Operational View of CodeQL

Thinking of CodeQL as Preprocessor, Compiler, and Runtime

Michael Hohn, hohn@github.com

Things to come

You already know a lot about codeql…

… you just didn't realize it.

You already use C/Python/Java etc. as a combination of preprocessor + compiler + libraries

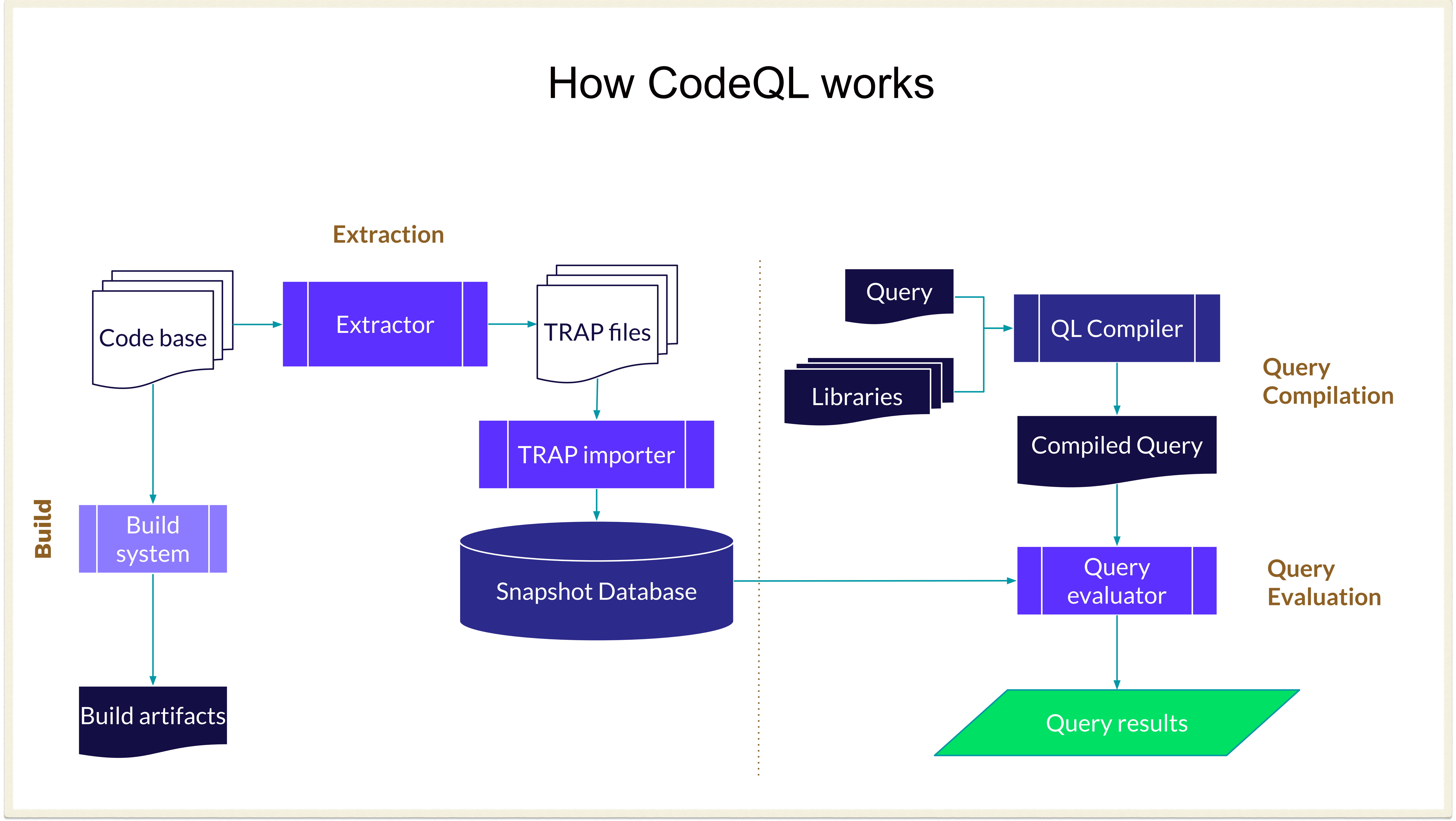
CodeQL is a preprocessor + compiler + libraries

This leads to many analogies and patterns you already know and understand — just apply them to codeql

In the following, we use this analogy to get some best (codeql) practices for

- query re-use
- query structuring
- query customization
- tool use points (what tool when and where)
- larger system integration

You may have seen this slide



It’s correct but abstract

You might be thinking…

What does this mean?

