

# 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;  
}
```

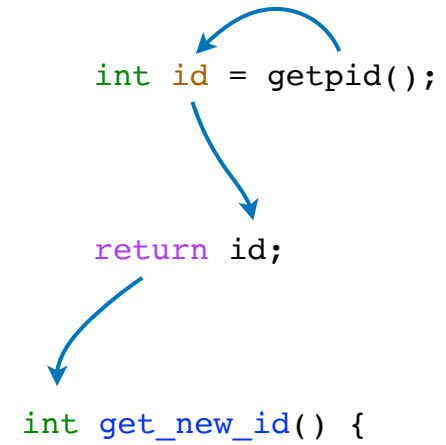
```
graph TD; A["int id = getpid();"] --> B["return id;"]
```

The diagram illustrates the flow of execution between two lines of code. A blue 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;`. A second blue arrow starts from the end of the assignment statement and points back to the start of the return statement, indicating the sequence of execution.

```
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;
}
```

Agent Smith

```
graph TD
    AS[Agent Smith] --> R[count = read(STDIN_FILENO, buf, BUFSIZE);]
    R --> Ret[return buf;]
```

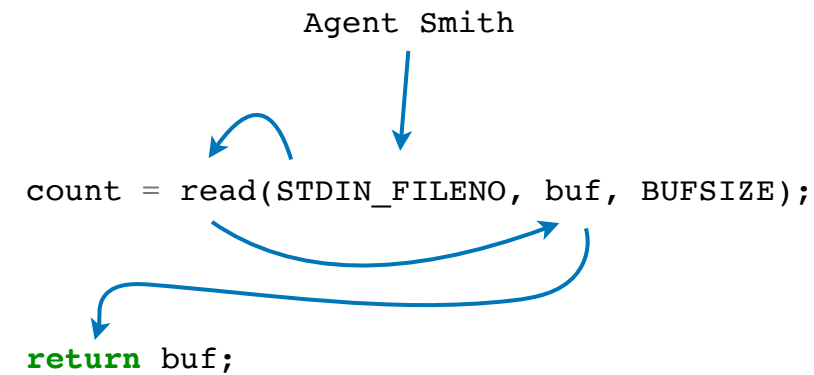
The diagram illustrates the flow of data in the `get_user_info` function. An arrow labeled "Agent Smith" points to the `read` function call in the line `count = read(STDIN_FILENO, buf, BUFSIZE);`. A curved arrow points from the `count` variable in this line to the `return buf;` statement, indicating the return path.

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



# Flow in get\_user\_info

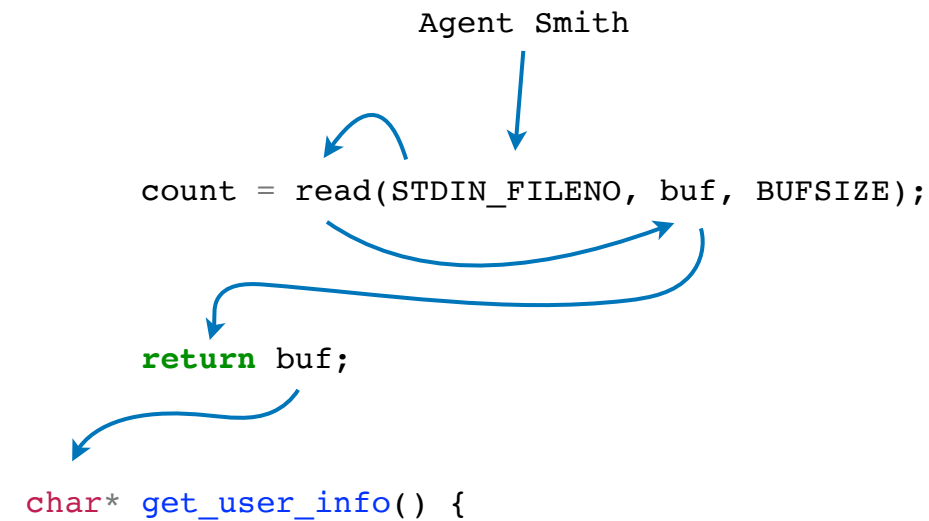
```
char* get_user_info() {  
#define BUFSIZE 1024  
    char* buf = (char*) malloc(BUFSIZE * sizeof(char));  
    int count;  
    // Disable buffering to avoid need for fflush  
    // after printf().  
    setbuf( stdout, NULL );  
    printf("*** Welcome to sql injection ***\n");  
    printf("Please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

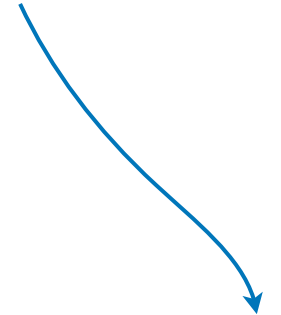
    /* open db */
    rc = sqlite3_open("users.sqlite", &db);
    abort_on_error(rc, db);

    /* 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`

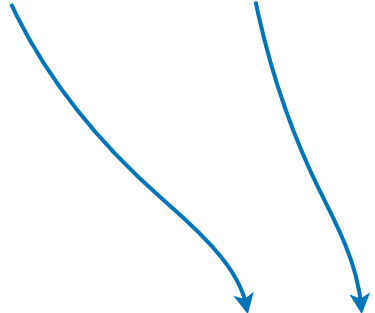
```
void write_info(int id, char* info) {
    sqlite3 *db;
    int rc;
    int bufsize = 1024;
    char *zErrMsg = 0;
    char query[bufsize];

    /* 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);
}
```

```
graph TD
    subgraph Signature
        direction LR
        S1["void write_info(int id, char* info)"]
    end
    subgraph QueryConstruction
        direction LR
        Q1["snprintf(query, bufsize, \"INSERT INTO users VALUES (%d, '%s')\", id, info);"]
    end
    subgraph Execution
        direction LR
        E1["rc = sqlite3_exec(db, query, NULL, 0, &zErrMsg);"]
    end
    S1 --> Q1
    S1 --> E1
    Q1 --> E1
```

The diagram illustrates the flow of data in the `write_info` function. It shows the function signature `void write_info(int id, char* info)` at the top. Two blue arrows originate from the parameters `id` and `info` and point to their respective arguments in the `snprintf` call: `id` points to `(%d, '%s')` and `info` points to `id, info`. A third blue arrow points from the `snprintf` call to the `query` argument in the `sqlite3_exec` call, indicating that the formatted query string is passed to the database execution function.

```
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)
{
    char query[100];
    sprintf(query, "INSERT INTO users VALUES (%d, '%s')", id, info);
    sqlite3_exec(db, query, NULL, 0, &zErrMsg);
    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);
```

The `snprintf` call then flows to the `sqlite3_exec` 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);
}
```

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 `snprintf` call. Arrows indicate the flow of data:

- From `id` to the first format specifier `%d` in the `snprintf` call.
- From `info` to the second format specifier `'%s'` in the `snprintf` call.
- From the `snprintf` call to the `sqlite3_exec` call.

The `snprintf` call is shown below the function signature:

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

The `sqlite3_exec` call is shown below the `snprintf` call:

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

# Flow in `write_info`

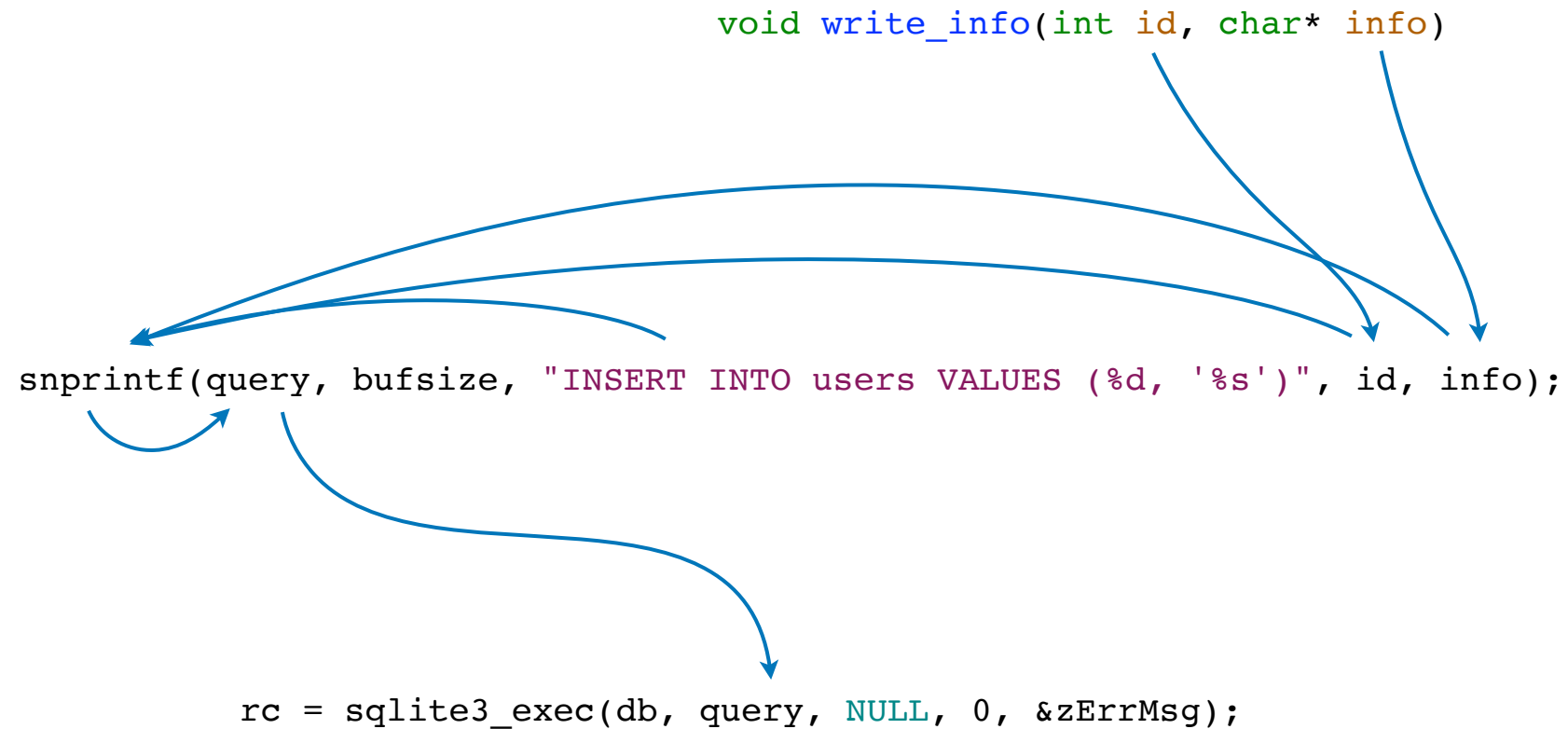
```
void write_info(int id, char* info) {
    sqlite3 *db;
    int rc;
    int bufsize = 1024;
    char *zErrMsg = 0;
    char query[bufsize];

