

# Running the program

```
---- SQL injection demo ----  
1:$ # Build  
1:$ ./build.sh
```

# Running the program

```
---- SQL injection demo ----
```

```
1:$ # Build
```

```
1:$ ./build.sh
```

```
---- SQL injection demo ----
```

```
0:$ # Prepare db
```

```
0:$ ./admin rm-db
```

```
---- SQL injection demo ----
```

```
0:$ ./admin create-db
```

```
---- SQL injection demo ----
```

```
0:$ ./admin show-db
```

# Running the program

```
---- SQL injection demo ----
```

```
1:$ # Build
```

```
1:$ ./build.sh
```

```
---- SQL injection demo ----
```

```
0:$ # Prepare db
```

```
0:$ ./admin rm-db
```

```
---- SQL injection demo ----
```

```
0:$ ./admin create-db
```

```
---- SQL injection demo ----
```

```
0:$ ./admin show-db
```

```
---- SQL injection demo ----
```

```
0:$ # Add regular user interactively
```

```
0:$ ./add-user 2>> users.log
```

```
*** Welcome to sql injection ***
```

```
Please enter name: First User
```

# Running the program

---- SQL injection demo ----

1:\$ # Build

1:\$ ./build.sh

---- SQL injection demo ----

0:\$ # Prepare db

0:\$ ./admin rm-db

---- SQL injection demo ----

0:\$ ./admin create-db

---- SQL injection demo ----

0:\$ ./admin show-db

---- SQL injection demo ----

0:\$ # Add regular user interactively

0:\$ ./add-user 2>> users.log

\*\*\* Welcome to sql injection \*\*\*

Please enter name: First User

---- SQL injection demo ----

0:\$ # Check

0:\$ ./admin show-db

81750|First User

# Running the program

---- SQL injection demo ----

1:\$ # Build

1:\$ ./build.sh

---- SQL injection demo ----

0:\$ # Prepare db

0:\$ ./admin rm-db

---- SQL injection demo ----

0:\$ ./admin create-db

---- SQL injection demo ----

0:\$ ./admin show-db

---- SQL injection demo ----

0:\$ # Add regular user interactively

0:\$ ./add-user 2>> users.log

\*\*\* Welcome to sql injection \*\*\*

Please enter name: First User

---- SQL injection demo ----

0:\$ # Check

0:\$ ./admin show-db

81750|First User

---- SQL injection demo ----

0:\$ # Regular user via "external" process

0:\$ echo "User Outside" | ./add-user 2>> users.log

\*\*\* Welcome to sql injection \*\*\*

Please enter name:

# Running the program

---- SQL injection demo ----

```
1:$ # Build
1:$ ./build.sh
```

---- SQL injection demo ----

```
0:$ # Prepare db
0:$ ./admin rm-db
```

---- SQL injection demo ----

```
0:$ ./admin create-db
```

---- SQL injection demo ----

```
0:$ ./admin show-db
```

---- SQL injection demo ----

```
0:$ # Add regular user interactively
0:$ ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name: First User
```

---- SQL injection demo ----

```
0:$ # Check
0:$ ./admin show-db
81750|First User
```

---- SQL injection demo ----

```
0:$ # Regular user via "external" process
0:$ echo "User Outside" | ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name:
```

---- SQL injection demo ----

```
0:$ ./admin show-db
81750|First User
81757|User Outside
```

# Running the program

---- SQL injection demo ----

```
1:$ # Build
1:$ ./build.sh
```

---- SQL injection demo ----

```
0:$ # Prepare db
0:$ ./admin rm-db
```

---- SQL injection demo ----

```
0:$ ./admin create-db
```

---- SQL injection demo ----

```
0:$ ./admin show-db
```

---- SQL injection demo ----

```
0:$ # Add regular user interactively
0:$ ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name: First User
```

---- SQL injection demo ----

```
0:$ # Check
0:$ ./admin show-db
81750|First User
```

---- SQL injection demo ----

```
0:$ # Regular user via "external" process
0:$ echo "User Outside" | ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name:
```

---- SQL injection demo ----

```
0:$ ./admin show-db
81750|First User
81757|User Outside
```

---- SQL injection demo ----

```
0:$ # Add Johnny Droptable
0:$ ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name: Johnny'); DROP TABLE users; --
```

# Running the program

---- SQL injection demo ----

```
1:$ # Build
1:$ ./build.sh
```

---- SQL injection demo ----

```
0:$ # Prepare db
0:$ ./admin rm-db
```

---- SQL injection demo ----

```
0:$ ./admin create-db
```

---- SQL injection demo ----

```
0:$ ./admin show-db
```

---- SQL injection demo ----

```
0:$ # Add regular user interactively
0:$ ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name: First User
```

---- SQL injection demo ----

```
0:$ # Check
0:$ ./admin show-db
81750|First User
```

---- SQL injection demo ----

```
0:$ # Regular user via "external" process
0:$ echo "User Outside" | ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name:
```

---- SQL injection demo ----

```
0:$ ./admin show-db
81750|First User
81757|User Outside
```

---- SQL injection demo ----

```
0:$ # Add Johnny Droptable
0:$ ./add-user 2>> users.log
*** Welcome to sql injection ***
Please enter name: Johnny'); DROP TABLE users; --
```

---- SQL injection demo ----

```
0:$ # And the problem:
0:$ ./admin show-db
Error: near line 2: no such table: users
```



# Oops! A command disguised as data

```
---- SQL injection demo ----
```

```
0:$ # Add Johnny Droptable
```

```
0:$ ./add-user 2>> users.log
```

```
*** Welcome to sql injection ***
```

```
Please enter name: Johnny'); DROP TABLE users; --
```

```
---- SQL injection demo ----
```

```
0:$ # And the problem:
```

```
0:$ ./admin show-db
```

```
Error: near line 2: no such table: users
```

# Flow in `get_new_id`

```
int get_new_id() {  
    int id = getpid();  
    return id;  
}
```

```
int id = getpid();
```


```
return id;
```

```
int get_new_id() {
```

# Flow in `get_new_id`

```
int get_new_id() {  
    int id = getpid();  
    return id;  
}
```

```
int id = getpid();
```

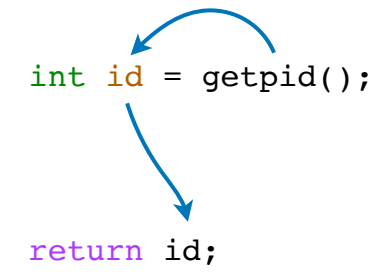


```
return id;
```

```
int get_new_id() {
```

# Flow in `get_new_id`

```
int get_new_id() {  
    int id = getpid();  
    return id;  
}
```



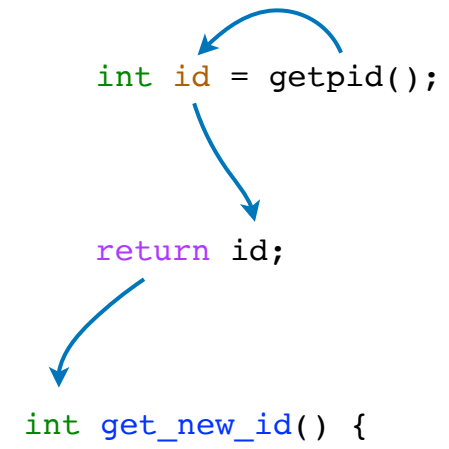
```
int id = getpid();  
return id;
```

The diagram illustrates the flow of execution between two lines of code. A blue curved arrow originates from the variable `id` in the assignment statement `int id = getpid();` and points to the variable `id` in the return statement `return id;`, indicating that the value of `id` is passed back to the caller.

```
int get_new_id() {
```

# Flow in `get_new_id`

```
int get_new_id() {  
    int id = getpid();  
    return id;  
}
```



# 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;  
}
```

Agent Smith

```
count = read(STDIN_FILENO, buf, BUFSIZE);
```


```
return buf;
```

```
char* get_user_info() {
```

# 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;  
}
```

Agent Smith




```
count = read(STDIN_FILENO, buf, BUFSIZE);  
  
return buf;  
  
char* get_user_info() {
```

# 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;
}
```

Agent Smith



```
count = read(STDIN_FILENO, buf, BUFSIZE);

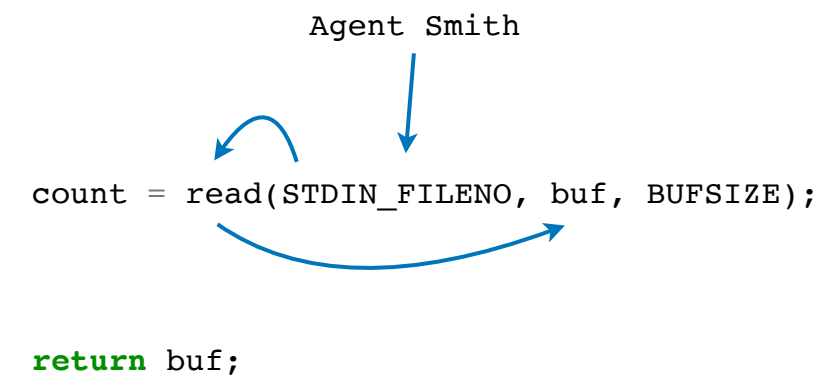
return buf;
```

```
char* get_user_info() {
```



# Flow in get\_user\_info

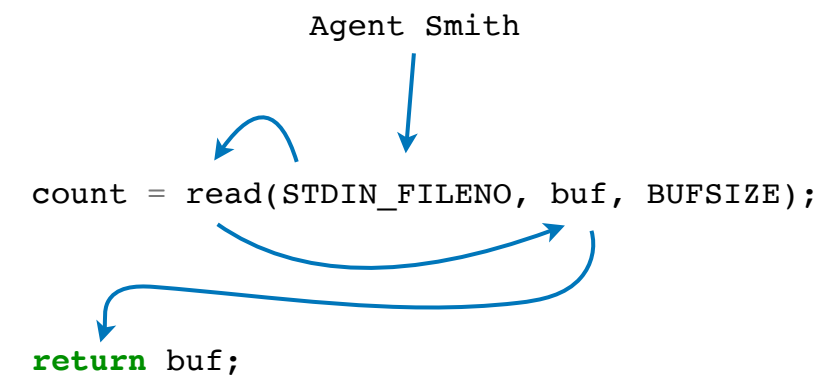
```
char* get_user_info() {  
#define BUFSIZE 1024  
    char* buf = (char*) malloc(BUFSIZE * sizeof(char));  
    int count;  
    // Disable buffering to avoid need for fflush  
    // after printf().  
    setbuf( stdout, NULL );  
    printf("*** Welcome to sql injection ***\n");  
    printf("Please 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;  
}
```