I write what and put it where?

How does this fit into a large system?

You already know most of those answers when
you think of C/Python/Java:

C is a preprocessor + compiler + libraries

Now we look at this:

CodeQL is a preprocessor + compiler + libraries

CodeQL is a preprocessor + compiler + libraries

Most (all?) of what you know about setup and use of compilers and scripting languages applies 1-1 to codeql

With that in mind, let's jump right in

Think Compiler (C) with library:

```
# Prepare System
./admin -c

# Convert data if needed
cat users.txt

# Edit your code
edit add-user.c

# Compile & run your code
clang -Wall add-user.c \
    -lc \
    -lsqlite3 -o add-user
for user in `cat input.txt` ; do
    echo "$user" | \
    ./add-user 2>> users.log ; done

# Examine results
./admin -s
```

The general sequence

1. set up the system
2. prepare data
3. edit code
4. compile & run
5. examine results

Note: this is the sequence that is always run, whether in the CLI, github actions, or VS Code

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&                                                    \
    codeql database create --language=cpp                        \
           -s . -j 8 -v                                          \
           $DB                                                   \
           --command='clang -Wall add-user.c -lsqlite3 -o add-user'

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze                         \
       -v --ram=14000 -j12 --rerun              \
       --search-path ~/local/vmsync/ql          \
       --format=sarif-latest                    \
       --output=$RESULTS                        \
       --                                       \
       $DB                                      \
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.jq < cpp-sqli.sarif | less
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

Think of preprocessor + compiler + libraries

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&                                              \
    codeql database create --language=cpp                  \
           -s . -j 8 -v                                    \
           $DB                                             \
           --command='clang -Wall add-user.c -lsqlite3 -o add-user'

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze                        \
       -v --ram=14000 -j12 --rerun             \
       --search-path ~/local/vmsync/ql         \
       --format=sarif-latest                   \
       --output=$RESULTS                       \
       --                                      \
       $DB                                     \
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.jq < cpp-sqli.sarif | less
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

Q: What does CodeQL do for us?

Partial Answer: What does clang/gcc do for us?

Quite a lot, actually. The core:

1. fully lexes and parses the source code
2. gives us an Abstract Syntax Tree to work with
3. provides a concise domain-specific language

On top of the language, we have the fundamental libraries; they

1. give us the Control Flow Graph
2. give us a Data Flow Graph

And we have the modeling libraries; they

1. provide a high-level view of libraries / frameworks
2. provide static analysis tools, e.g., range analysis, guard conditions

Last not least, there are many queries; they

1. find commonly encountered bugs (language specific)
2. find possible CWE vulnerabilities

Think of preprocessor + compiler + libraries

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&                                                   \
    codeql database create --language=cpp                       \
           -s . -j 8 -v                                         \
           $DB                                                  \
           --command='clang -Wall add-user.c -lsqlite3 -o add-user'

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze                         \
       -v --ram=14000 -j12 --rerun              \
       --search-path ~/local/vmsync/ql          \
       --format=sarif-latest                    \
       --output=$RESULTS                        \
       --                                       \
       $DB                                      \
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.jq < cpp-sqli.sarif | less
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

Q: The C language is a great start. Is the C standard library supported?

Structural Answer: the C standard library is linked at runtime via the -L search path. CodeQL libraries also have a search path.