    /* 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`

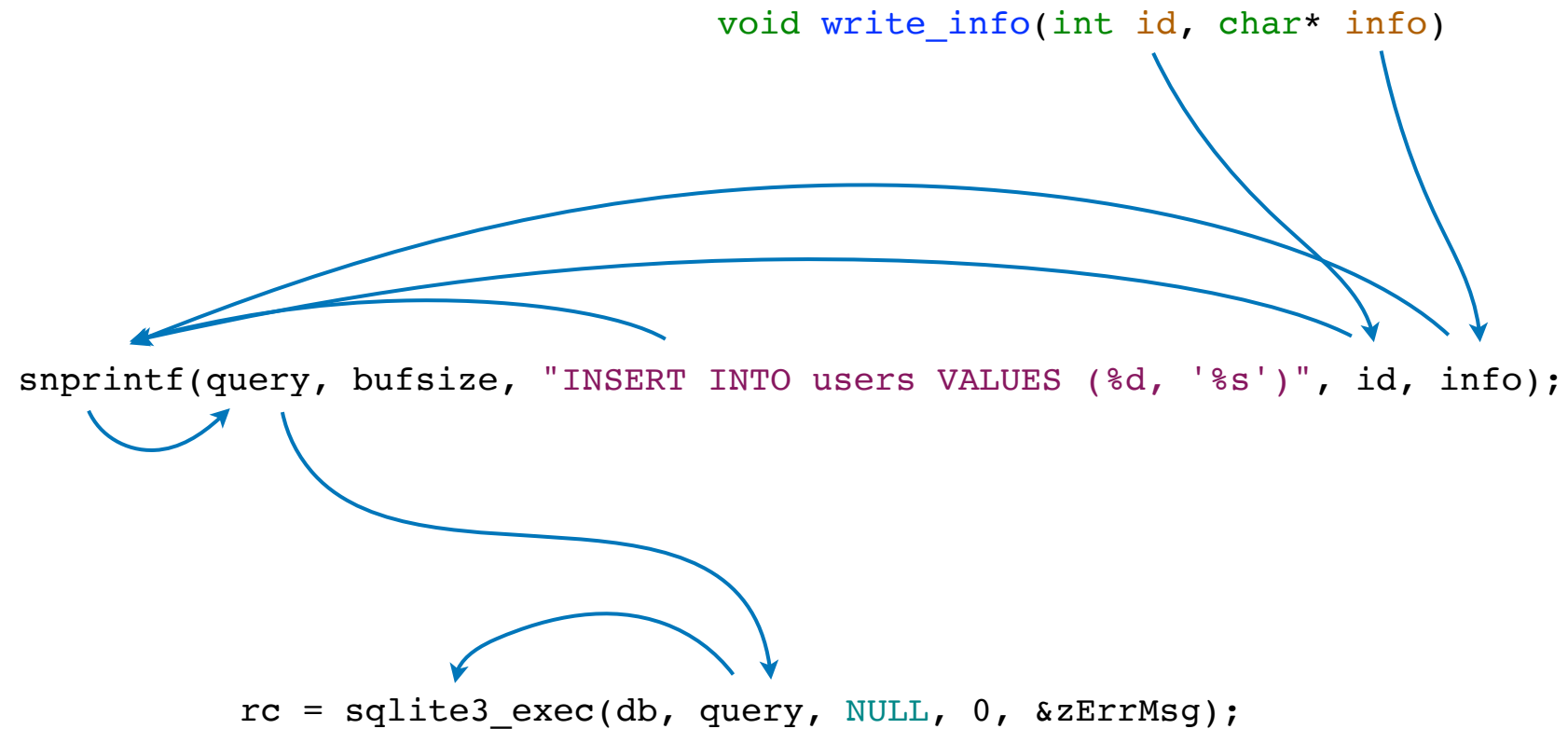
```
void write_info(int id, char* info) {
    sqlite3 *db;
    int rc;
    int bufsize = 1024;
    char *zErrMsg = 0;
    char query[bufsize];

    /* 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();`

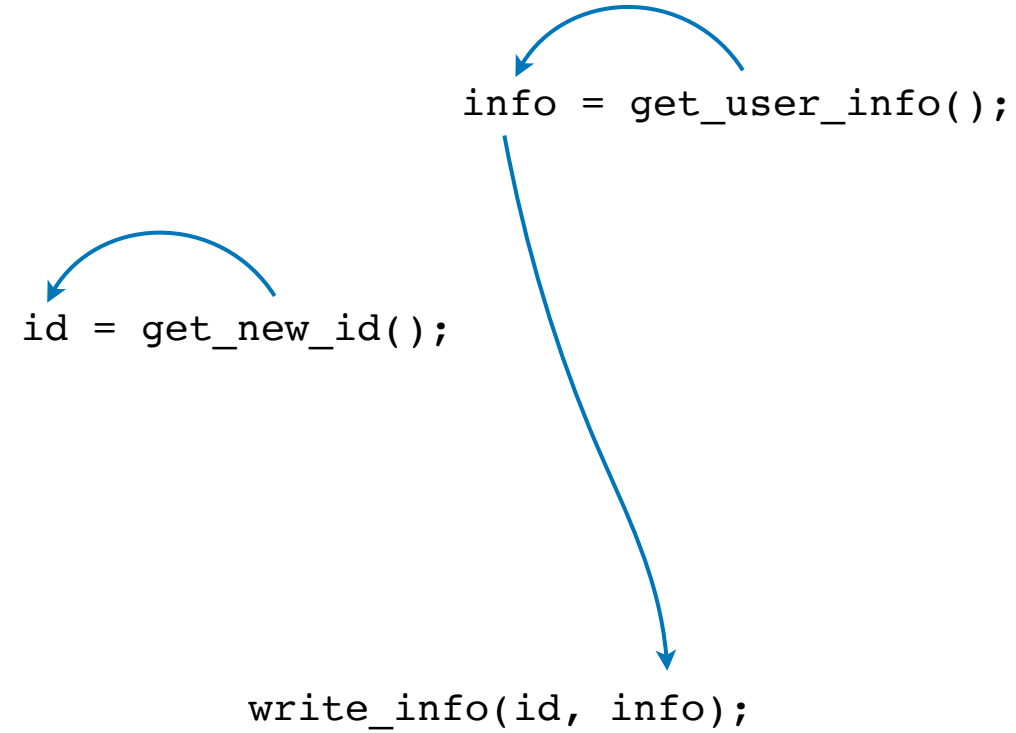


`id = get_new_id();`

`write_info(id, info);`

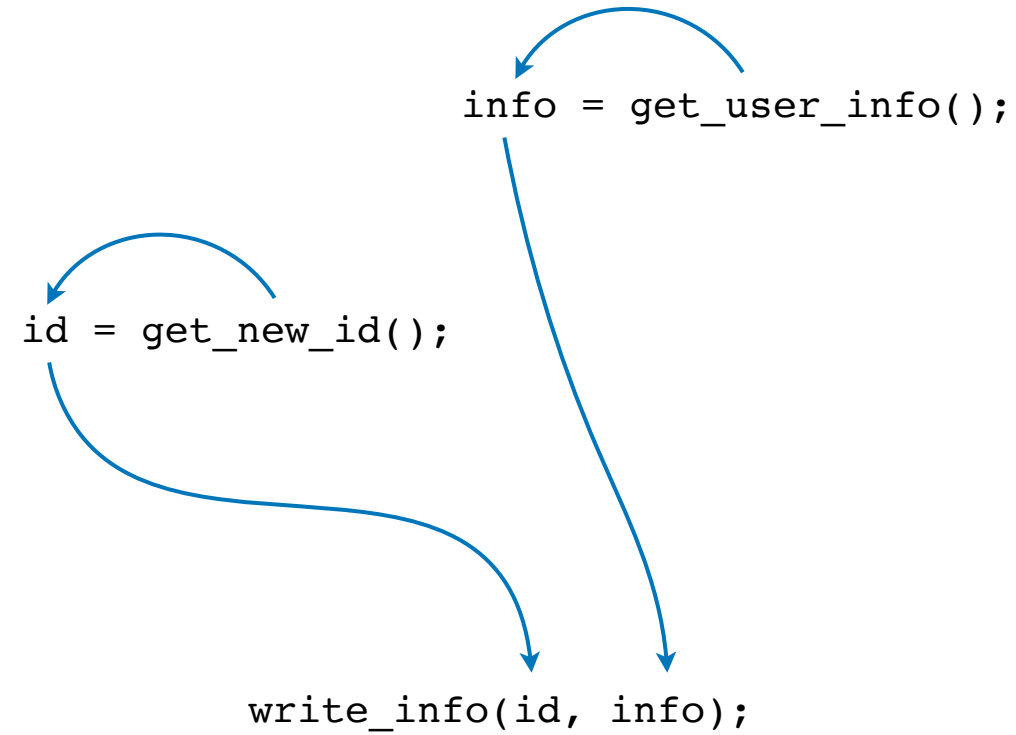
# Flow in `main`

