndex()))
  }

  override predicate dependsOnExpr(StackVariable v, Expr e) {
    v = param
    and e = call.getArgument(param.getIndex())
  }
}
```

*Figure 32: Simple bounding of function parameters; see an alternative implementation here*

With support for bound propagation of function arguments, our query now produces **575 findings**. With the addition of the `ConstrainArgs` class, the range analysis is no longer sound; the `FunctionCall` class may not include all possible calls to the function; for

example, it will miss calls via function pointers. To mitigate that problem, we do not try to bound function arguments for functions whose address is ever taken (this is implemented with the `addressIsTaken` predicate).

## IR-Based Range Analysis

The value range propagation algorithm working on Intermediate Representation aims to represent variables' bounds in terms of relations between variables and numeric deltas:

```
a <= b + delta
a >= b + delta
```

The algorithm somewhat reassembles the abstract interpretation technique—it is "stepping" through the code like the control flow algorithm would, propagating relations, and weakening them on some "back edges" (when a control flow loops back).

For example, in the following code, the analysis can find that the lower bound of a value assigned to the `size` variable is `value <= ValueNumber(len) + 1`.

```
int len = vsnprintf(NULL, 0, f, tmplist);
// Code omitted
if (len < 0)
{
    goto out;
}

// Code omitted
size_t size = len + 1;
```

*Figure 33: Example code for IR-based range analysis (openvpn/src/openvpn/argv.c#378–389)*

`ValueNumber` is an abstract class containing every instruction producing equivalent results. This means we cannot learn a specific number of the `len` variable's lower bound; the analysis only provides information that the "`len` variable at this point in the program is equivalent to results of these other instructions."

IR-based analysis seems to be experimental[3]. For example, it does not allow us to easily concretize bounds (i.e., produce a numeric bound from the relative bounds).

To get a usable number from the analysis, we will combine the simple range analysis with IR-based range analysis to find bounds for `ValueNumbers`, as shown in figure 34. This update implements new `safeBounds` predicates that use two kinds of range analysis. Note that we had to skip IR-based analysis for known values because the delta in CodeQL is of type `int`, so the analysis will not work correctly for large constants.

[3] See footnote 2.

While this approach is obviously oversimplified, it trims down the number of findings to **435**. Note that the execution time of the query is now about three times longer; for large codebases, IR-based range analysis may have to be disabled.

```
private import experimental.semmle.code.cpp.rangeanalysis.RangeAnalysis
private import semmle.code.cpp.ir.IR

predicate safeLowerBound(Expr cast, IntegralType toType) {
  exists(float lowerB |
    lowerB = lowerBound(cast) and
    lowerB >= typeLowerBound(toType)
  )
  or
  exists(Instruction instr, Bound b, int delta |
    not exists(float knownValue | knownValue = cast.getValue().toFloat())
    and instr.getUnconvertedResultExpression() = cast and
    boundedInstruction(instr, b, delta, false, _) and // false = lower bound

lowerBound(b.getInstruction().getUnconvertedResultExpression().getExplicitlyConverte
d()) + delta >=
      typeLowerBound(toType)
  )
}

predicate safeUpperBound(Expr cast, IntegralType toType) {
  exists(float upperB |
    upperB = upperBound(cast) and
    upperB <= typeUpperBound(toType)
  )
  or
  exists(Instruction instr, Bound b, int delta |
    not exists(float knownValue | knownValue = cast.getValue().toFloat())
    and instr.getUnconvertedResultExpression() = cast and
    boundedInstruction(instr, b, delta, true, _) and // true = upper bound

upperBound(b.getInstruction().getUnconvertedResultExpression().getExplicitlyConverte
d()) + delta <=
      typeUpperBound(toType)
  )
}
```

*Figure 34: IR-based range analysis combined with simple range analysis*

## Modeling OpenVPN2 and Other Codebases

The next idea for improving our query is to model the OpenVPN2 codebase and related codebases in CodeQL. This is where CodeQL really shines—the ease with which one can improve static analysis with a few simple additions informing the analysis engine about facts that are complex to deduce automatically but are otherwise simple.

We will start by adding the `BufLenFunc` class, which represents a function call to one of a few OpenVPN2 functions operating on buffers, to our query. These functions can return only positive values. With this class, our query produces **372 findings**.