A: To find the supported APIs, search the ql/ library source tree. Practically, much of the standard C library is supported, typically from a conceptual level. Don't try to find 1-1 mappings between C headers and the ql/ library.

```
Ex: For example, for a top-down search start with cpp.qll and notice the statement

    import semmle.code.cpp.commons.Printf

Follow this to find the cpp.commons module and see what it models:
Alloc.qll       Dependency.qll   NullTermination.qll   StringAnalysis.qll
Assertions.qll  Environment.qll  PolymorphicClass.qll  StructLikeClass.qll
Buffer.qll      Exclusions.qll   Printf.qll            Synchronization.qll
CommonType.qll  File.qll         Scanf.qll             VoidContext.qll
DateTime.qll    NULL.qll         Strcat.qll            unix/
```

Think of preprocessor + compiler + libraries

Q: Is library X supported?

A: If it is, you'll find it in the ql/ library source tree. A whole-tree search, grep-style, is easiest.

```
E: For example, to check support for sqlite:
0:$ cd ~/local/vmsync/ql/cpp/ql/src
0:$ grep -l -R sqlite *
Security/CWE/CWE-313/CleartextSqliteDatabase.ql
Security/CWE/CWE-313/CleartextSqliteDatabase.c
semmle/code/cpp/security/Security.qll
```

So we have a query (.ql) and a library (.qll); look at both to get some ideas:

Security/CWE/CWE-313/CleartextSqliteDatabase.ql has some info in the header

```
/**
 * @name Cleartext storage of sensitive information in an SQLite database
 * @description Storing sensitive information in a non-encrypted
 *              database can expose it to an attacker.
 */
```

and a promising class:

```
class SqliteFunctionCall extends FunctionCall {
    SqliteFunctionCall()
{ this.getTarget().getName().matches("sqlite%") }

    Expr getASource() { result = this.getAnArgument() }
}
```

semmle/code/cpp/security/Security.qll has notes on extending and offers a source/sink framework:

```
/**
 * Extend this class to customize the security queries for
 * a particular code base. Provide no constructor in the
 * subclass, and override any methods that need customizing.
 */
class SecurityOptions extends string {
    predicate sqlArgument(string function, int arg) {
        // SQLite3 C API
        function = "sqlite3_exec" and arg = 1
    }
    /**
     * The argument of the given function is filled in from user input.
     */
    predicate userInputArgument(FunctionCall functionCall, int arg) {
        fname = "scanf" and arg >= 1
    }
}
```

Aside: this class and its documentation have been updated

semmle/code/cpp/security/Security.qll is a library, so some sample uses would be nice. Another search via

```
    grep  -nH  -R SecurityOptions *
```

- finds (potential) documentation:

```
    docs/codeql/ql-training/cpp/global-data-flow-cpp.rst:59:The library
class ``SecurityOptions`` provides a (configurable) model of what counts as
user-controlled data:
```

- and an extension point:

```
    cpp/ql/src/semmle/code/cpp/security/SecurityOptions.qll:16:class
CustomSecurityOptions extends SecurityOptions
/**
 * This class overrides `SecurityOptions` and can be used to add project
 * specific customization.
 */
class CustomSecurityOptions extends SecurityOptions {...}
```

Think of preprocessor + compiler + libraries

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&                                                       \
    codeql database create --language=cpp                           \
           -s . -j 8 -v                                             \
           $DB                                                      \
           --command='clang -Wall add-user.c -lsqlite3 -o add-user'

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze                        \
       -v --ram=14000 -j12 --rerun             \
       --search-path ~/local/vmsync/ql         \
       --format=sarif-latest                   \
       --output=$RESULTS                       \
       --                                      \
       $DB                                     \
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.jq < cpp-sqli.sarif | less
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

Q: What do we have to help with?

A: What does your code use beyond the C/Python/Java standard library?

- For this example, the `sqlite3` library.
- Provide the entry / exit points of your own APIs. CodeQL won't trace through unknown external functions.

Q: What else should we do?