```
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`

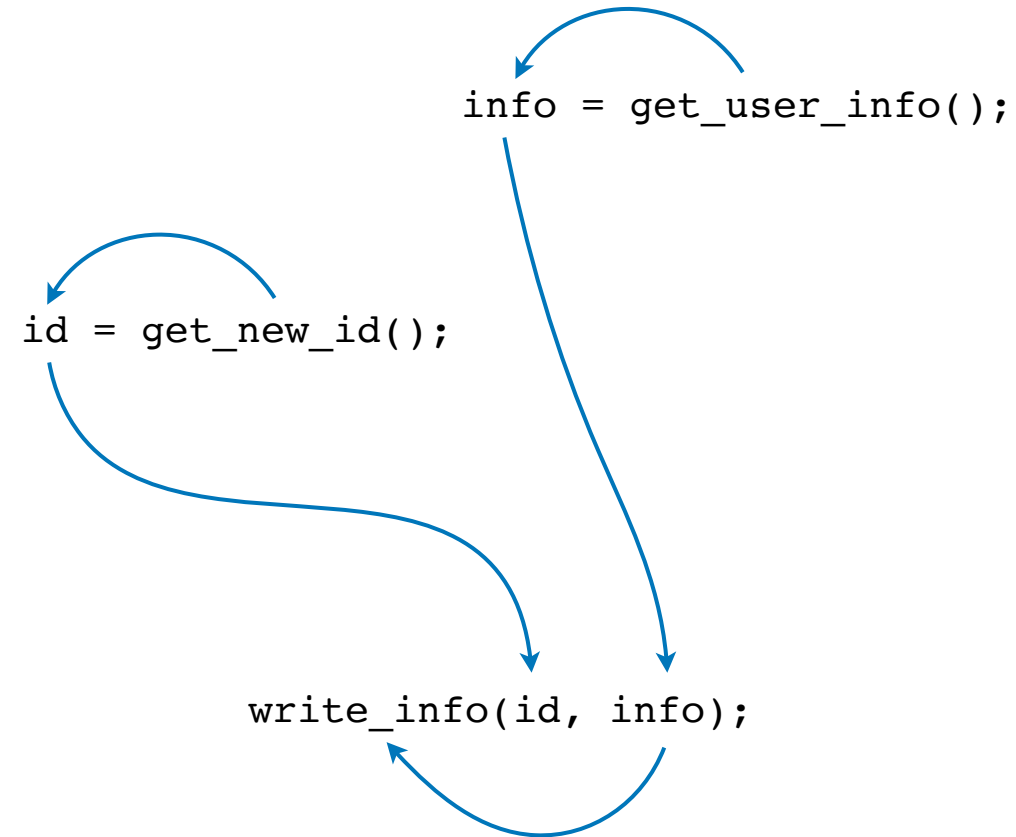
```
int main(int argc, char* argv[]) {  
    char* info;  
    int id;  
    info = get_user_info();  
    id = get_new_id();  
    write_info(id, info);  
}
```





