# CodeQL を使ってみよう

## まえがき

こんにちは！いわんこ(X: @iwancof_weakptr)と申します。普段は、天井を眺めるか pwn をしています。

wordian なのか wordian でないのか自分でも確定していなかったのですが、意味不明なので記事を書いて確定させたいと思います。

## CodeQL とは？

CodeQL とは、GitHub が開発した**プログラミング言語に対する**クエリ言語です。ここではまず導入として簡単な例を提示し、その解説を行います。 どんなツールかが雰囲気だけでも伝わると嬉しいです。

ちゃんとした解説を次以降のセクションで行いますので、導入を面白いと思ってくださる方がいらっしゃいましたら、ぜひご一読ください！

さて、さっそく次の C 言語のコードを、後述するクエリで処理してみます。

```c
#include <stdio.h>

int main(void) {
    printf("it works!");
}
```

```
import cpp

from FunctionCall fc
where fc.getTarget().getName() = "printf"
select fc, fc.getEnclosingFunction(), fc.getLocation()
```

すると、このような結果を得ることができます。

```
Compiling query plan for /home/iwancof/WorkSpace/CodeQL/article/queries/printf.ql.
[1/1 comp 8.9s] Compiled /home/iwancof/WorkSpace/CodeQL/article/queries/printf.ql.
Starting evaluation of qlsample/article/queries/printf.ql.
Evaluation completed (243ms).
|       fc       | col1 |                            col2                             |
+----------------+------+-------------------------------------------------------------+
| call to printf | main | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:4:5:4:10 |
Shutting down query evaluator.
```

クエリの内容を解説しつつ、結果を考察してみましょう。

`from` や `where`、`select` など、クエリ言語らしいキーワードが散見されます。ざっくりとした解説になりますが...

```
from FunctionCall fc
```

まず、`from` 句で変数の定義を行います。ここでは、`FunctionCall` という型を持つ `fc` という実体を定義したと考えてください。`FunctionCall` とは、名前の通り**関数呼び出し**を表しており、`fc` は関数呼び出しという**事象**が入ると思ってください。

```
where fc.getTarget().getName() = "printf"
```

次に、where 句で fc をフィルタしています。型にはメソッドが存在して、getTarget や getName がそれに該当します。雰囲気的に、「関数呼び出しの内、そのターゲット（呼び出される関数）の名前が"printf"のもの」をフィルタしていそうです。

```
select fc, fc.getEnclosingFunction(), fc.getLocation()
```

最後に、select 句を用いてフィルタした fc や fc に付随する情報を表示します。getEnclosingFunction はその関数呼び出しを行っている関数を返し、getLocation は関数呼び出しが行われた場所を返します。

結果を再掲します。

```
|       fc       | col1 |                                col2                                 |
+----------------+------+---------------------------------------------------------------------+
| call to printf | main | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:4:5:4:10 |
```

３つのカラムが返ってきていて、それぞれ fc, col1, col2 という名前がついています。fc は"printf"という関数呼び出し実体が入っています。

col1 は fc.getEnclosingFunction()の結果が入っていて、実際に printf を呼んでいる main 関数の実体を取得しています。

col2 は fc.getLocation()の結果で、その呼び出しが行われている場所を一意に特定する文字列が返ってきます。この場合、printf.c という名前でテストしていたので、そのファイル名と行番号などを得ることができています。

この例では、CodeQL を使ってコード自体をデータベースとして解析し、関数呼び出しの実体とそれを含む関数やその場所を取得しました。 CodeQL ではより高度なクエリを書くことで、更に多くの情報をソースコードから得ることができます。また、C 言語だけでなく、数多くの言語を扱うことができ、それぞれ専用のライブラリが存在しています。詳しくは公式のドキュメントを参照してください。

次のセクションから、環境構築、実行方法、クエリの基礎、クエリの応用、原理を紹介します。 もちろん公式の包括的なドキュメントも存在し、いずれはそちらを参考にするときが来ると思います。しかし、ドキュメントの読解に際し、CodeQL への最低限の理解が要求され、そのための日本語の情報が乏しいということでこの記事を執筆しています。ただ、筆者の CodeQL への理解が甘いことに起因し、記事を読んでいるだけでは理解できない部分も多々あると思います。 実際に手を動かし自分で検証を行ったり、もしくは X や Discord(@Iwancof_ptr) で質問していただいても結構です！連絡お待ちしています。

また、記事の作成には x86_64 Linux を使用しています。 それ以外の環境に関しては詳しくありませんが、MSVC や C# .NET にも対応していることから、多くの環境で使えると思います。

## 環境構築

CodeQL はパッケージマネージャで配布されていないため、GitHub からバイナリを取ってきます。

https://github.com/github/codeql/tags

ここから、適当に latest を取ってきて展開し、展開したディレクトリに PATH を通します。codeql というコマンドが動かせたら準備完了です。

VSCode に CodeQL 用の拡張があり、結果を GUI で扱いたい方は併せて導入をおすすめします。 LSP もありますので、Vim を使っている方も安心してください。