```
char* get_user_info() {
```

# Flow in get\_user\_info

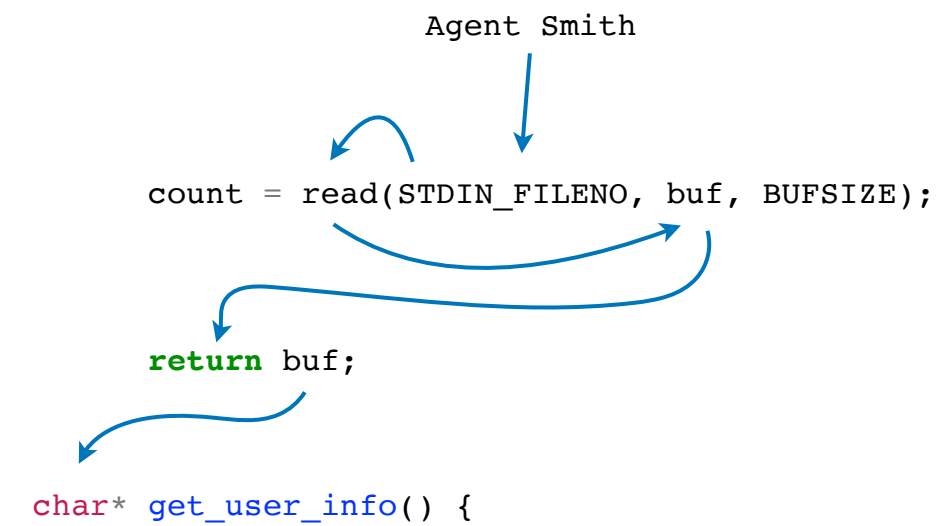
```
char* get_user_info() {  
#define BUFSIZE 1024  
    char* buf = (char*) malloc(BUFSIZE * sizeof(char));  
    int count;  
    // Disable buffering to avoid need for fflush  
    // after printf().  
    setbuf( stdout, NULL );  
    printf("*** Welcome to sql injection ***\n");  
    printf("Please 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;  
}
```



```
char* get_user_info() {
```

# 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`

```
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);
}
```

```
void write_info(int id, char* info)

    snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);

    rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

# Flow in write\_info

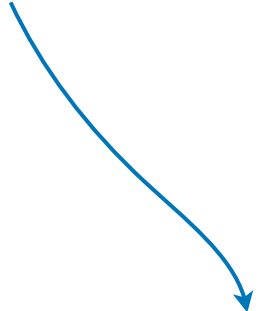
```
void write_info(int id, char* info) {
    sqlite3 *db;
    int rc;
    int bufsize = 1024;
    char *zErrMsg = 0;
    char query[bufsize];

    /* 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);
}
```

```
void write_info(int id, char* info)
    
    snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);
```

```
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

# Flow in `write_info`

```
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);
}
```

```
void write_info(int id, char* info)
    ↘
    ↘
    snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);
```

```
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

# Flow in `write_info`

```
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);
}
```

```
void write_info(int id, char* info)
```

The diagram illustrates the flow of data from a function call to its definition. Two blue arrows originate from the arguments 'id' and 'info' in the function call 'write\_info(id, info);' and point to the corresponding parameters 'id' and 'info' in the function definition 'void write\_info(int id, char\* info)'. This visualizes how the values are passed to the function's local variables.

```
int main() {  
    write_info(id, info);  
}
```

```
void write_info(int id, char* info){  
    // ...  
    printf("INSERT INTO users VALUES (%d, '%s')", id, info);  
}
```

```
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

# Flow in write\_info

```
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);
}
```

```
void write_info(int id, char* info)
                                     ↘
                                     ↘
                                     ↘
snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);
```

```
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```



# Flow in `write_info`

```
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);
}
```

Diagram illustrating the flow of data in the `write_info` function:

```
void write_info(int id, char* info)
```

Arrows indicate the flow of data from the function parameters to the `snprintf` call:

- The `id` parameter flows to the `%d` format specifier in the `snprintf` call.
- The `info` parameter flows to the `'%s'` format specifier in the `snprintf` call.

```
snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);
```

```
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

# Flow in write\_info

```
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);
}
```

Diagram illustrating the flow of data in the `write_info` function:

```
void write_info(int id, char* info)
```

The function signature is shown above the main code block. Arrows indicate the flow of data from the parameters `id` and `info` to the `snprintf` call in the code block. The `snprintf` call is shown below the function signature, with arrows indicating the flow of data from the parameters `id` and `info` to the `snprintf` call. The `snprintf` call is shown below the function signature, with arrows indicating the flow of data from the parameters `id` and `info` to the `snprintf` call.

```
snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);
```

The `snprintf` call is shown below the function signature, with arrows indicating the flow of data from the parameters `id` and `info` to the `snprintf` call. The `snprintf` call is shown below the function signature, with arrows indicating the flow of data from the parameters `id` and `info` to the `snprintf` call.

```
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

# Flow in `write_info`

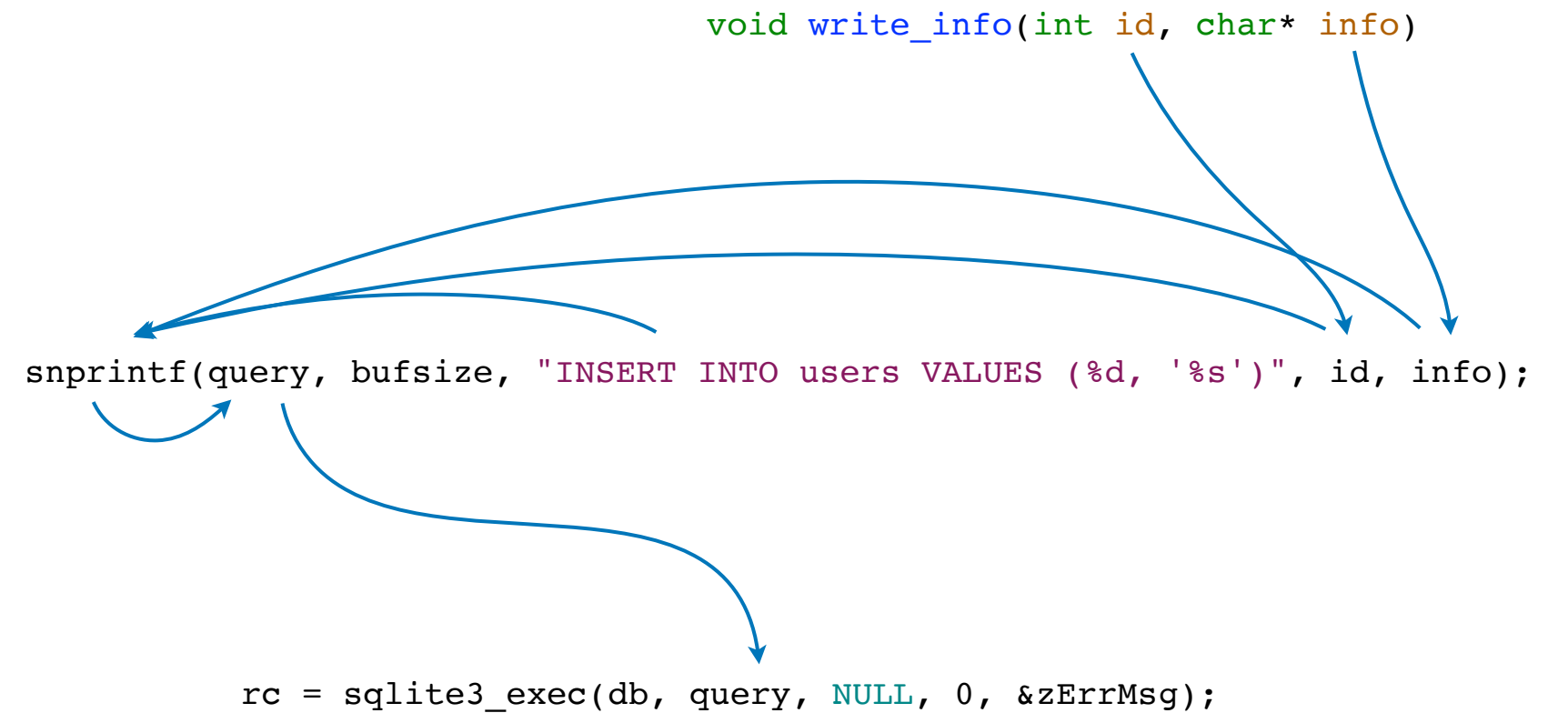
```
void write_info(int id, char* info) {
    sqlite3 *db;
    int rc;
    int bufsize = 1024;
    char *zErrMsg = 0;
    char query[bufsize];

    /* 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 `write_info`

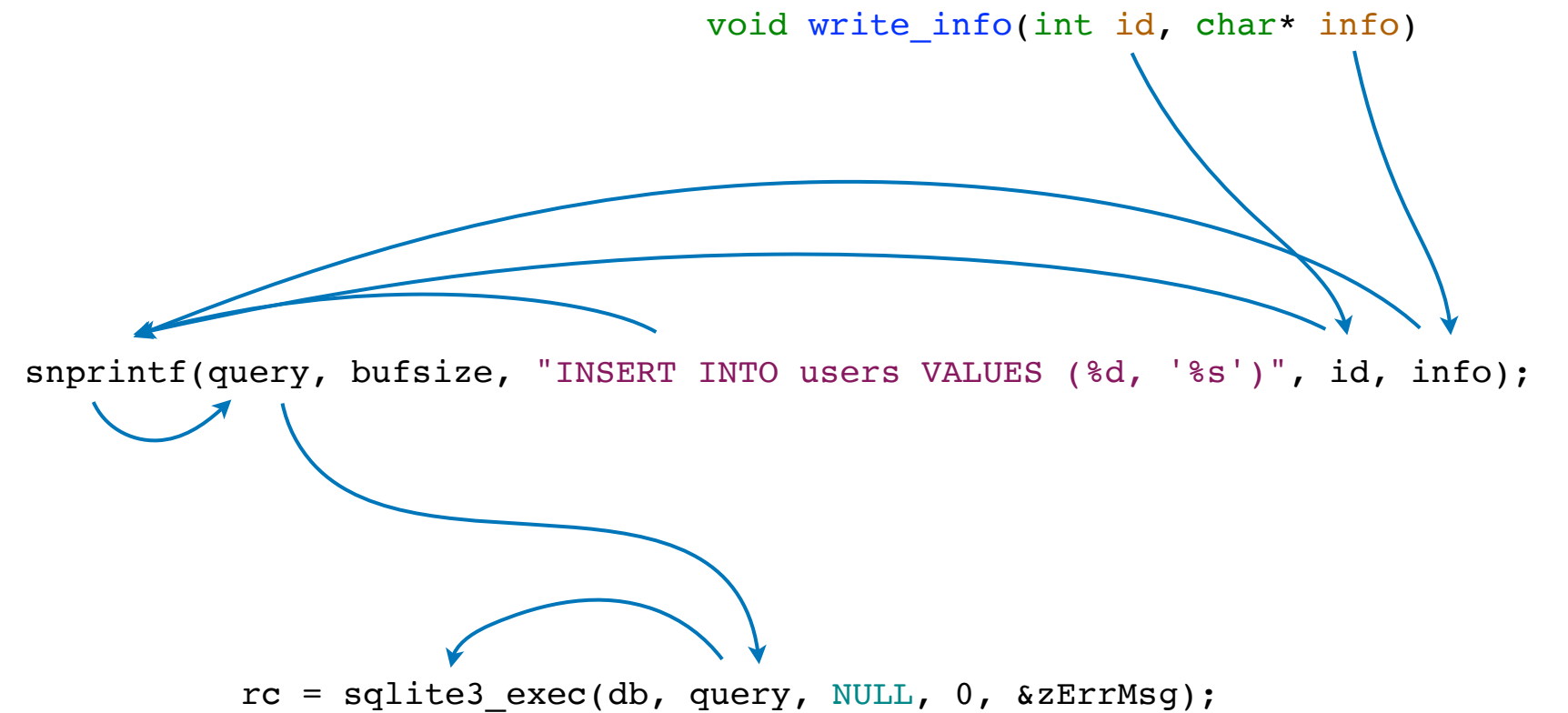
```
void write_info(int id, char* info) {
    sqlite3 *db;
    int rc;
    int bufsize = 1024;
    char *zErrMsg = 0;
    char query[bufsize];