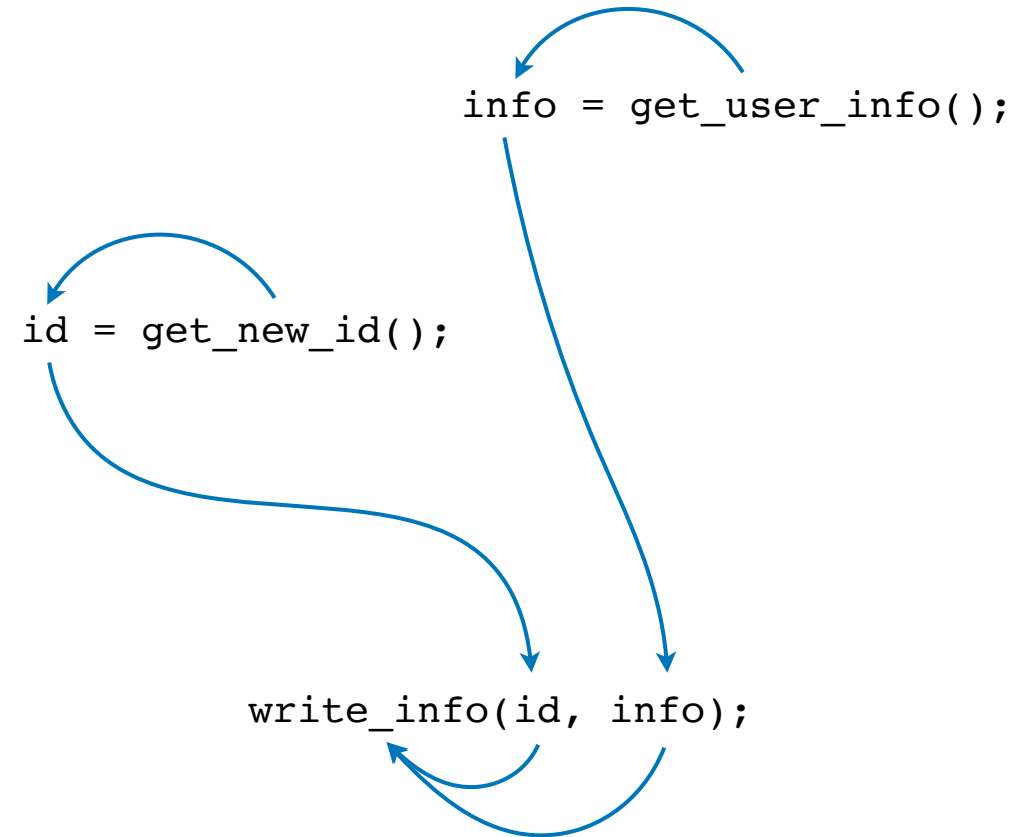
# Flow in `main`

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



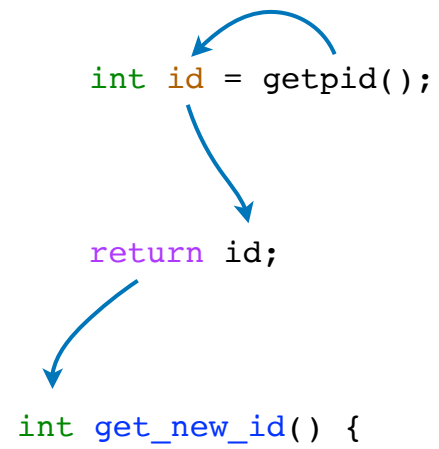
# Flow in `main`

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



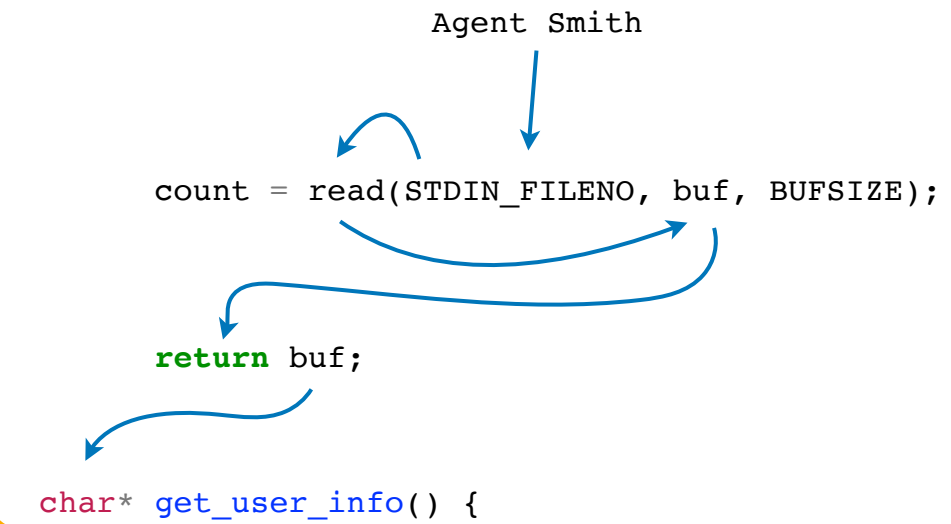
# Flow combined

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



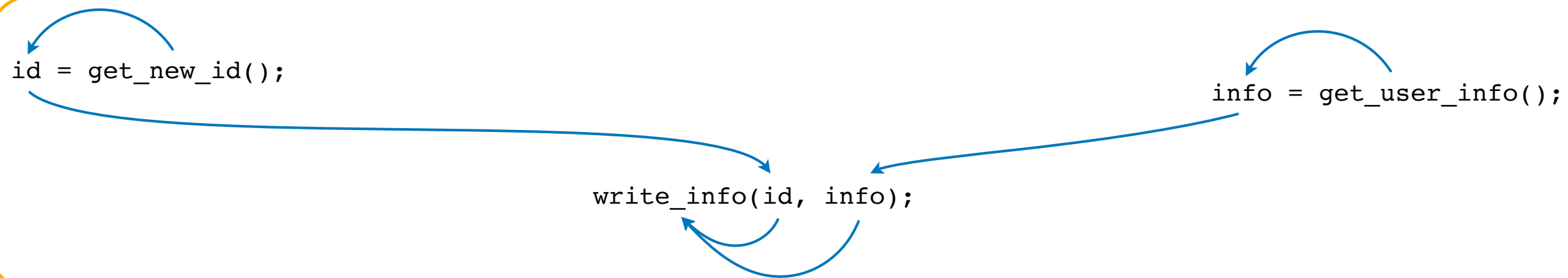
Flow diagram for the `get_new_id()` function. It shows a self-loop on the `int id = getpid();` line, an arrow pointing down to `return id;`, and another arrow pointing down to the start of the function definition `int get_new_id() {`.

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



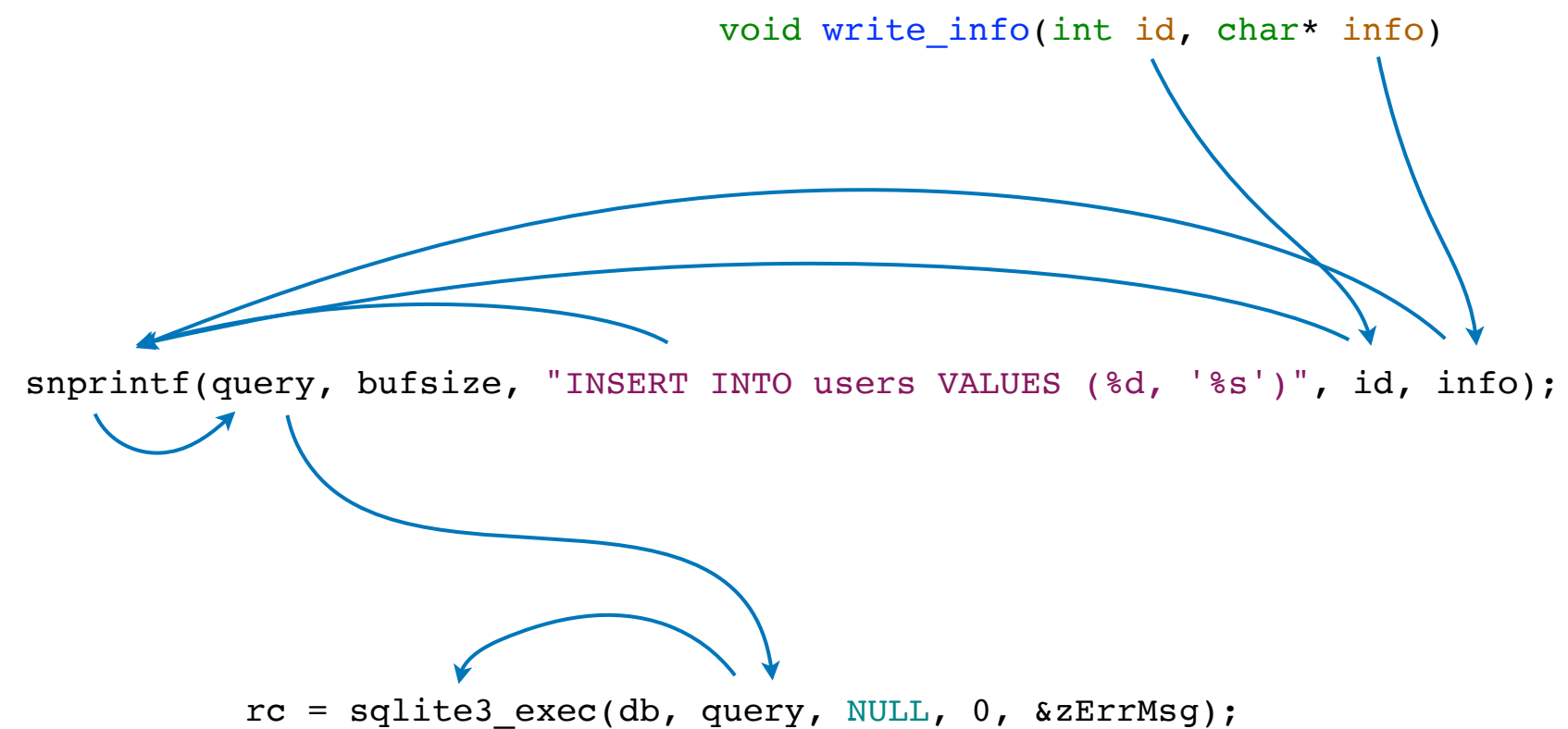
Flow diagram for the `get_user_info()` function. It starts with an arrow from "Agent Smith" to the `count = read(STDIN_FILENO, buf, BUFSIZE);` line. There is a self-loop on this line, an arrow pointing down