Write queries for known & patched vulnerabilities. This will uncover points in your code where CodeQL gets stuck (those you encode in your custom codeql library for other queries) and provide a regression test for the vulnerability

Think of preprocessor + compiler + libraries

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&                                                    \
    codeql database create --language=cpp                        \
           -s . -j 8 -v                                          \
           $DB                                                   \
           --command='clang -Wall add-user.c -lsqlite3 -o add-user'

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze                         \
       -v --ram=14000 -j12 --rerun              \
       --search-path ~/local/vmsync/ql          \
       --format=sarif-latest                    \
       --output=$RESULTS                        \
       --                                       \
       $DB                                      \
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.jq < cpp-sqli.sarif | less
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

Q: How should we go about modeling our libraries with CodeQL?

A: Follow the way you use a C library, say `sqlite3`.
Your code only `#includes sqlite3.h`;
you use, but don't care about, `libsqlite3.a`.

Thus for CodeQL: don't try to model the library internals, only model the parts of the API you actually use.

Q: How should we structure our CodeQL queries?

A: Follow the way you structure a C/Python/Java library
Use separate query files (*.ql) and library files (*.qll)

A: Structure the query set by size and complexity
Some examples are given in this gist; use the simplest one that fits your problem.

Think of preprocessor + compiler + libraries

Q: We don't want to reinvent the wheel. How do we extend existing queries?

A: For C/Python etc.
Common approaches to extending libraries are configurations, callbacks & other hooks.
When those fail, patching the source.

A: For CodeQL
Customizations can be injected when libraries use abstract base classes. When those are extended before use by queries, the additions are part of the queries.

Q: Say what?

A: The steps to customize a query:

Examine the query and libraries it uses

Subclass abstract base classes, if there are any (otherwise, see next answer)

Add these to Customizations/Options.qll so they are included by all queries. The files by language are:

```
ql/csharp/ql/src/Customizations.qll
ql/java/ql/src/Customizations.qll
ql/javascript/ql/src/Customizations.qll
ql/python/ql/src/Customizations.qll
ql/cpp/ql/src/Options.qll
```

A: In some cases, you will need heavy modifications.
Clone the ql/ tree, patch it as needed, and use your customized version.

Tool coverage: Keep Thinking of preprocessor + compiler + libraries

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&
    codeql database create --language=cpp
           -s . -j 8 -v
           $DB
           --command='clang -Wall add-user.c -ls

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze
       -v --ram=14000 -j12 --rerun
       --search-path ~/local/vmsync/ql
       --format=sarif-latest
       --output=$RESULTS
       --
       $DB
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

The general sequence	shell/scripts	vs code	emacs/vi/lsp editors
1. set up the system	✓		
2. prepare data	✓		
3. edit code		✓	✓
4. compile & run	✓	✓	*
5. examine results	*	✓	
Use cases	direct control, setup, debugging, automation, result transformation	CodeQL editing with jump-to-definition etc. and integrated result review on desktop	CodeQL editing with jump-to-definition etc.

Integration: Keep Thinking of
preprocessor + compiler + libraries

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&
    codeql database create --language=cpp
           -s . -j 8 -v
           $DB
           --command='clang -Wall add-user.c -ls

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze
       -v --ram=14000 -j12 --rerun
       --search-path ~/local/vmsync/ql
       --format=sarif-latest
       --output=$RESULTS
       --
       $DB
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

The general sequence	shell/scripts	github actions or any ci/cd	github security alerts
1. set up the system	✓	✓	
2. prepare data	✓	✓	
3. edit code			
4. compile & run	✓	✓	
5. examine results	✱		✓
Use cases	direct control, setup, debugging, automation, result transformation	fully automated pipeline for the three indicated steps	developer review of alerts, linking github, query, and source code

Key takeaways

You already use C/Python/Java etc. as a combination of preprocessor + compiler + libraries

CodeQL is a preprocessor + compiler + libraries

The general development sequence is

1. set up the system
2. prepare data
3. edit code
4. compile & run
5. examine results

Apply your existing best practices to CodeQL

The end… Questions?