    /* 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);  
}  
  
write_info(id, info);
```

# 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);  
}
```

  
`info = get_user_info();`

`id = get_new_id();`

`write_info(id, info);`

# 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);  
}
```

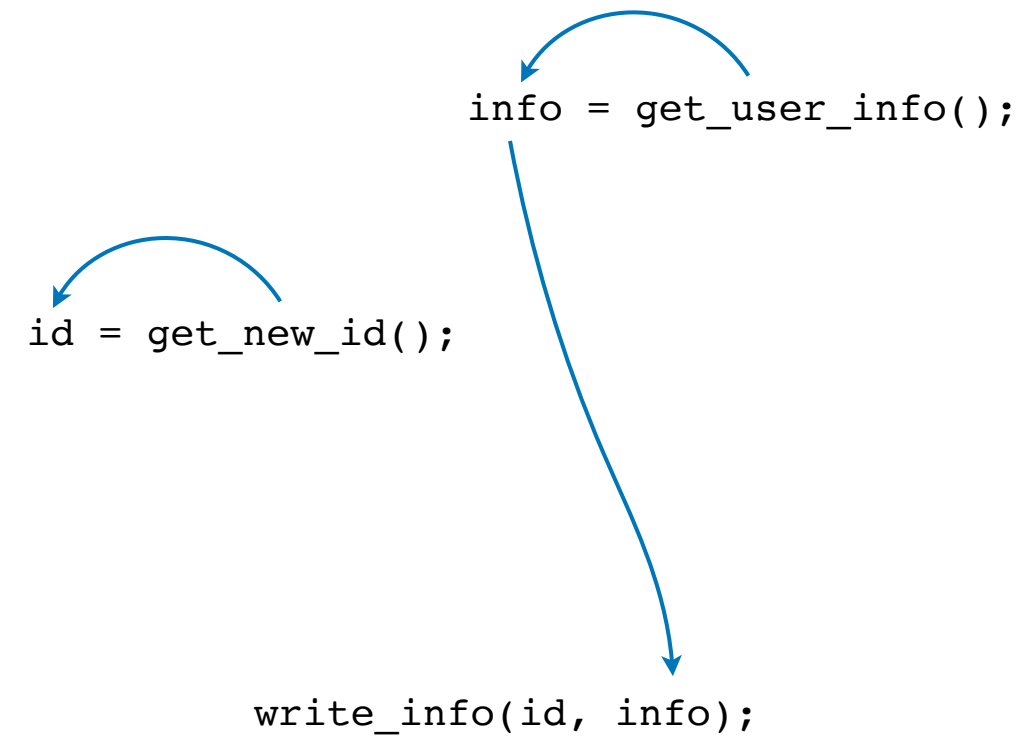
  
`info = get_user_info();`

  
`id = get_new_id();`

`write_info(id, info);`

# Flow in `main`

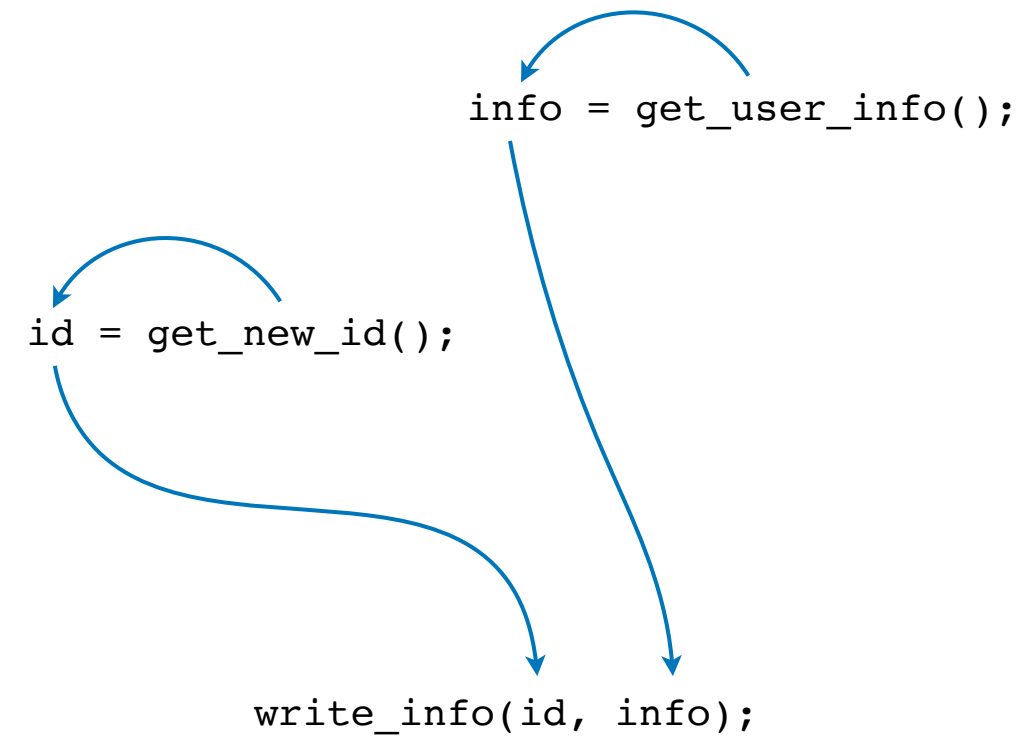
```
int main(int argc, char* argv[]) {  
    char* info;  
    int id;  
    info = get_user_info();  
    id = get_new_id();  
    write_info(id, info);  
}
```





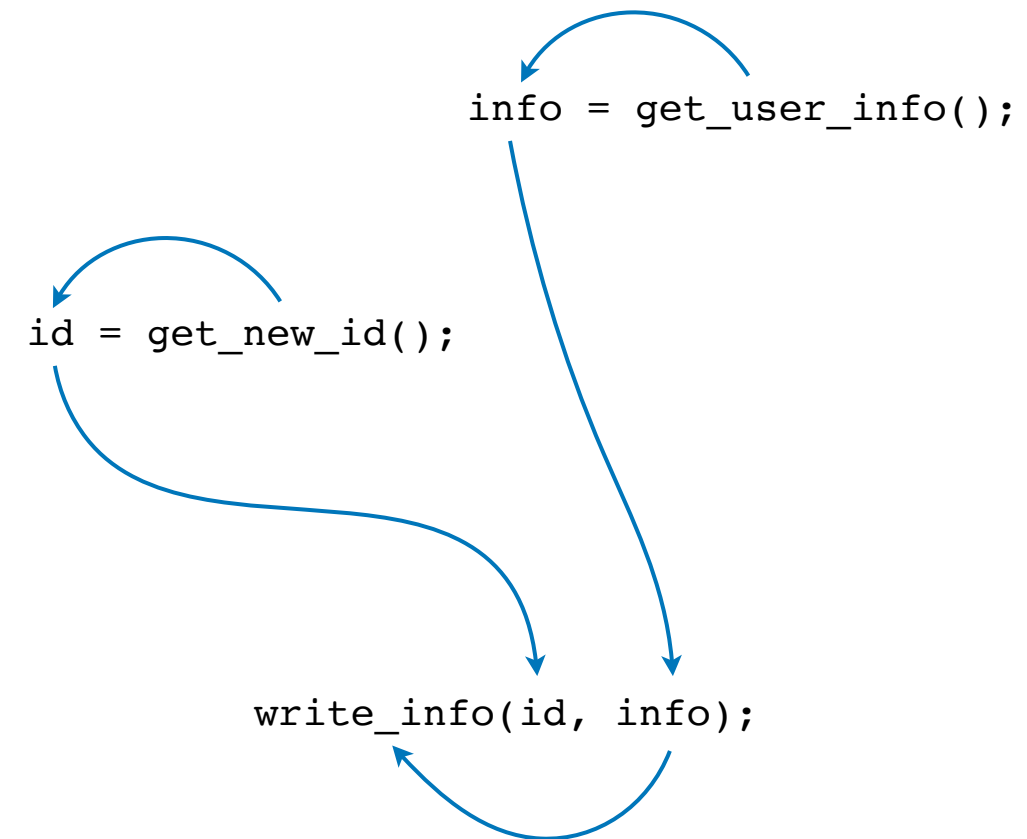
# Flow in `main`

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



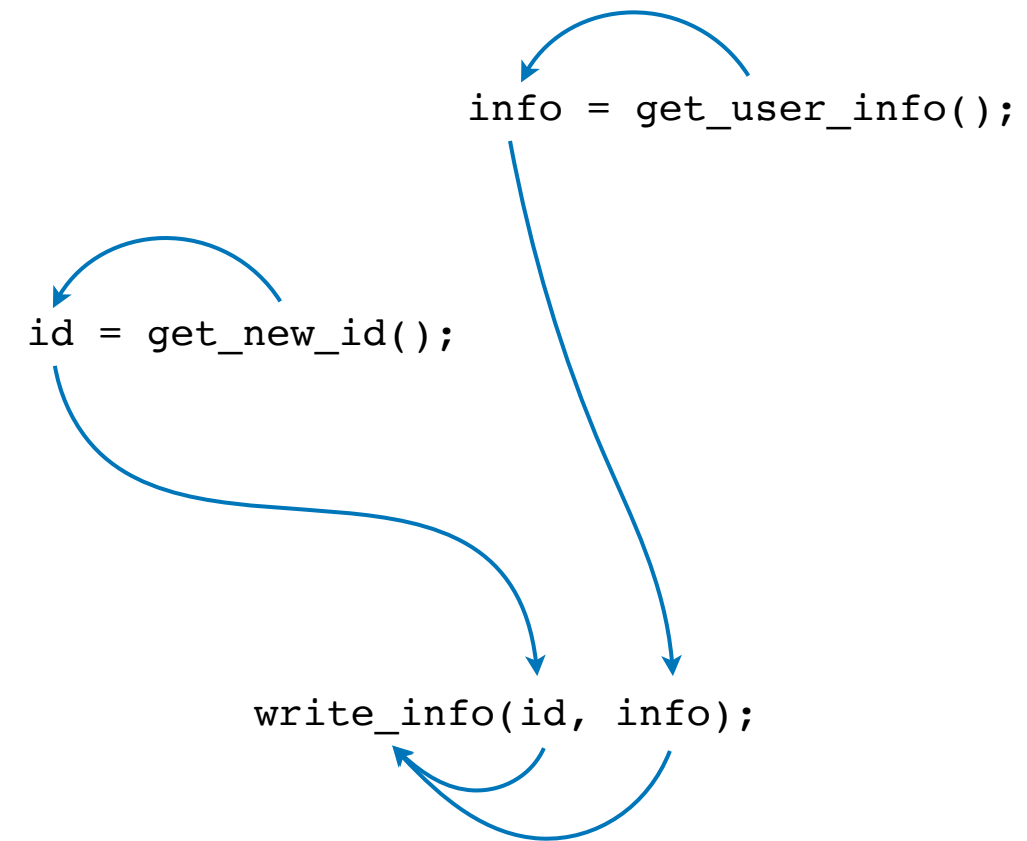
# Flow in `main`

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



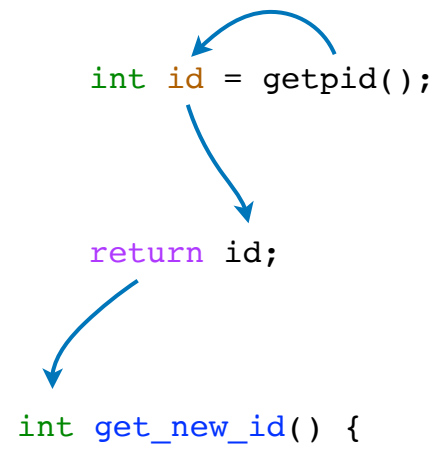
# Flow in `main`

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



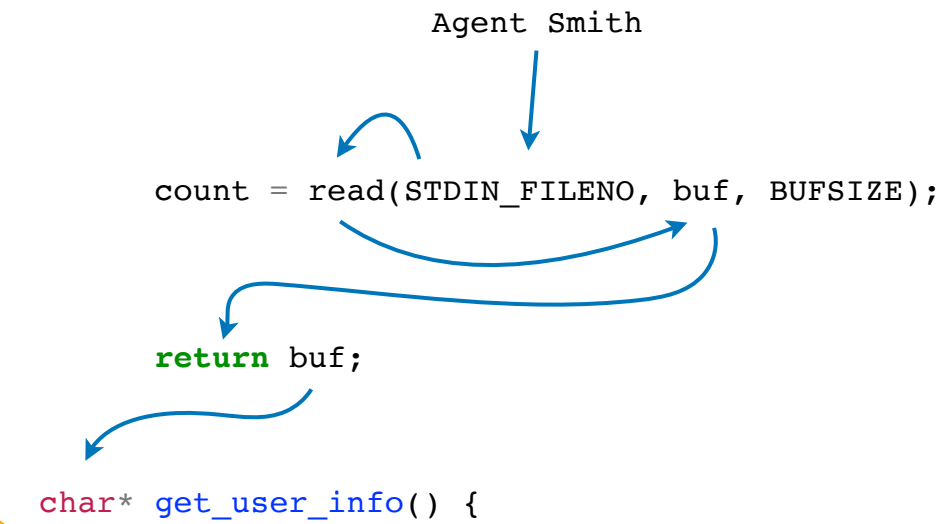
# Flow combined