to `return buf;`, and another arrow pointing down to the start of the function definition `char*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

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



Flow diagram for the `write_info(id, info);` call. It shows an arrow from `id = get_new_id();` and another from `info = get_user_info();` both pointing to the `write_info(id, info);` line. 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 `id` and `info` in the function signature point to their respective arguments in the `snprintf` call. Arrows from `query` and `bufsize` in the `snprintf` call point to the `sqlite3_exec` call. There is also a self-loop on the `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() {
```

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

```
id = get_new_id();
```

```
info = get_user_info();
```

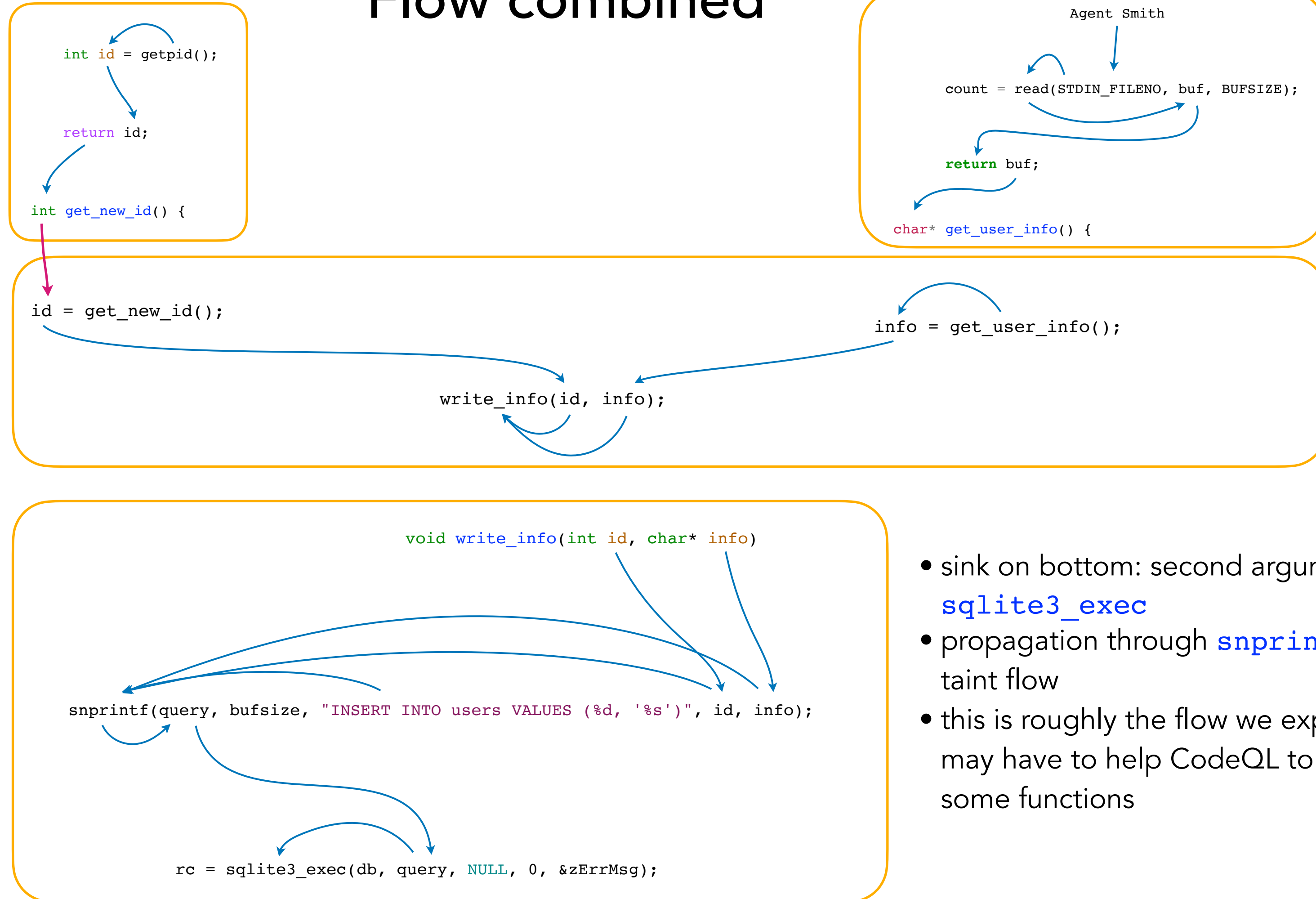
```
write_info(id, info);
```