On to the GHAS overview

Integration: Keep Thinking of preprocessor + compiler + libraries

Think Compiler (CodeQL) with library:

```
# Prepare System
export PATH=$HOME/local/vmsync/codeql250:"$PATH"

# Convert data if needed
SRCDIR=.
DB=add-user.db
cd $SRCDIR &&                                              \
    codeql database create --language=cpp                  \
           -s . -j 8 -v                                    \
           $DB                                             \
           --command='clang -Wall add-user.c -lsqlite3 -o add-user'

# Edit your code
edit SqlInjection.ql

# Compile & run your code
RESULTS=cpp-sqli.sarif
codeql database analyze                        \
       -v --ram=14000 -j12 --rerun             \
       --search-path ~/local/vmsync/ql         \
       --format=sarif-latest                   \
       --output=$RESULTS                       \
       --                                      \
       $DB                                     \
       $SRCDIR/SqlInjection.ql

# Examine results
# Plain text, look for
#     "results" : [ {
#     and
#     "codeFlows" : [ {
edit $RESULTS
# Or
jq --raw-output --join-output  -f sarif-summary.jq < cpp-sqli.sarif | less
# Or use vs code's sarif viewer
# Or use the GHAS integration via actions
```

Q: Should we use the most recent version of codeql at all times?

A: Follow the way you use your compiler. Do you use the most recent version of compiler at all times, or do you use a rolling release cycle?

To get your current version's info:

```
hohn@gh-hohn ~/local/vmsync/ql/cpp/ql/src
0:$ codeql --version
CodeQL command-line toolchain release 2.5.0.
Copyright (C) 2019-2021 GitHub, Inc.
Unpacked in: /Users/hohn/local/vmsync/codeql250
   Analysis results depend critically on separately distributed query
and
   extractor modules. To list modules that are visible to the
toolchain,
   use 'codeql resolve qlpacks' and 'codeql resolve languages'.
```

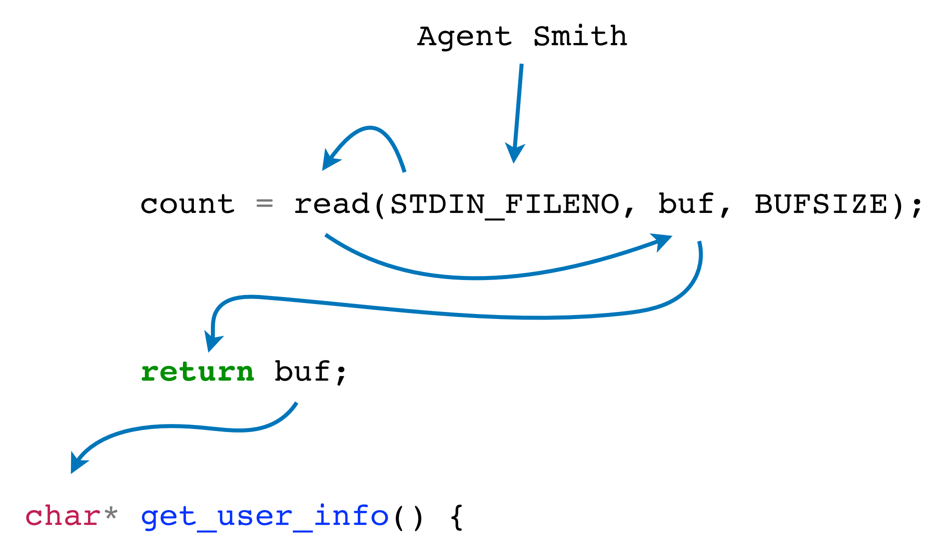
You should match the CodeQL cli version to the CodeQL library version; the library releases have `codeql-cli/<VERSION>` tags to allow matching with the binaries.

