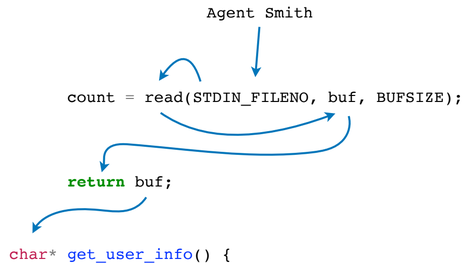
# Operational View of CodeQL

## or: thinking of CodeQL as compiler and runtime

**Michael Hohn, hohn@github.com**

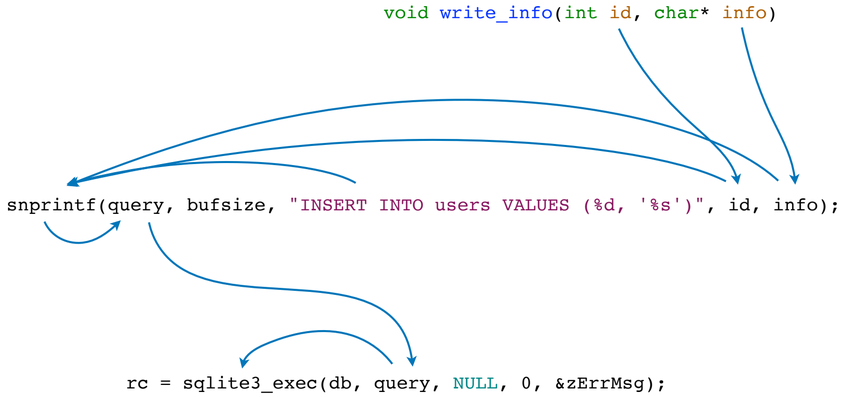
# Flow in `get_user_info`

```
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;
}
```



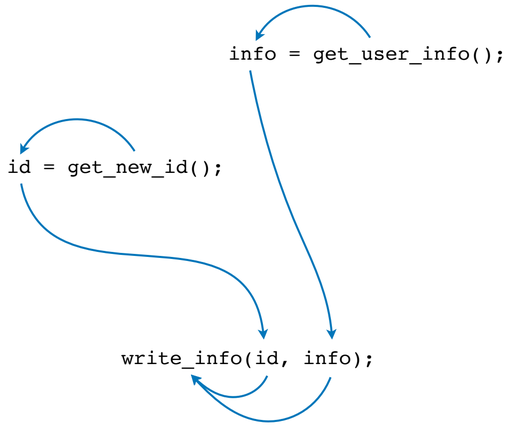
# Flow in `write_info`

```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);
}
```



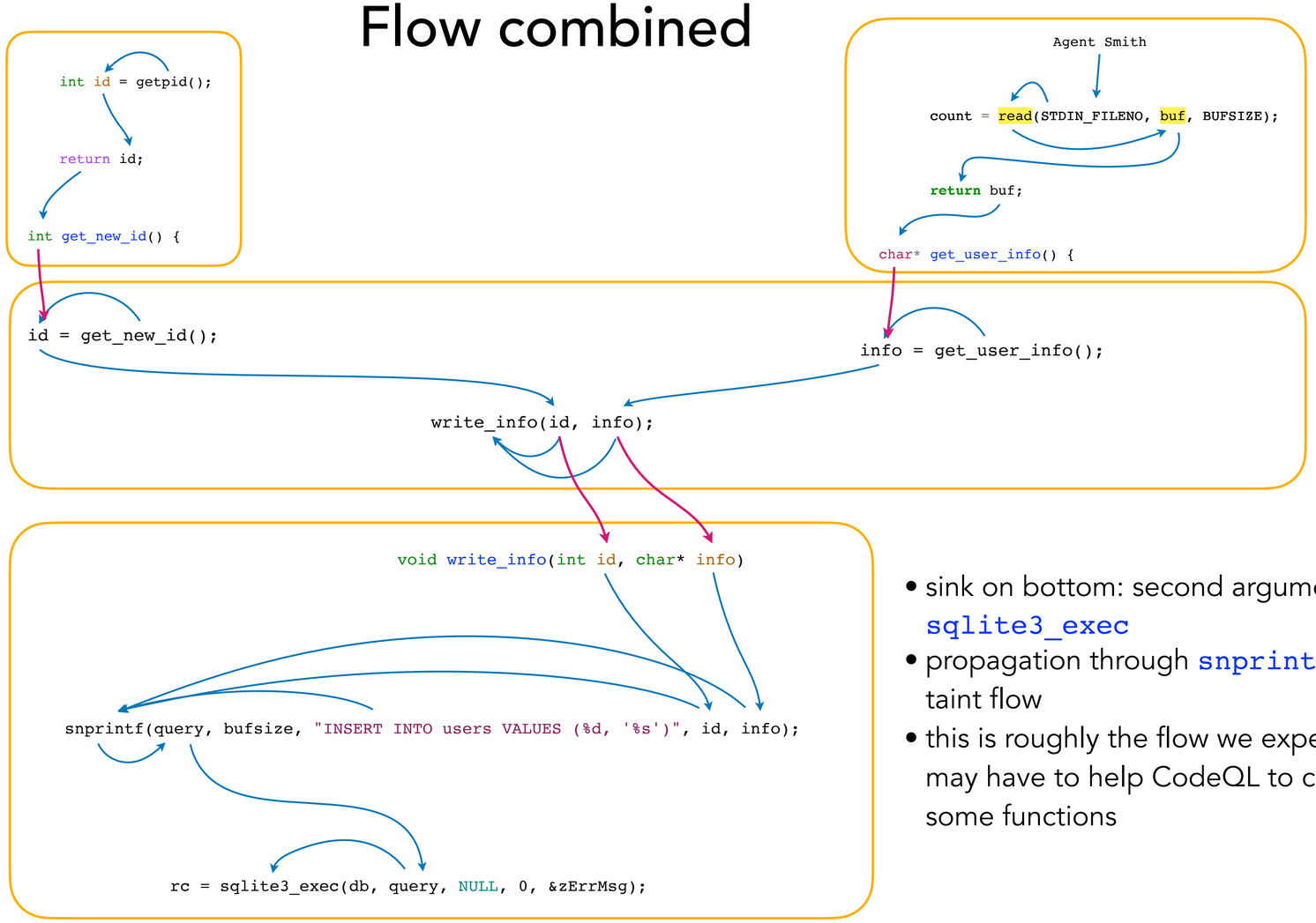
# Flow in `main`

```
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

# What does CodeQL do for us?

What do we have to help with?

This depends on the problem…

But thinking about

codeql as compiler

gives us a good start in clarifying the problem

## 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
```

## 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
```

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

Q: Is the C standard library supported?
A: Much of it, typically from a conceptual level.
To find the supported APIs, search the `ql/` library source tree.
For example, for a top-down search start with `cpp.qll` and notice the import `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/
```

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.
For example, to check support for sqlite:

```
hohn@gh-hohn ~/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 some very promising entries

```
/**
 * 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
        ;;
    }
    ;;
}
```

This is a library, so some sample uses would be nice. Another search via

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

finds documentation:

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

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 {...}
```