```
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

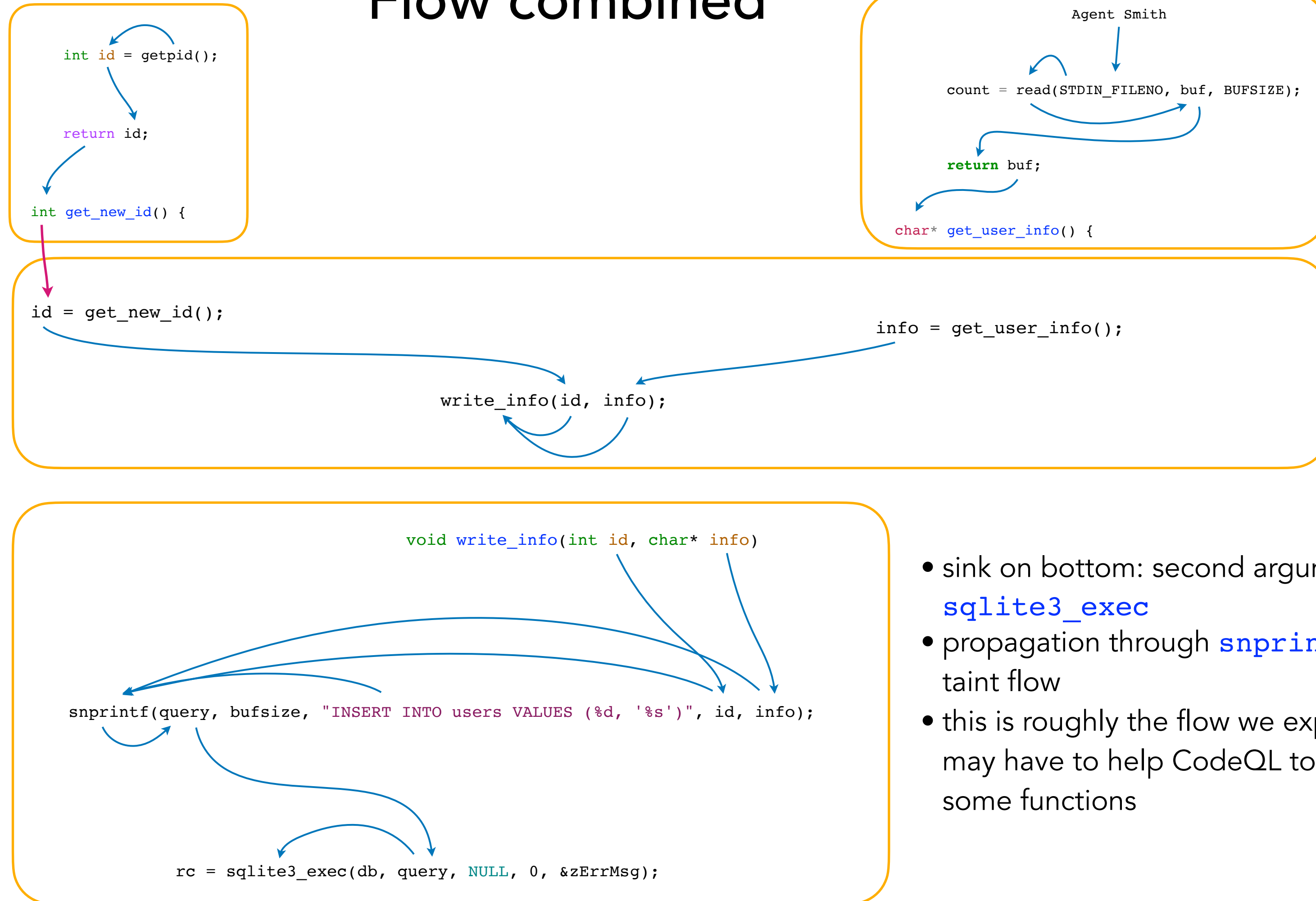
# Flow combined



- 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

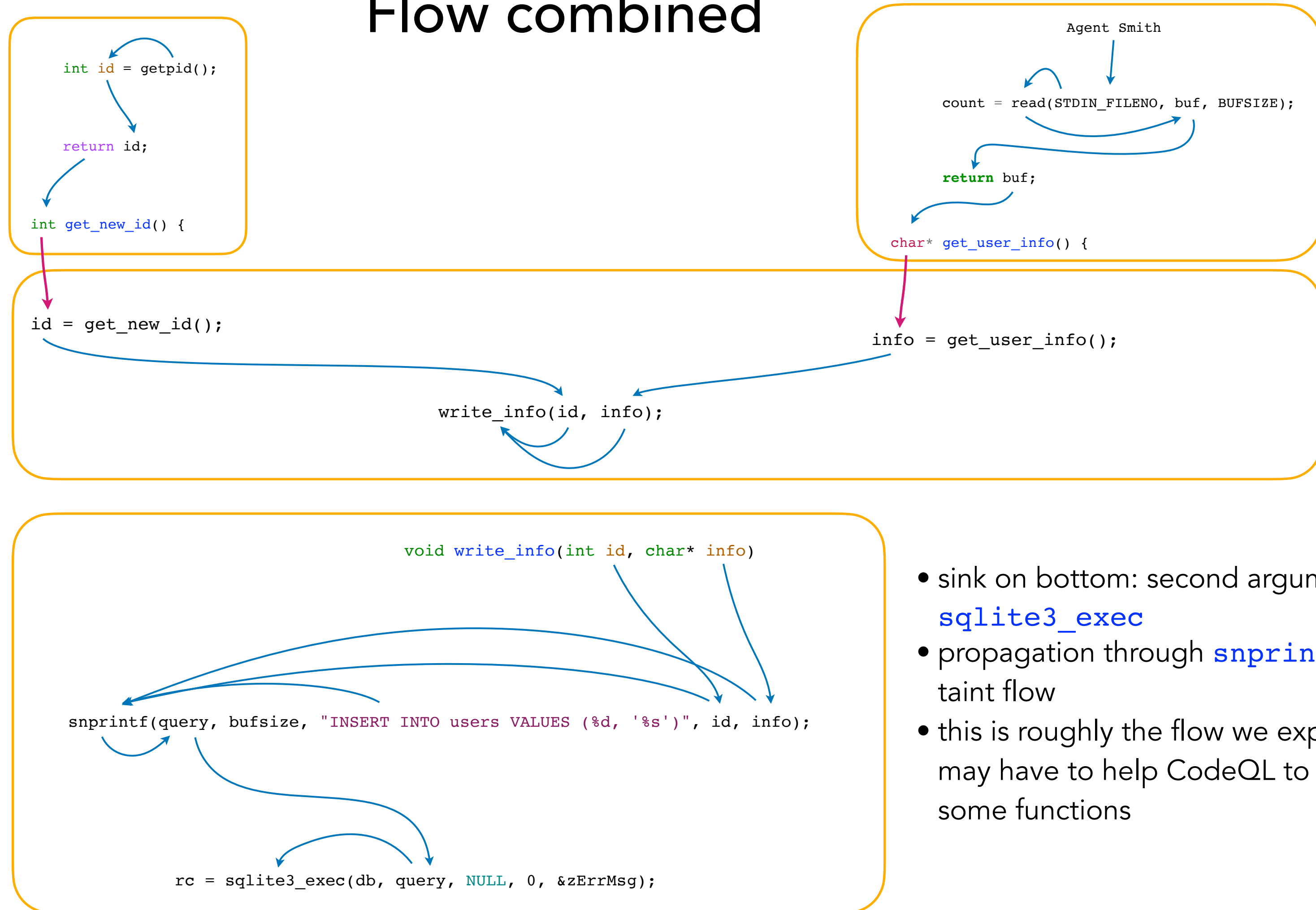
# Flow combined



- 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

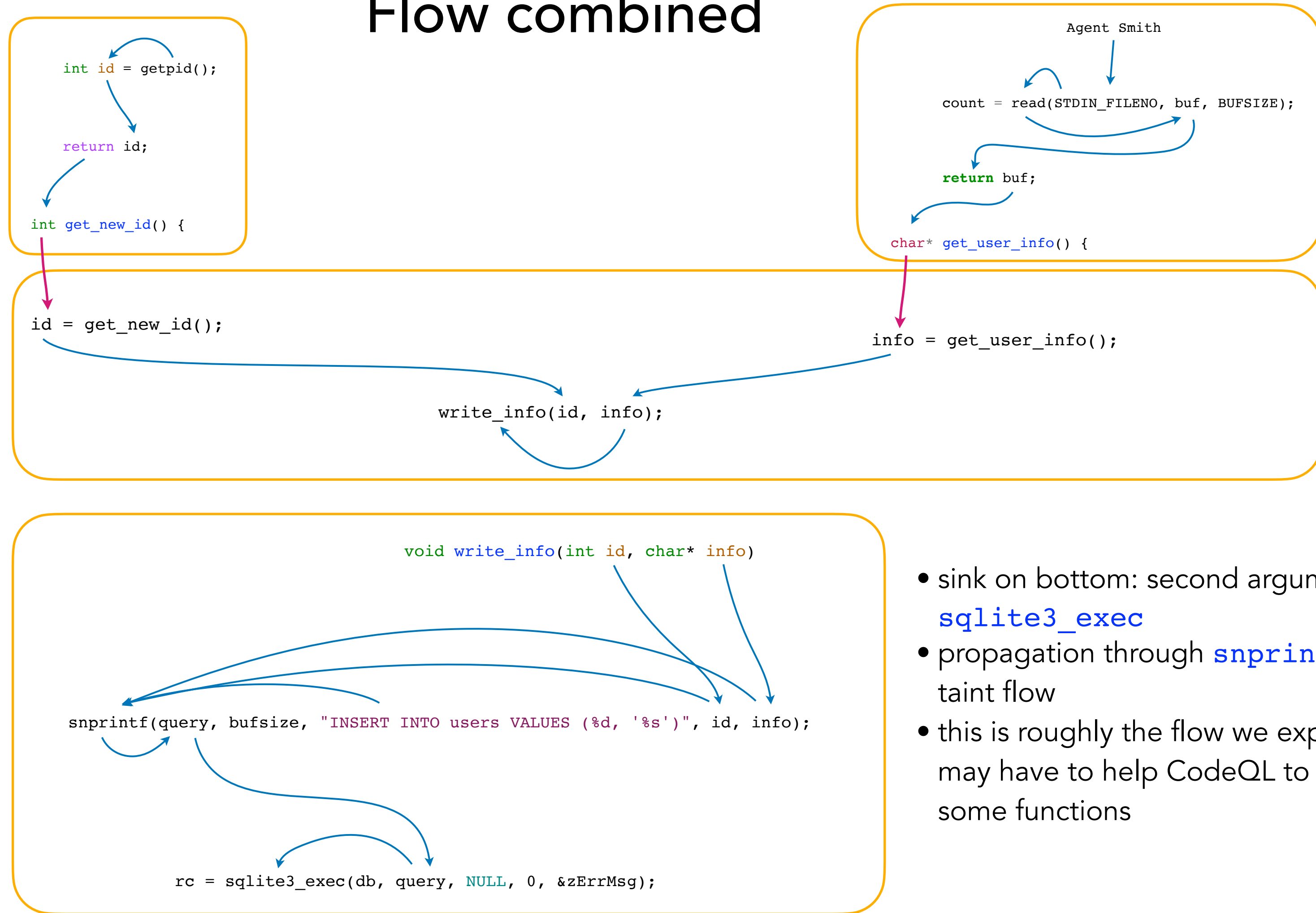


- sink on bottom: second argument to `sqlite3_exec`
- propagation through `snprintf` needs taint flow
- this is roughly the flow we expect to see; may have to help CodeQL to capture flow across some functions