```
int id = getpid();  
return id;  
int get_new_id() {
```



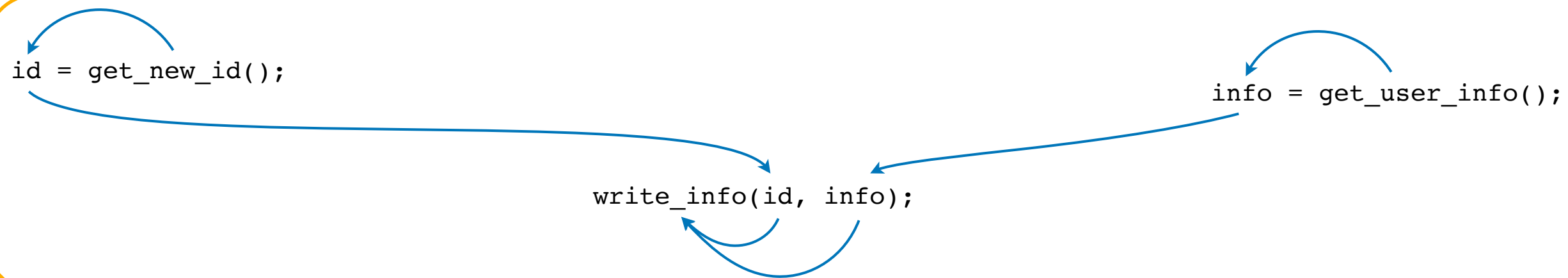
Flow diagram for the `get_new_id()` function. It shows a self-loop on the `int id = getpid();` line, an arrow pointing down to the `return id;` line, and another arrow pointing down to the start of the function definition `int get_new_id() {`.

```
Agent Smith  
count = read(STDIN_FILENO, buf, BUFSIZE);  
return buf;  
char* get_user_info() {
```



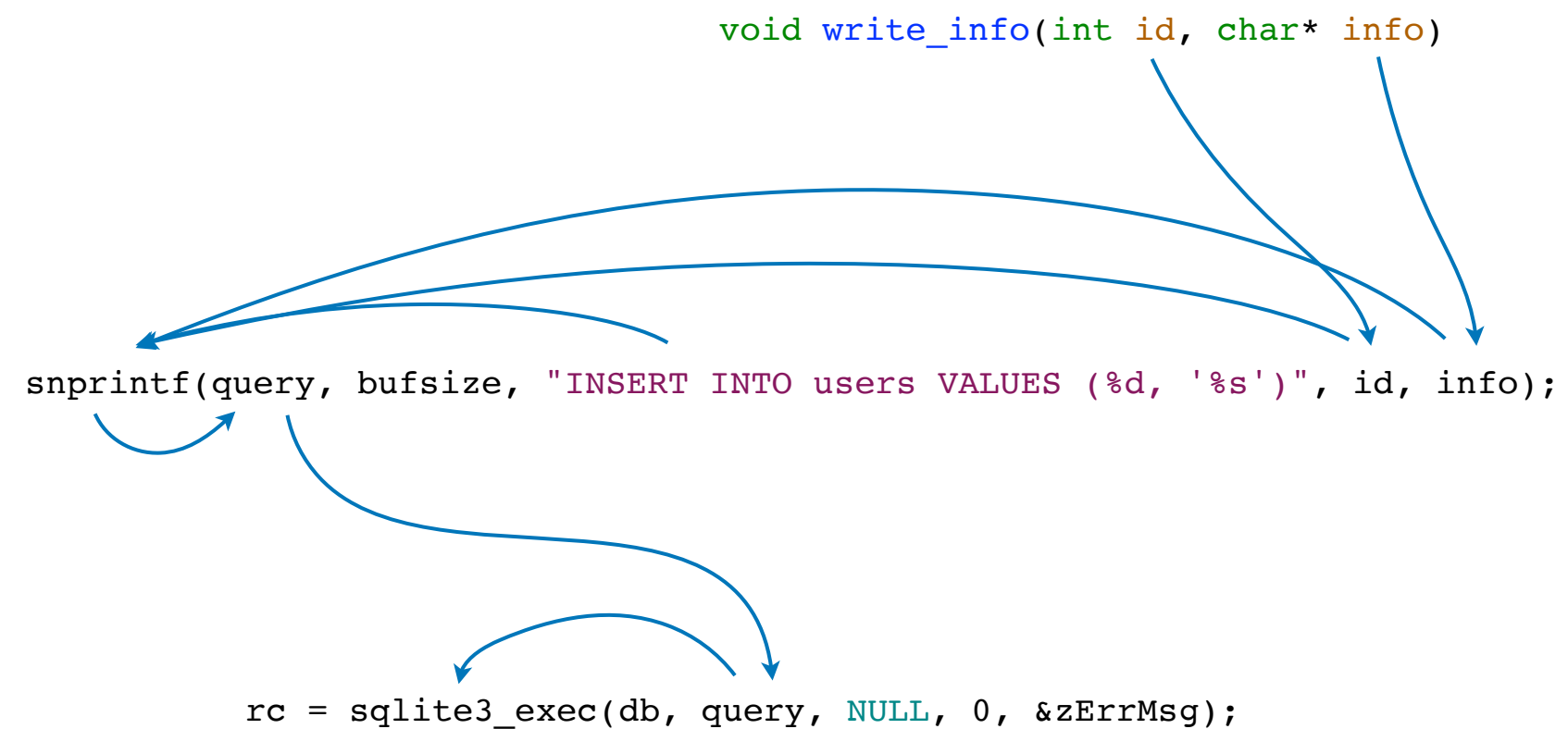
Flow diagram for the `get_user_info()` function. It starts with an arrow from "Agent Smith" to the `count = read(STDIN_FILENO, buf, BUFSIZE);` line. There is a self-loop on this line, an arrow pointing down to the `return buf;` line, and another arrow pointing down to the start of the function definition `char* get_user_info() {`.