When using git for the library, you should check out the appropriate version via, e.g.,

```
cd $HOME/local/vmsync/ql && git checkout codeql-cli/v2.5.9
```

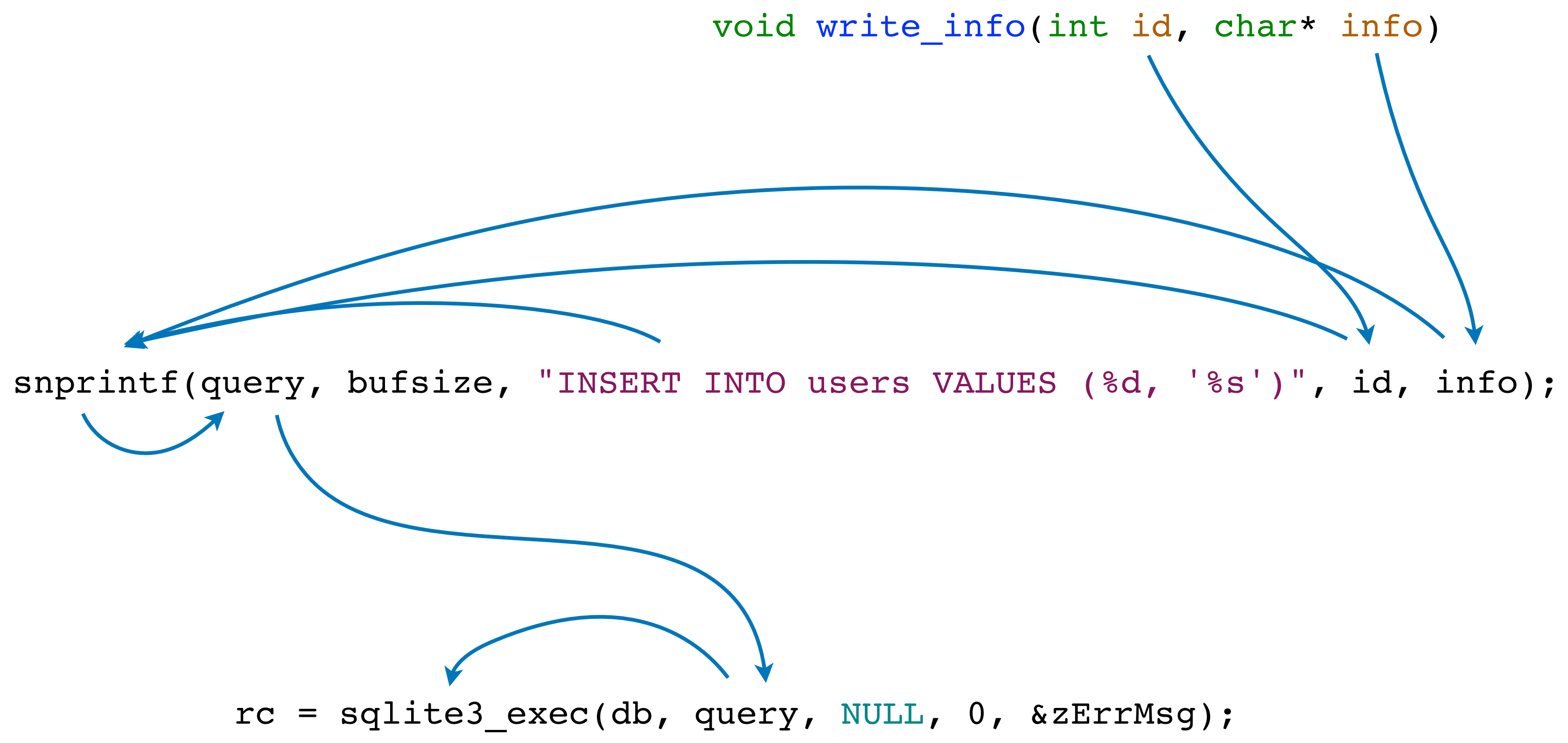
Flow in get_user_info

```c
char* get_user_info() {
#define BUFSIZE 1024
    char* buf = (char*) malloc(BUFSIZE * sizeof(char));
    int count;
    // Disable buffering to avoid need for fflush
    // after printf().
    setbuf( stdout, NULL );
    printf("*** Welcome to sql injection ***\n");
    printf("Please enter name: ");
    count = read(STDIN_FILENO, buf, BUFSIZE);
    if (count <= 0) abort();
    /* strip trailing whitespace */
    while (count && isspace(buf[count-1])) {
        buf[count-1] = 0; --count;
    }
    return buf;
}
```