- inter-procedural (global) data flow

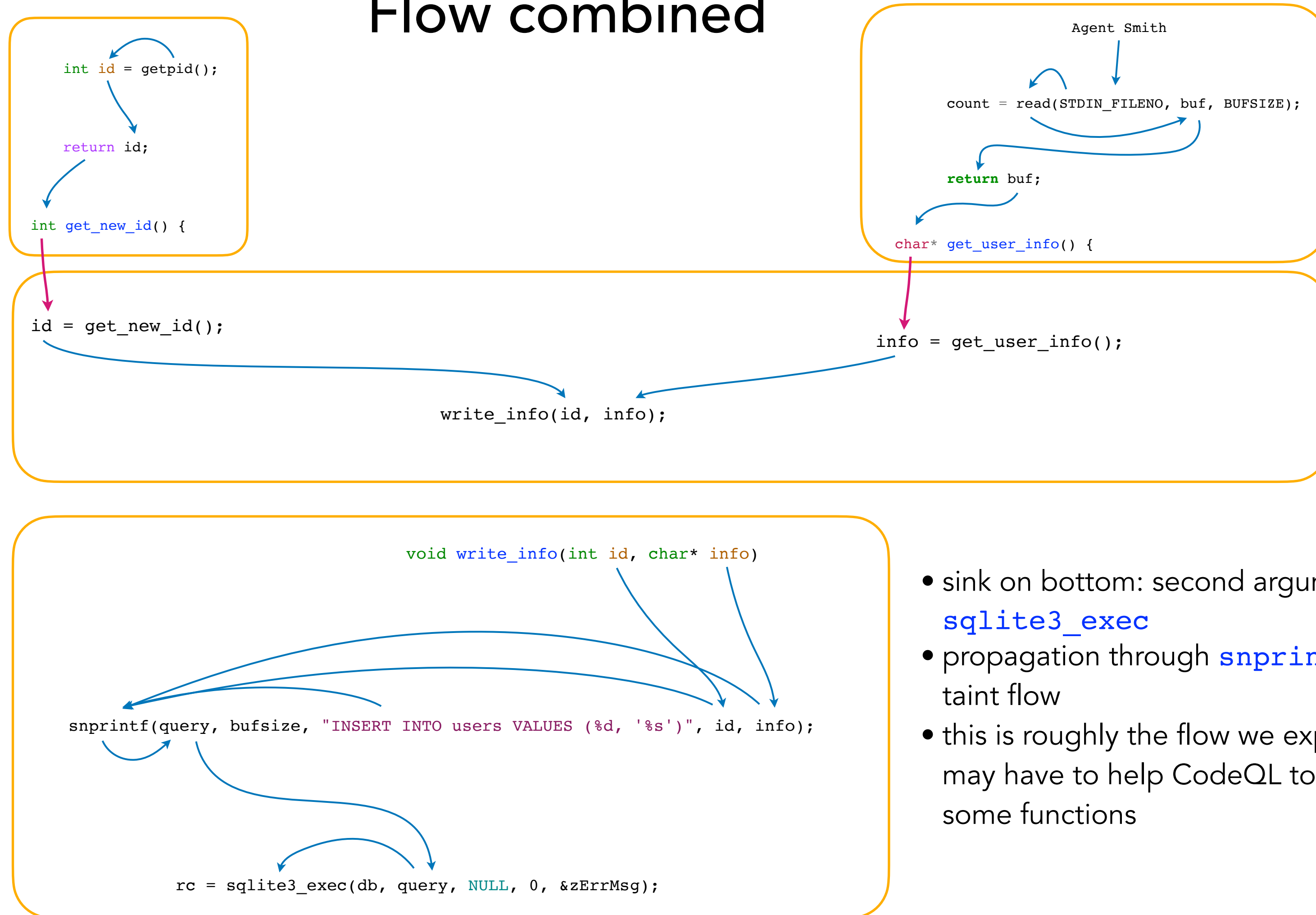
# Flow combined



- 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

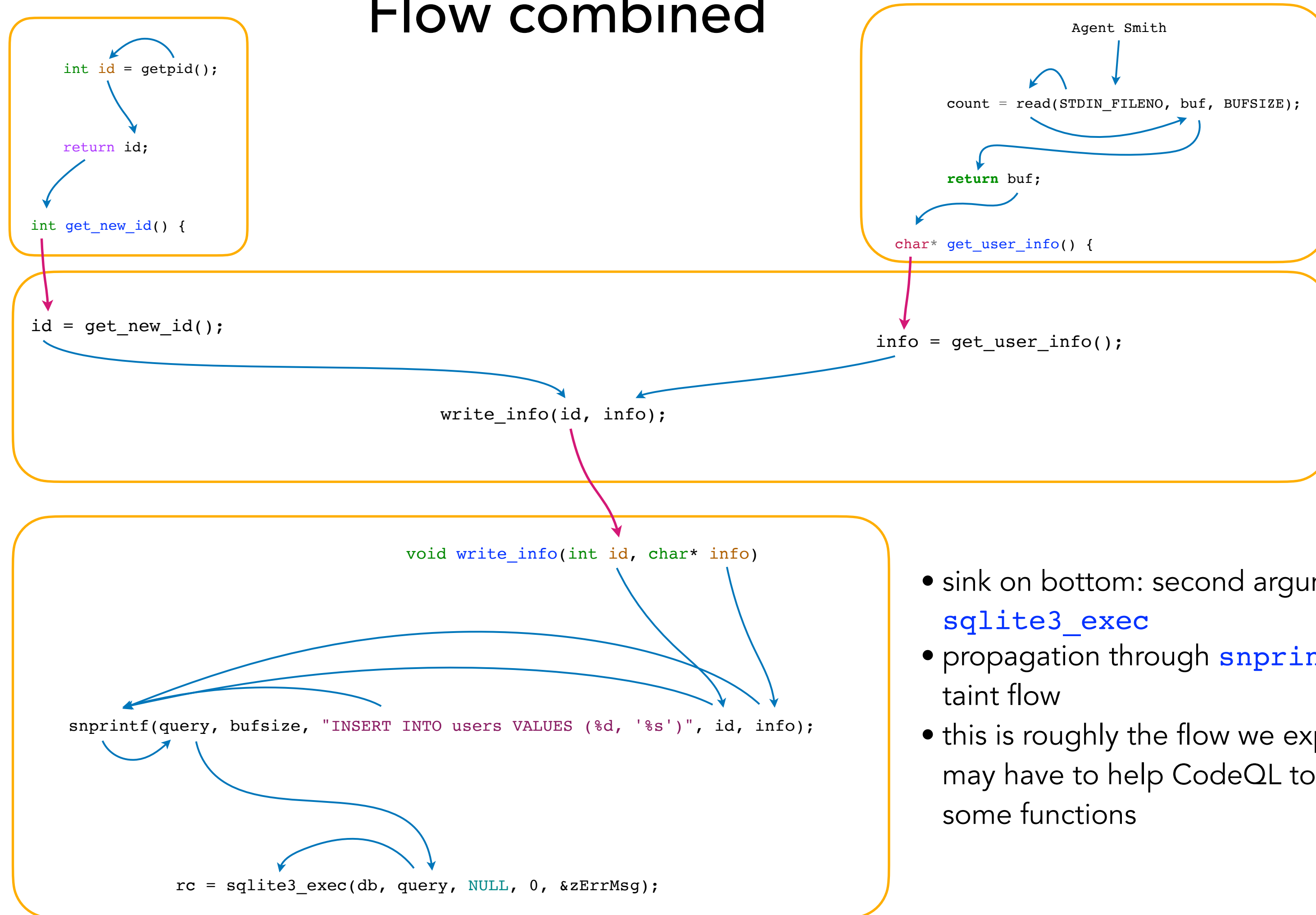
# Flow combined



- 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

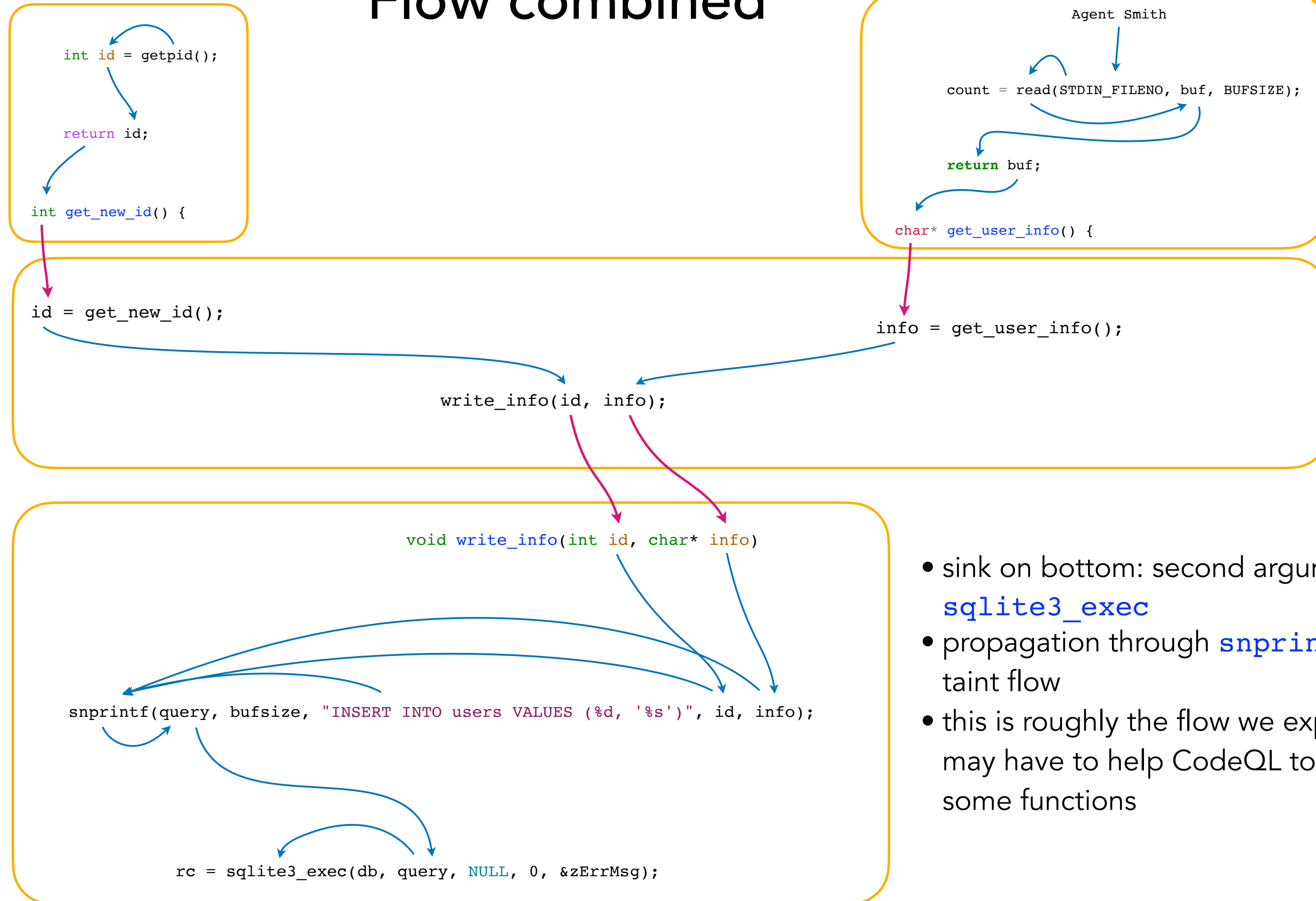
# Flow combined



- 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

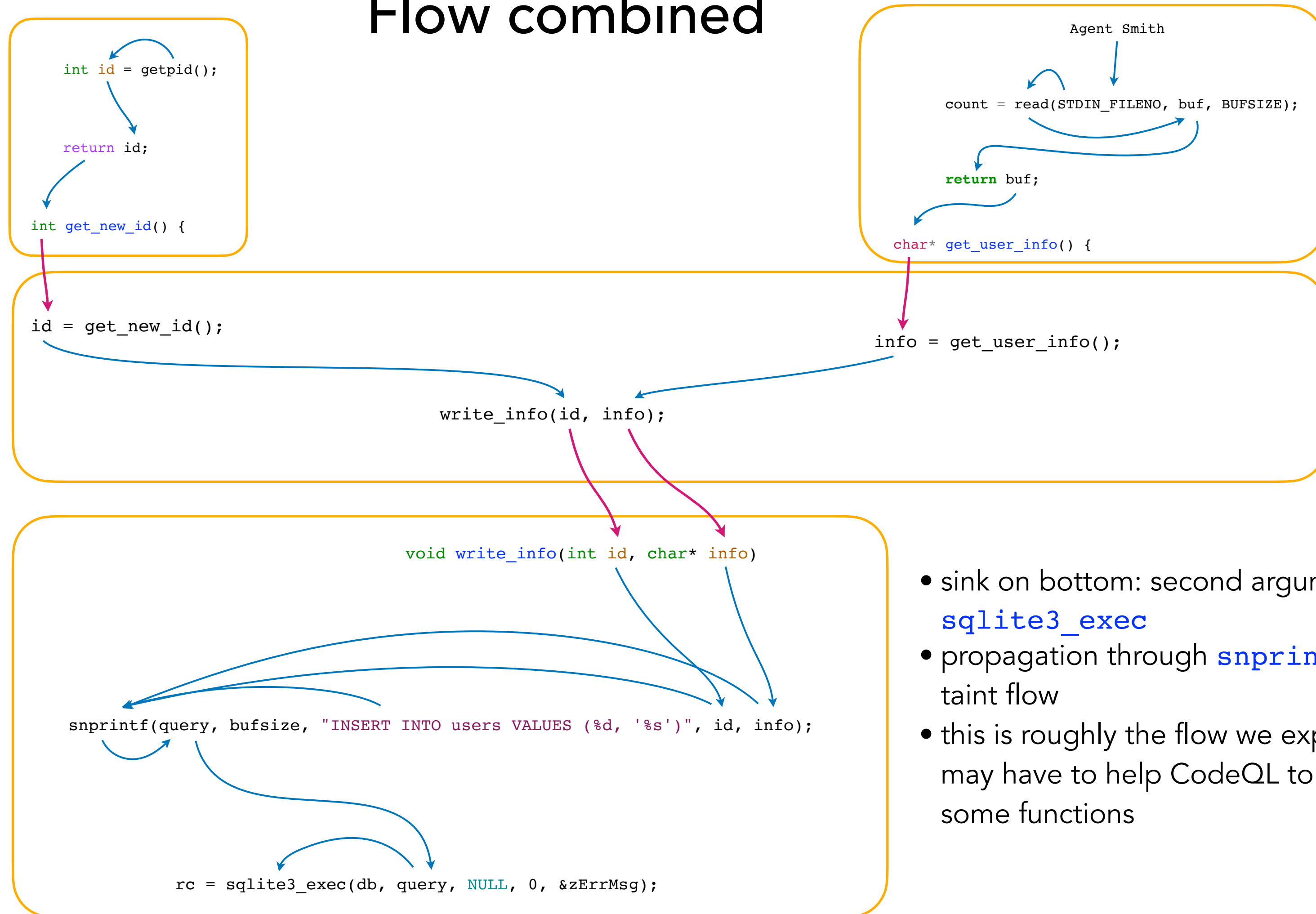