```
id = get_new_id();  
info = get_user_info();  
write_info(id, info);
```



Flow diagram for the `write_info(id, info);` call. Arrows from `id = get_new_id();` and `info = get_user_info();` point to the `id` and `info` arguments respectively. There is a self-loop on the `write_info(id, info);` line.

```
void write_info(int id, char* info)  
snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);  
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```



Flow diagram for the `write_info` function. It shows a self-loop on the `snprintf` line. Arrows from the `id` and `info` parameters point to their respective arguments in the `snprintf` call. Arrows from the `query` and `bufsize` arguments of `snprintf` point to the `query` and `db` arguments of the `sqlite3_exec` call. There is a self-loop on the `rc = sqlite3_exec` line.

- 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

- inter-procedural (global) data flow

# Flow combined

```
int id = getpid();  
return id;  
  
int get_new_id() {
```

```
Agent Smith  
count = read(STDIN_FILENO, buf, BUFSIZE);  
return buf;  
  
char* get_user_info() {
```

```
id = get_new_id();  
  
info = get_user_info();  
  
write_info(id, info);
```

```
void write_info(int id, char* info)  
  
snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);  
  
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

- 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

- inter-procedural (global) data flow

# Flow combined

```
int id = getpid();  
return id;  
  
int get_new_id() {
```

```
graph TD; A[getpid()] --> B[id]; B --> C[return id]; C --> D[get_new_id()];
```

```
Agent Smith  
count = read(STDIN_FILENO, buf, BUFSIZE);  
return buf;  
  
char* get_user_info() {
```

```
graph TD; A[Agent Smith] --> B[read]; B --> C[return buf]; C --> D[get_user_info];
```

```
id = get_new_id();  
  
info = get_user_info();  
  
write_info(id, info);
```

```
graph TD; A[id = get_new_id()] --> B[id]; C[info = get_user_info()] --> D[info]; B --> E[write_info(id, info)]; D --> E;
```

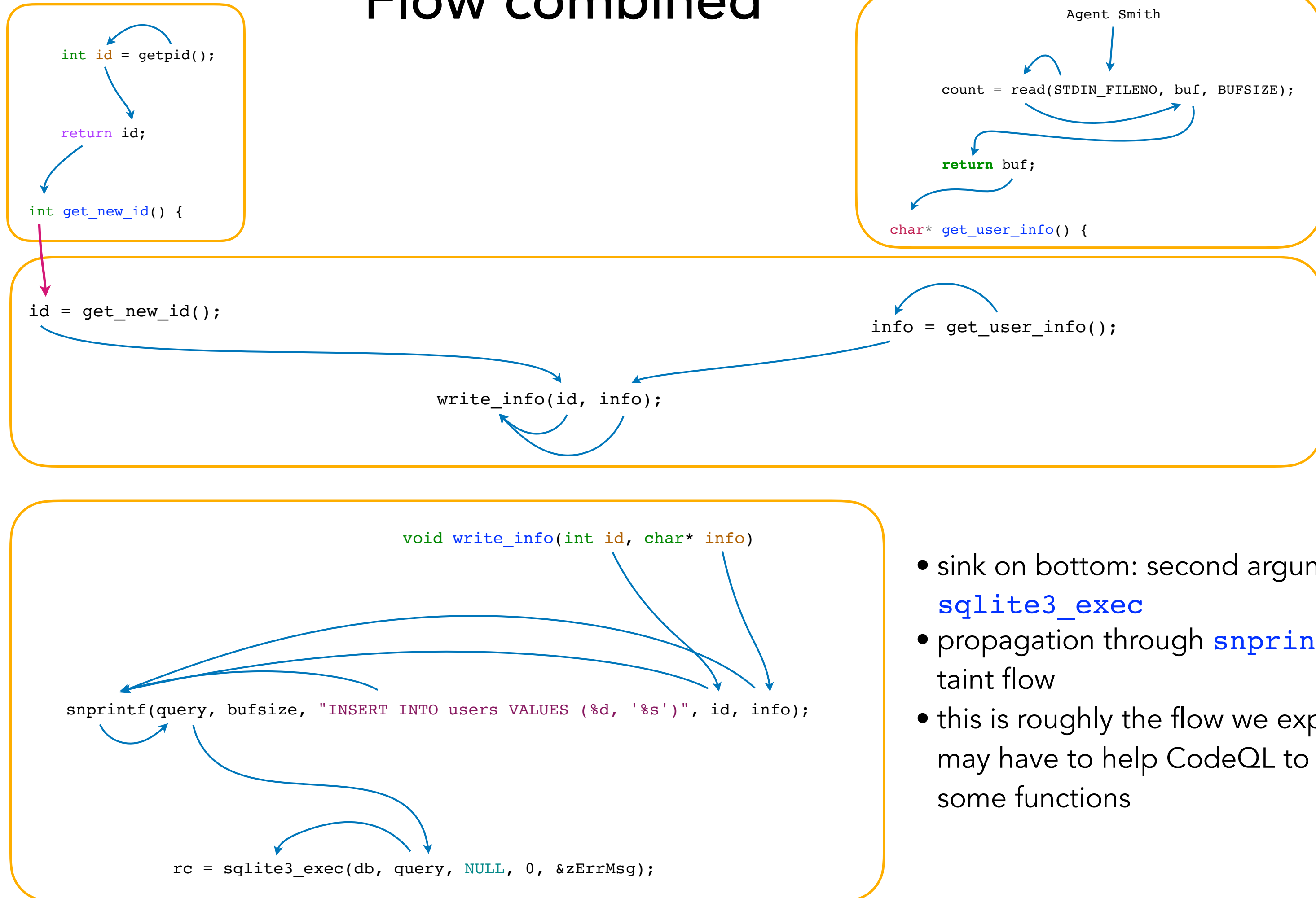
```
void write_info(int id, char* info)  
  
snprintf(query, bufsize, "INSERT INTO users VALUES (%d, '%s')", id, info);  
  
rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);
```

```
graph TD; A[id] --> C[snprintf]; B[info] --> C; C --> D[sqlite3_exec];
```

- 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

- inter-procedural (global) data flow

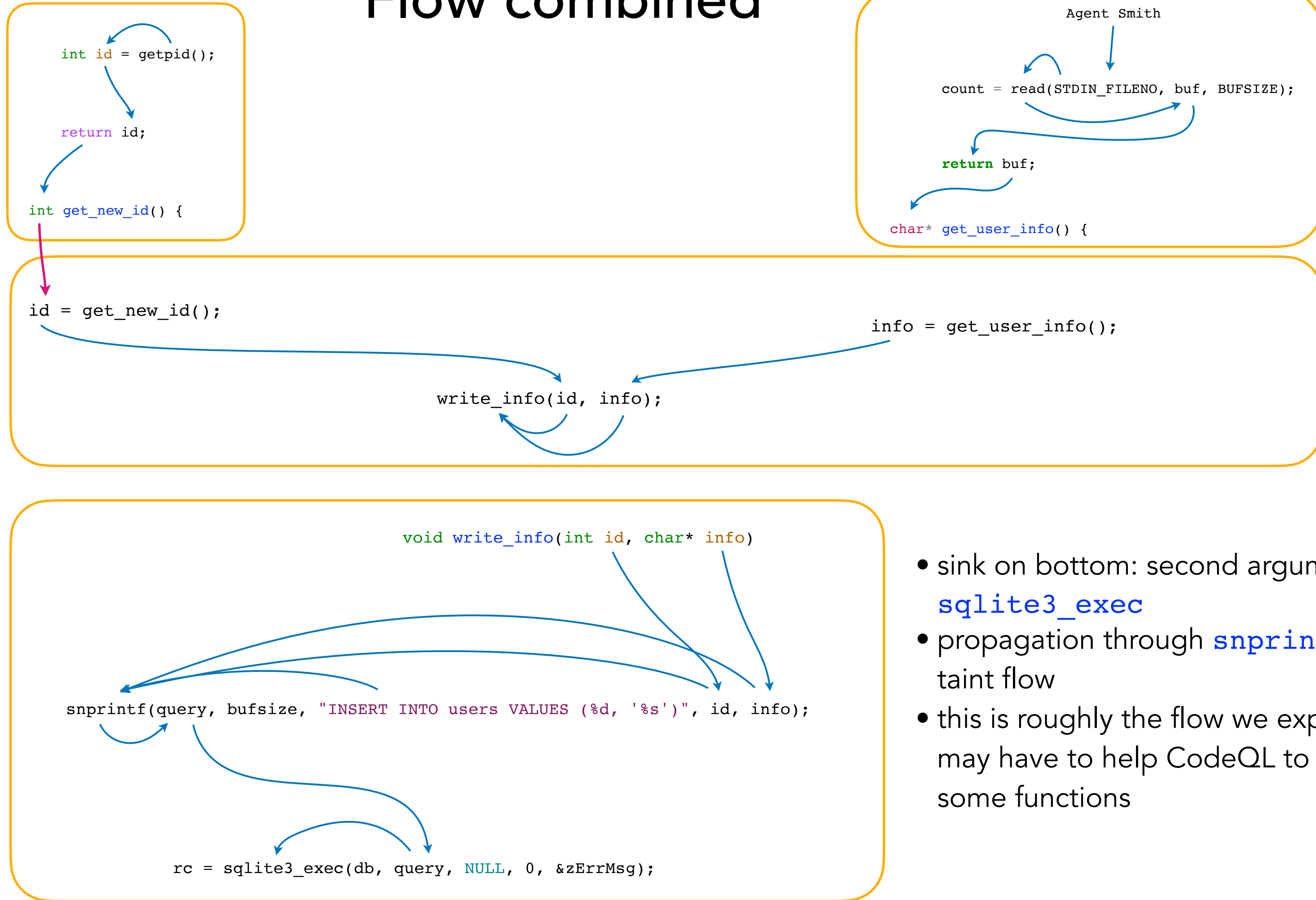
# Flow combined



- 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

- inter-procedural (global) data flow

# Flow combined

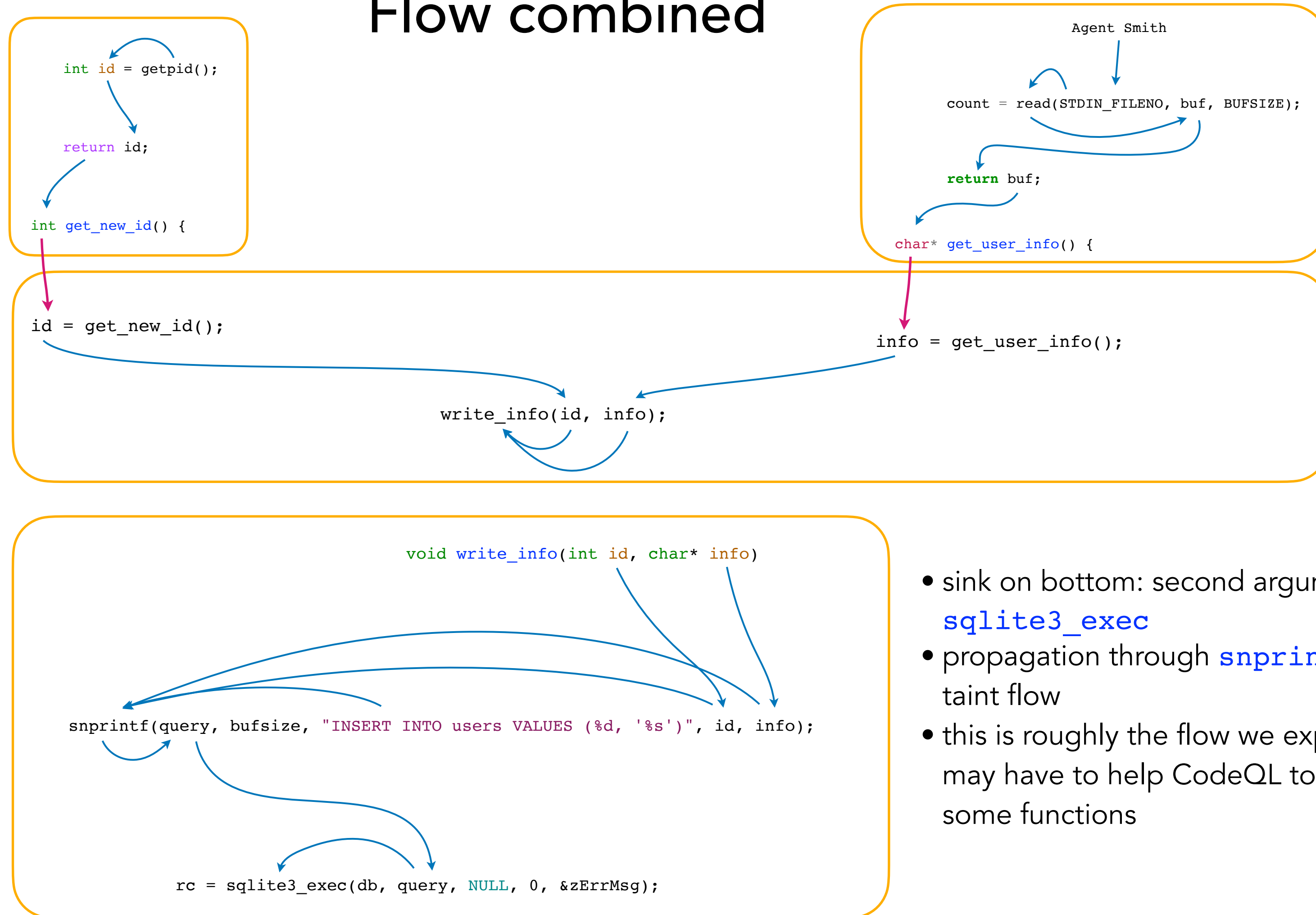


- sink on bottom: second argument to `sqlite3_exec`
- propagation through `snprintf` needs taint flow
- this is roughly the flow we expect to see; may have to help CodeQL to capture flow across some functions