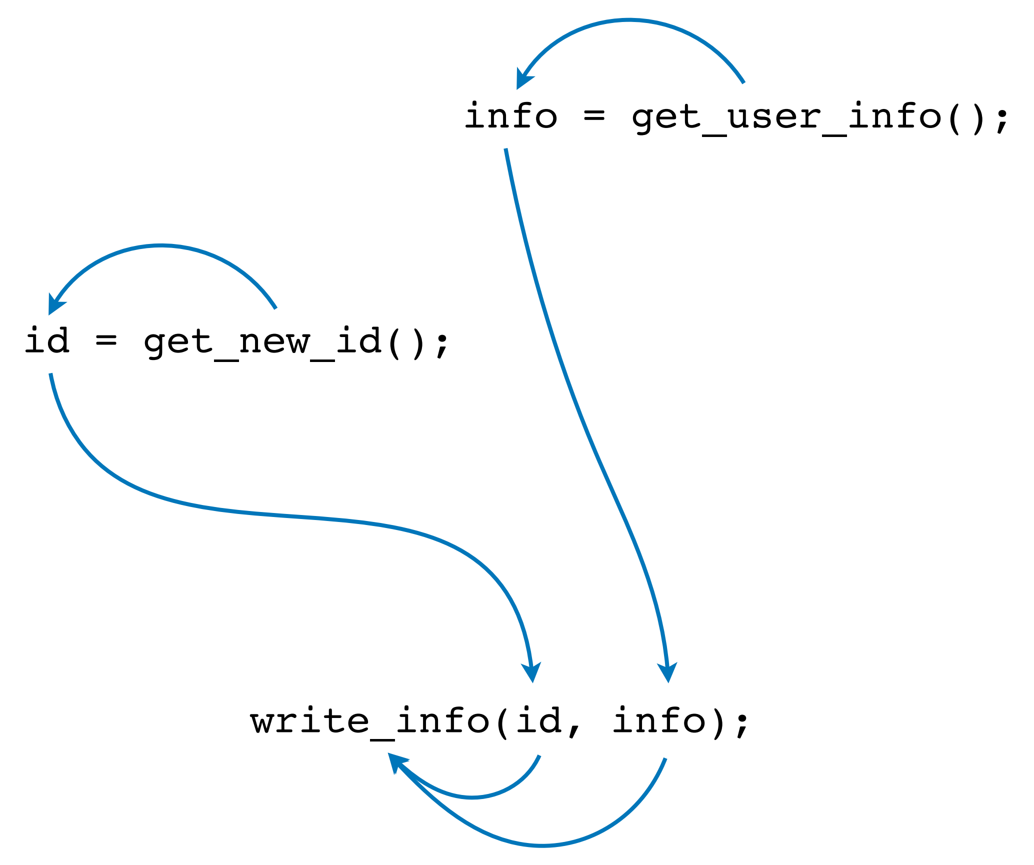
Flow in write_info

```c
void write_info(int id, char* info) {
    sqlite3 *db;
    int rc;
    int bufsize = 1024;
    char *zErrMsg = 0;
    char query[bufsize];

    /* open db */
    rc = sqlite3_open("users.sqlite", &db);
    abort_on_error(rc, db);

    /* Format query */
    snprintf(query, bufsize,
             "INSERT INTO users VALUES (%d, '%s')",
             id, info);
    write_log("query: %s\n", query);

    /* Write info */
    rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
    abort_on_exec_error(rc, db, zErrMsg);

    sqlite3_close(db);
}
```