# Flow combined



- 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

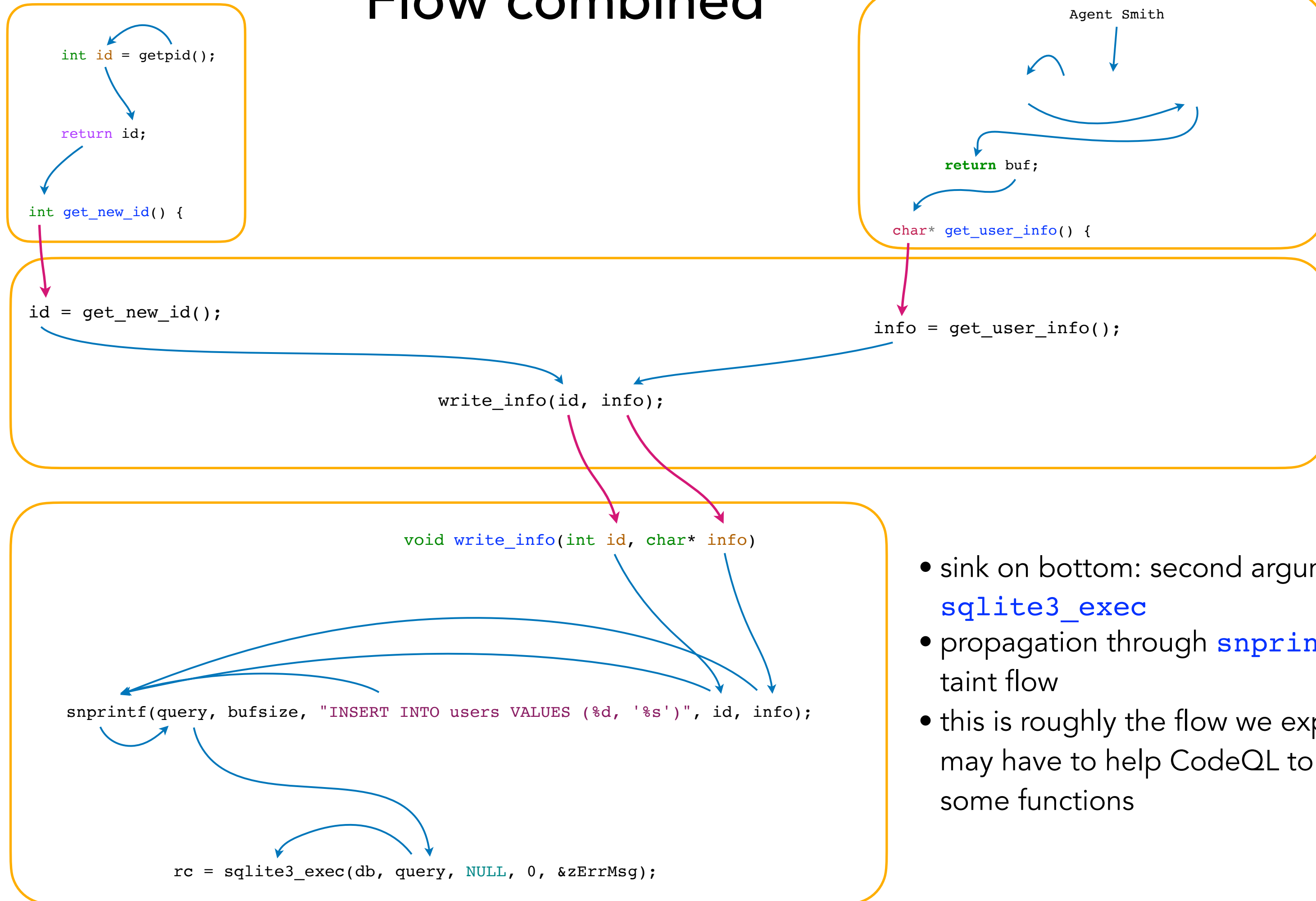
- inter-procedural (global) data flow
- source on top: second argument to `read`



- 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

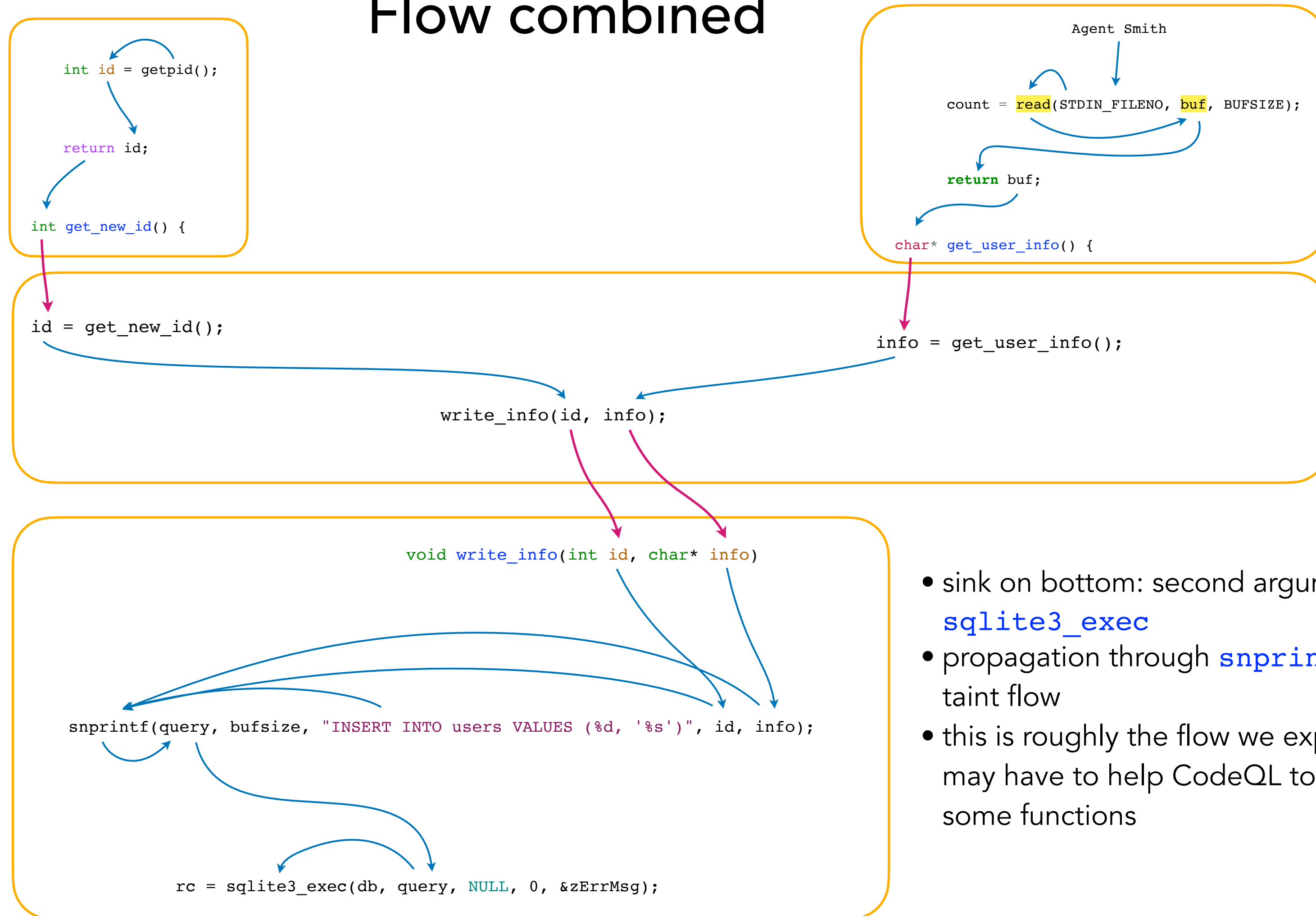
- inter-procedural (global) data flow
- source on top: second argument to `read`



- 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