- inter-procedural (global) data flow

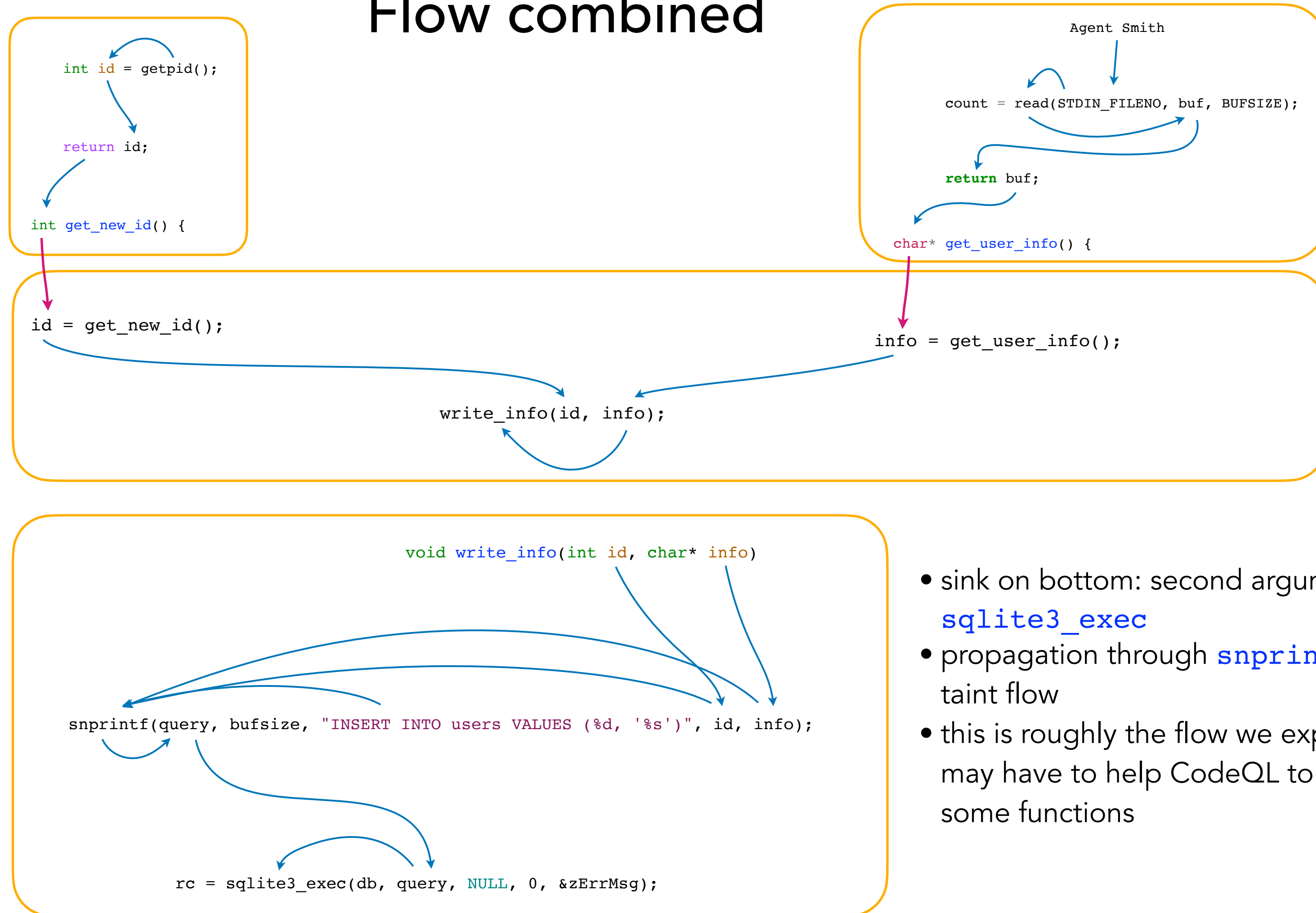
# Flow combined



- 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

- inter-procedural (global) data flow

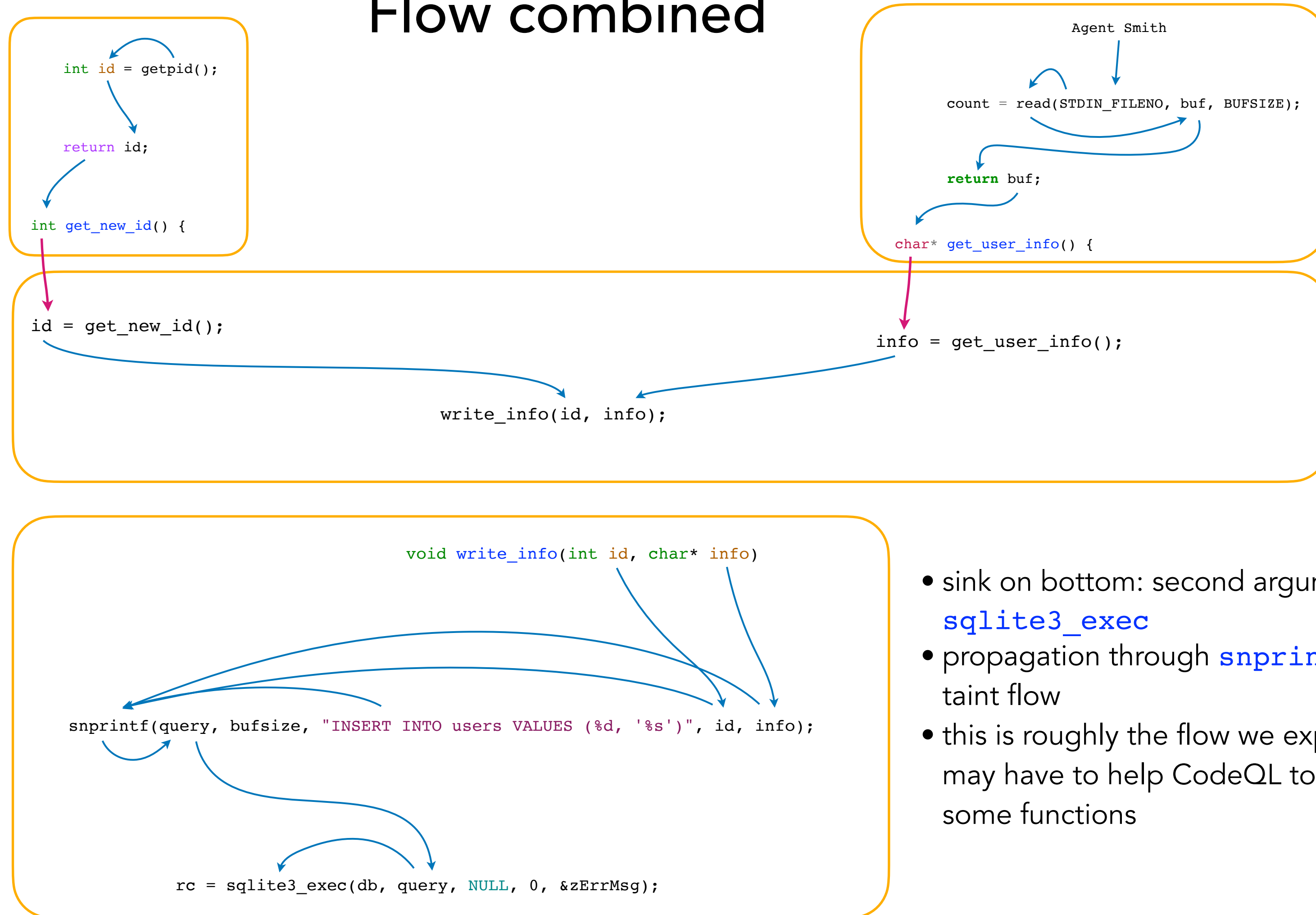
# Flow combined



- 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

- inter-procedural (global) data flow

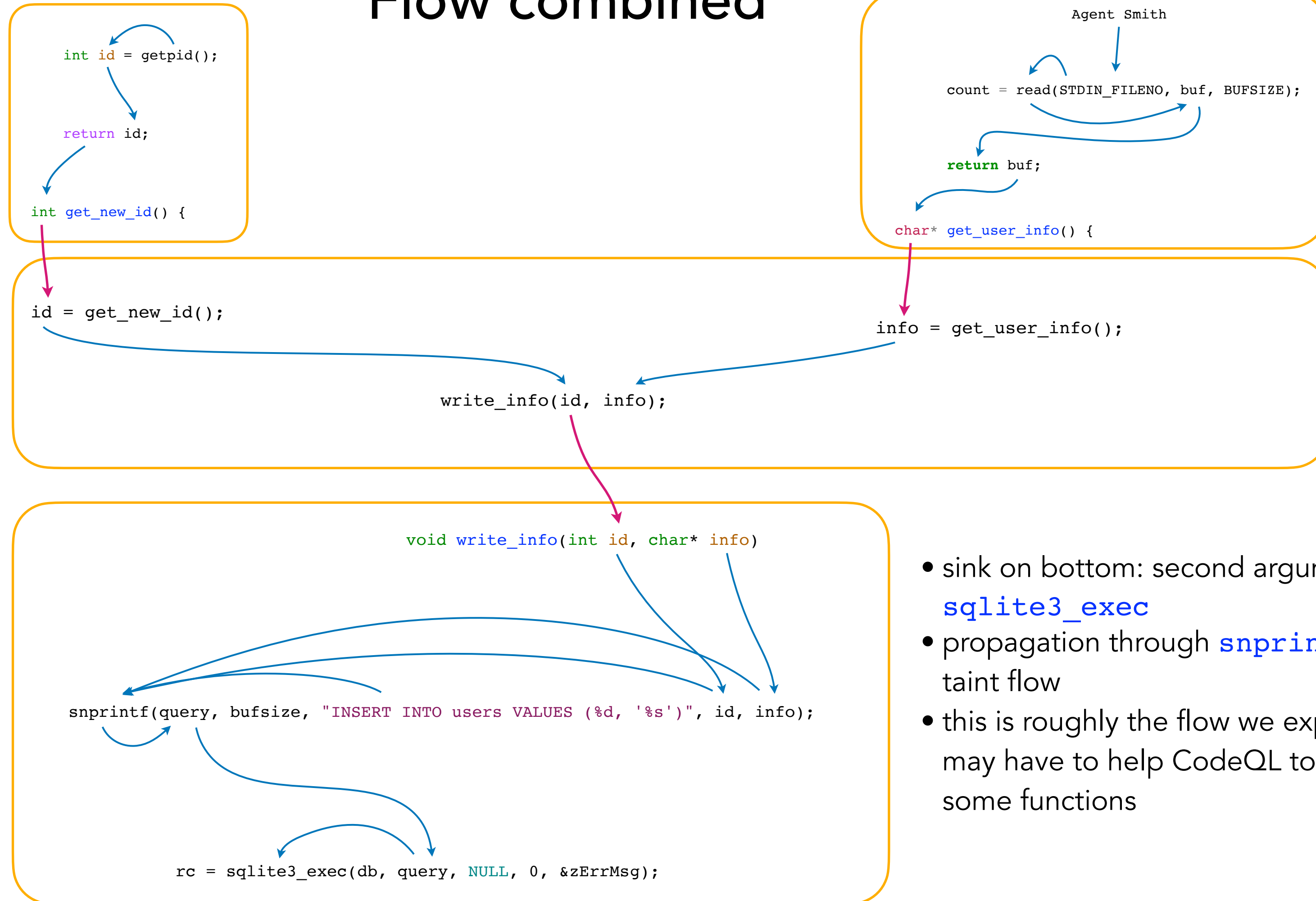
# Flow combined



- 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

- inter-procedural (global) data flow

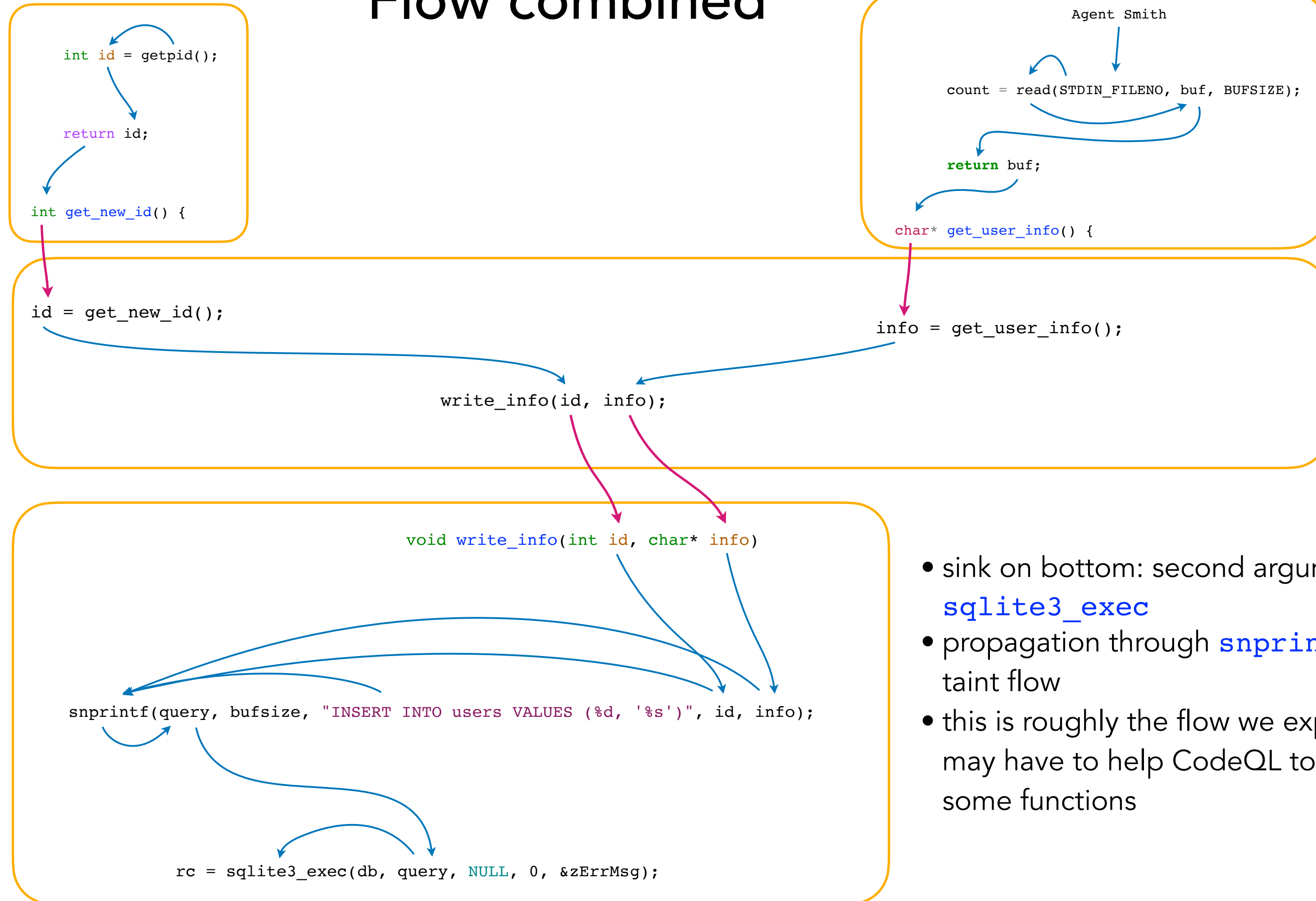
# Flow combined



- 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