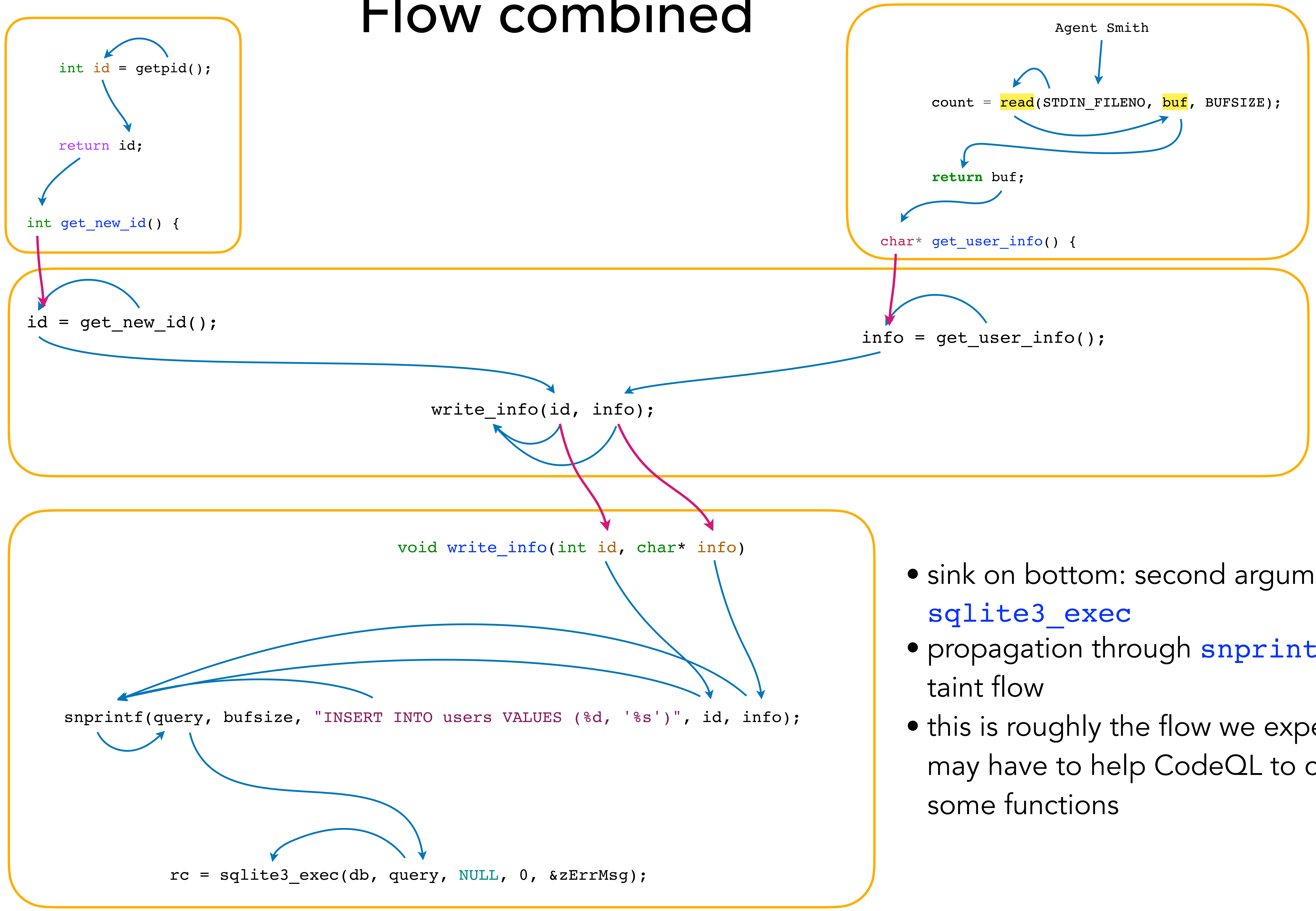
Flow in `main`

```
int main(int argc, char* argv[]) {
    char* info;
    int id;
    info = get_user_info();
    id = get_new_id();
    write_info(id, info);
}
```



Flow combined

- inter-procedural (global) data flow
- source on top: second argument to `read`



- sink on bottom: second argument to `sqlite3_exec`
- propagation through `snprintf` needs taint flow
- this is roughly the flow we expect to see; may have to help CodeQL to capture flow across some functions