```
private class BufLenFunc extends SimpleRangeAnalysisExpr, FunctionCall {
  BufLenFunc() {
    this.getTarget()
        .getName()
        .matches([
            "buf_len", "buf_reverse_capacity", "buf_forward_capacity",
"buf_forward_capacity_total"
          ])
  }

  override float getLowerBounds() { result = 0 }

  override float getUpperBounds() { result =
typeUpperBound(this.getExpectedReturnType()) }

  override predicate dependsOnChild(Expr child) { none() }
}
```

*Figure 35: Addition of the `BufLenFunc` class to model OpenVPN2 buffer functions*

We will now model functions from the C standard library that return -1 on error and otherwise return a positive number bounded by their second argument.

```
private class LenApproxFunc extends SimpleRangeAnalysisExpr, FunctionCall {
  LenApproxFunc() {
    this.getTarget().hasName(["recvfrom", "recv", "sendto", "send", "read", "write",
"readv"])
  }

  override float getLowerBounds() { result = -1 }

  override float getUpperBounds() { result =
getFullyConvertedUpperBounds(this.getArgument(2)) }

  override predicate dependsOnChild(Expr child) { child = this.getArgument(2) }
}
```

*Figure 36: Addition of the `LenApproxFunc` class to model standard library functions*

Next, we will add an assumption that the `strlen` function is used only with short strings (shorter than $2^{32}/8$ bytes). While not necessarily impossible, it is unlikely that we could exploit vulnerabilities that require us to send large strings.

```
private class StrlenFunAssumption extends SimpleRangeAnalysisExpr, FunctionCall {
  StrlenFunAssumption() {
    this.getTarget().hasName("strlen")
  }
```

```
  override float getLowerBounds() { result = 0 }

  override float getUpperBounds() { result = 536870912 }

  override predicate dependsOnChild(Expr child) { none() }
}
```

*Figure 37: Addition of the `StrlenFunAssumption` class to model the `strlen` assumption*

Lastly, we will model some OpenSSL methods that should return only small positive values.

```
private class OpenSSLFunc extends SimpleRangeAnalysisExpr, FunctionCall {
  OpenSSLFunc() {
    this.getTarget()
        .getName()
        .matches([
            "EVP_CIPHER_get_block_size", "cipher_ctx_block_size",
"EVP_CIPHER_CTX_get_block_size",
            "EVP_CIPHER_block_size", "HMAC_size", "hmac_ctx_size",
"EVP_MAC_CTX_get_mac_size",
            "EVP_CIPHER_CTX_mode", "EVP_CIPHER_CTX_get_mode",
"EVP_CIPHER_iv_length",
            "cipher_ctx_iv_length", "EVP_CIPHER_key_length", "EVP_MD_size",
"EVP_MD_get_size",
            "cipher_kt_iv_size", "cipher_kt_block_size", "EVP_PKEY_get_size",
"EVP_PKEY_get_bits",
            "EVP_PKEY_get_security_bits"
          ])
  }

  override float getLowerBounds() { result = 0 }

  override float getUpperBounds() { result = 32768 }

  override predicate dependsOnChild(Expr child) { none() }
}
```

*Figure 38: Addition of the `OpenSSLFunc` class to model OpenSSL methods*

With these three classes, our query produces **316 results**.

Finally, we will add some heuristics: ignore findings in unused functions, in calls to `msg` and `buf_safe` functions (known-safe functions), and in the `options.c` file (where inputs can be assumed to be trusted).