- inter-procedural (global) data flow

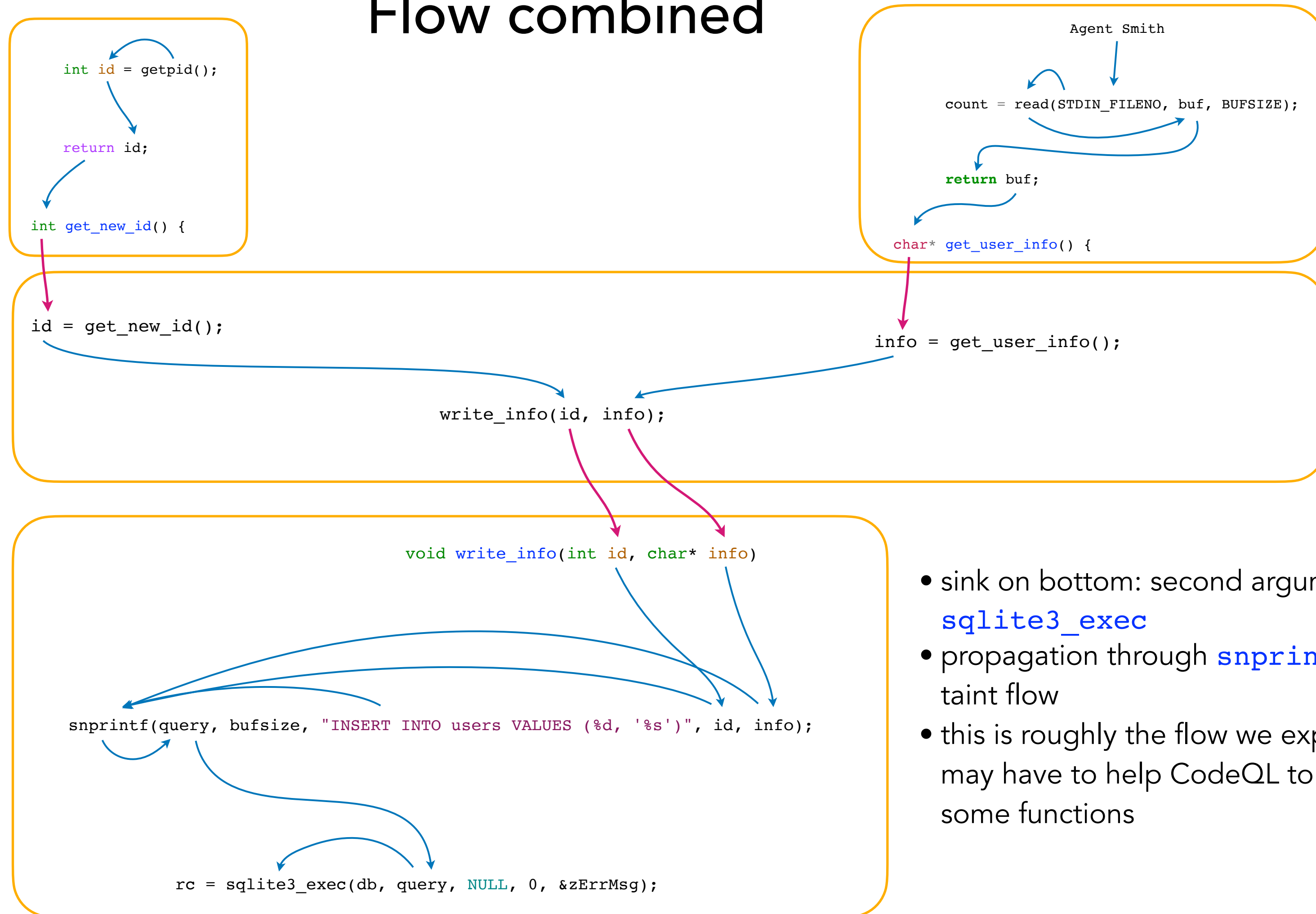
# Flow combined



- 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

# Flow combined

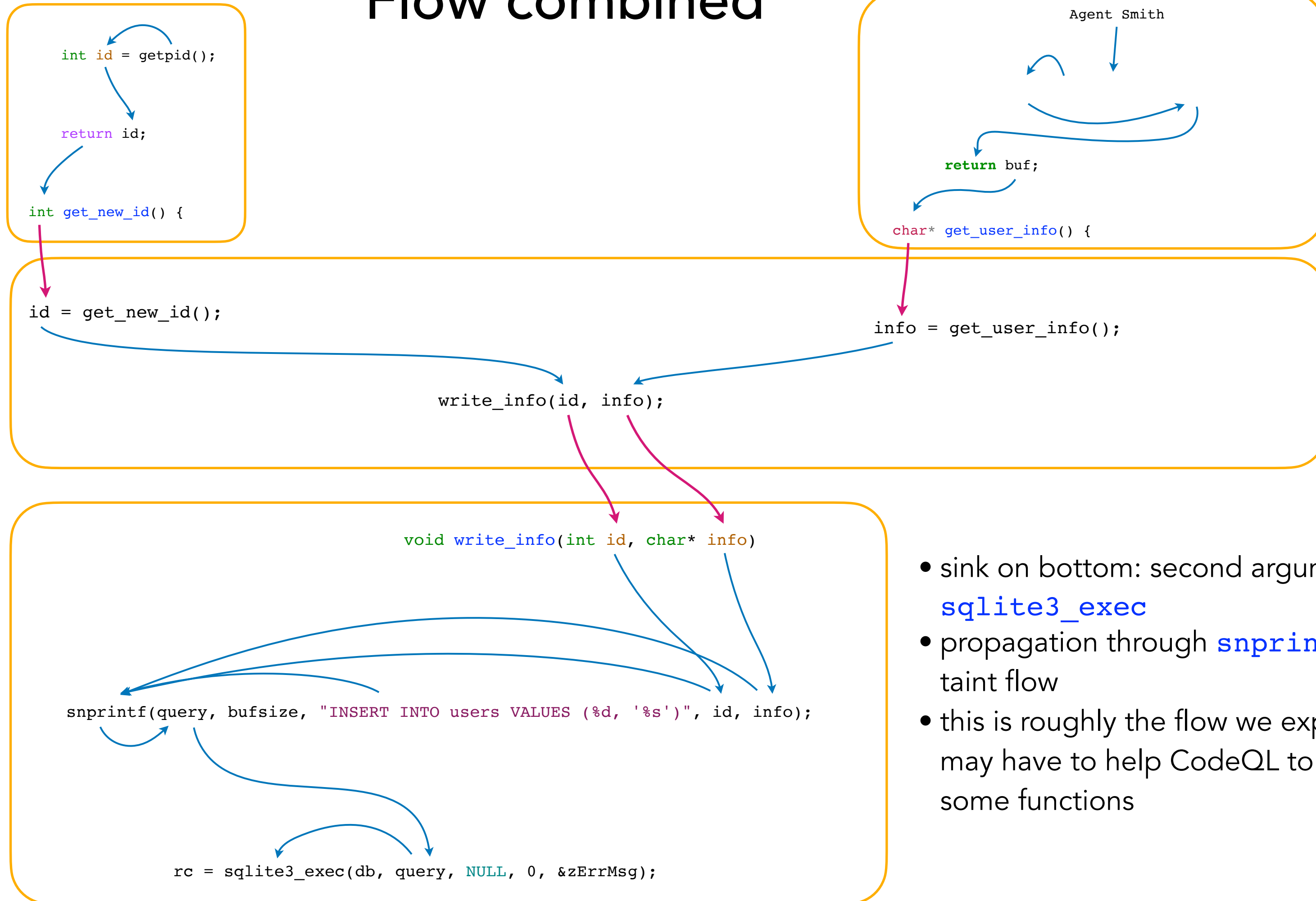
- inter-procedural (global) data flow
- source on top: second argument to `read`



- 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

# Flow combined

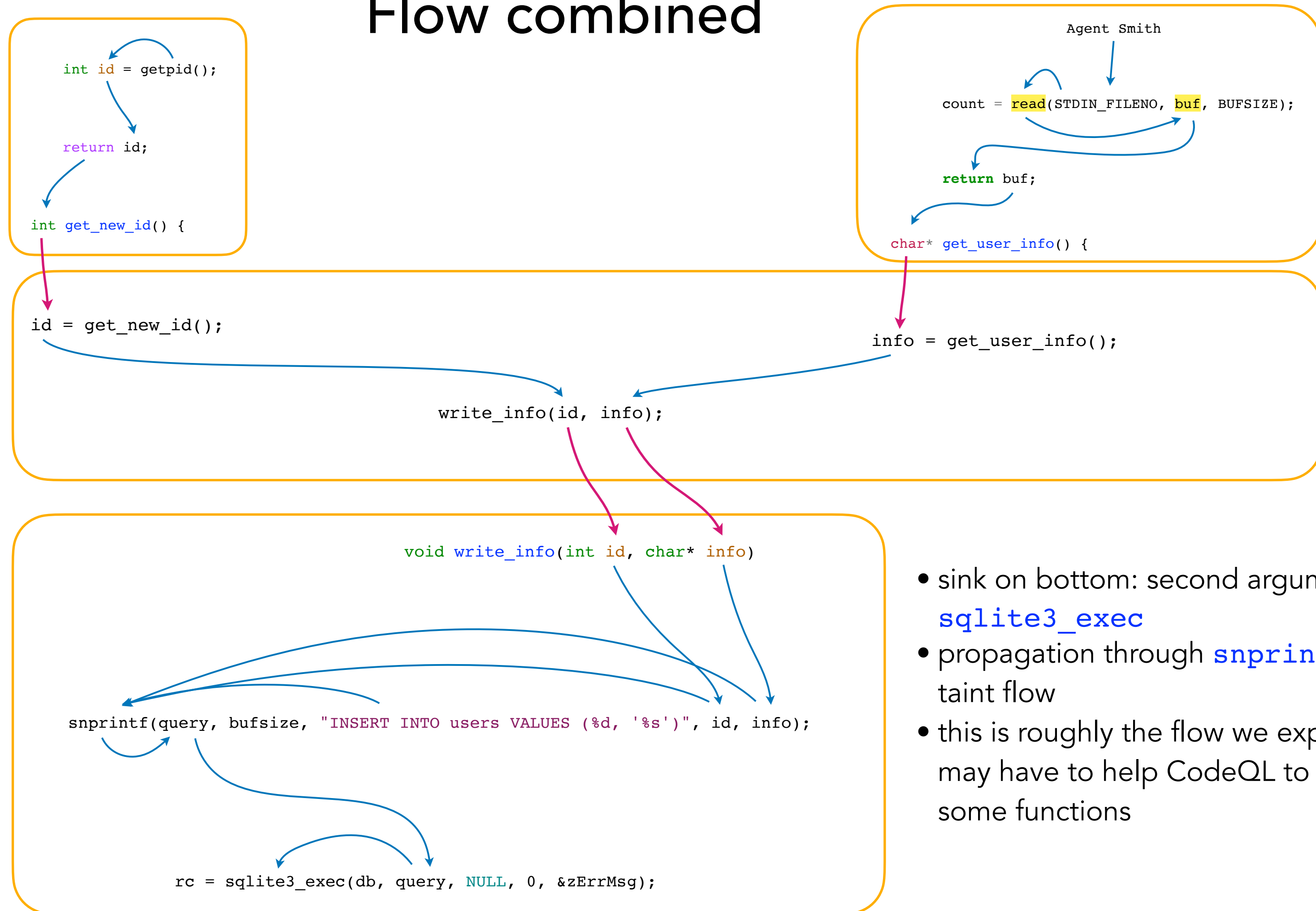
- inter-procedural (global) data flow
- source on top: second argument to `read`



- 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

# 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