```
// skip unused function
(
  exists(FunctionCall fc | fc.getTarget() = cast.getEnclosingFunction())
  or
  exists(FunctionAccess fc | fc.getTarget() = cast.getEnclosingFunction())
```

```
  or
  cast.getEnclosingFunction().getName() = "main"
  or
  addressIsTaken(cast.getEnclosingFunction())
) and
// skip casts in call to msg and buf_safe (known safe casts)
not exists(MacroInvocation msgCall |
  msgCall.getMacro().hasName("msg") and
  msgCall.getStmt().getAChild*() = cast.getEnclosingStmt()
) and
not exists(FunctionCall bufSafeCall |
  bufSafeCall.getTarget().hasName("buf_safe") and
  bufSafeCall.getAnArgument() = castExpr
) and
// not interesting file
not cast.getLocation().getFile().getBaseName().matches("options.c")
```

*Figure 39: Addition of simple heuristics for the OpenVPN2 codebase*

With these heuristics, we have trimmed down the number of unsafe implicit casts reported by our query to **254**.

## User-Controlled Inputs

If a manual review of 254 findings is too much effort, then there is one last improvement we can make: limiting results to conversions that involve user-controlled (a.k.a., untrusted) inputs. This should eliminate findings that are unlikely to be exploitable.

For this task, we can use the `FlowSource` class. This class—split into remote and local subclasses—includes all functions that may return user-controlled (untrusted) data. The class models both function return values and modified arguments (e.g., buffer pointers). However, the class does not include indirectly controlled data like return values from calls to the `read` function. These return values are not simply provided by a user but rather indicate a reading error or a number of bytes read. We want to include such indirectly controlled numbers in our analysis; for that, we will use the `RemoteFlowSourceFunction` and `LocalFlowSourceFunction` classes.

With these two classes, we can use taint tracking to find flows from user-controlled inputs to implicit conversions.

**This final improvement brings the number of findings our query produces to 20.** That is a small enough number for manual review, through which we found that all of these remaining findings are false positives.

```
module UserInputToImplicitConversionConfig implements DataFlow::ConfigSig {
  predicate isSource(DataFlow::Node source) {
    exists(RemoteFlowSourceFunction remoteFlow | remoteFlow =
source.asExpr().(Call).getTarget())
    or
```

```
    exists(LocalFlowSourceFunction localFlow | localFlow =
source.asExpr().(Call).getTarget())
    or
    source instanceof RemoteFlowSource
    or
    source instanceof LocalFlowSource
  }

  predicate isSink(DataFlow::Node sink) {
    exists(IntegralConversion cast |
      cast.isImplicit() and
      cast.getExpr() = sink.asExpr()
    )
  }
}

module UserInputToImplicitConversionConfigFlow =
TaintTracking::Global<UserInputToImplicitConversionConfig>;

where
  [skipped]
  and exists(UserInputToImplicitConversionConfigFlow::PathNode source,
UserInputToImplicitConversionConfigFlow::PathNode sink |
    castExpr = sink.getNode().asExpr() and
    UserInputToImplicitConversionConfigFlow::flowPath(source, sink)
  )
```

*Figure 40: Addition of taint tracking for user-controlled inputs*

Actually, if we had used the query above without any of the previously made improvements, we could get 57 findings right away. However, the `FlowSource` class may not cover all interesting data sources. On a security code review, we would rather use the query version without this filtering and manually review all the results.

## Recommendations for Safe Conversions

As we can see, implicit conversions can cause security problems and are not easy to detect with static analysis.

Because of this, developers should build their projects with `-Wconversion -Wsign-compare` flags to prevent insertion of problematic implicit conversions altogether. See the Fuchsia and Genodians blogs for instructive journeys on how they eliminated implicit conversions in large projects.

In general, when writing C or C++ code, follow these guidelines:

- Harmonize types to eliminate implicit conversions and reduce explicit conversions.
- Avoid mixing signedness.
- Default to using only signed integers when in doubt (see Google's C++ style guide and Jeffrey Hurchalla's article; note that there are opposing opinions on this approach, like in Dale Weiler's article).
- For explicit conversions, document expected ranges of values or why the conversions are safe.

Lastly, use the `trailofbits/cpp-queries` CodeQL query pack, which includes the query presented in this report.

# About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up to date with our latest news and announcements, please follow @trailofbits on X or LinkedIn, and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

**Trail of Bits, Inc.**
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

# Notices and Remarks

## Copyright and Distribution