## 実行方法

では実際にクエリを書き、実行してみましょう。

クエリの実行には大きく次のステップを踏む必要があります。

1. ターゲットビルド・データベースの作成
2. クエリの作成
3. クエリの実行

### ターゲットビルド・データベースの作成

まず解析対象のプログラムを作成し、そのプログラムから CodeQL 用のデータベースを作成します。このデータベースにはソースコードのすべての情報が含まれることになります。

例で使ったプログラムを流用しましょう。次のプログラムを `printf.c` として保存してください。

```
#include <stdio.h>

int main(void) {
    printf("it works!");
}
```

次に、`codeql` コマンドでデータベースを作成します。

```
$ codeql database create ./db --language=cpp --command="clang printf.c -o printf"
```

`./db` で出力先を指定します。 `--command` で指定したコマンドを実行し、ビルドをトラップして情報を収集します。

内部的には `codeql/cpp/tools/linux64/extractor` がこれを行っています。使用しているコンパイラが `extractor` と互換か否かは、ドキュメントで確認することができます。

https://codeql.github.com/docs/codeql-overview/supported-languages-and-frameworks/

ただし普通に嘘をついているので気をつけてください。

例えば、一部の GCC の拡張構文は `extractor` が対応していません。`database create` する際はなんのエラーも表示しませんが、該当ファイルのデータベースだけ作成されないためかなり見つけづらい不具合が発生します。 その場合、`db/log/build-tracer.log` を見ることでエラーを確認することができます。

具体的にこれが問題になるのは、Linux Kernel のデータベースを作成する際です。`clang-17` 以外のコンパイラを使うと `fs/select.c` など一部のファイルがデータベースに登録されなかったりします。

小さいプログラムで試す分には `gcc` でも良いですが、本格的に使う場合は、少し古めの `clang` を使うと良いです。

### クエリの作成

実際にクエリを作成します。

先ほど例示したクエリを再掲します。

```
import cpp

from FunctionCall fc
where fc.getTarget().getName() = "printf"
select fc, fc.getEnclosingFunction(), fc.getLocation()
```

これを、search_printf.ql 等、名前をつけて保存してください。

### クエリの実行

クエリの実行前に、内部で使っているパッケージ (import cpp) をローカルに入れます。

次の内容を、qlpack.yaml として保存してください。

```
name: sample
version: 0.1.0
library: true

dependencies:
  codeql/cpp-all: "*"
```

次に、qlpack.yaml と同じディレクトリで、

```
codeql pack install
```

とすることで、指定したパッケージをいれることができます。

最後に、指定したクエリと対象のデータベースを指定し、クエリを実行します。

```
codeql query run -j0 ./search_printf.ql -d ./db
```

以上が、指定したクエリを指定したデータベースで実行する方法になります。これ以外にも、CSV 出力や Graphvis で描画可能な dot ファイルを出力する方法もありますが、ここでは割愛します。

## クエリの基礎

CodeQL のクエリは Prolog の方言と言われています。 筆者は Prolog や Datalog に詳しくないため、ここでは、最低限クエリを書く具体的な方法だけを説明します。

クエリは基本的に次のような構造をしています。

```
from ValueType value
where cond
select value
```

### from 句

from 句では変数の宣言ができます。冒頭の例では FunctionCall という型の変数 fc を宣言していました。 ここで重要なのは、fc を宣言した瞬間、**すべての関数呼び出し**が fc に入っているということです。

FunctionCall fc と宣言した時点で、実際にすべての関数呼び出しを集合として評価し、その一つ一つが fc に入ると捉えてください。

例えば、「すべての関数を列挙する」というクエリを考えてみましょう。冒頭の例では関数コールを列挙するために FunctionCall を使っていました。今回は関数を列挙したいので、関数を表す型である Function を使います。

```
import cpp

from Function func
select func, func.getLocation()
```

先程の説明通りにこれを解釈すると、Function func とした時点ですべての**関数という実体**が集合としてまとめられ、その一つ一つが func に入ると予想されます。 実行してみましょう。

```
|       func       |                              col1                              |
+------------------+--------------------------------------------------------------+
| __overflow       | file:///usr/include/stdio.h:950:12:950:21                      |
| __uflow          | file:///usr/include/stdio.h:949:12:949:18                      |
 (蛮殛)
| rename           | file:///usr/include/stdio.h:160:12:160:17                      |
| remove           | file:///usr/include/stdio.h:158:12:158:17                      |
| main             | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:3:5:3:8 |
```

実行すると、100 行ぐらいの長めの結果が帰ってきます。これは、ソースコードの先頭で #includ <stdio.h> と書いた際に、stdio.h に定義されている関数を CodeQL がまとめて関数として認識するからです。

Function はデータベース内のすべての関数を扱うため、where 句で条件を絞らないと、すぐに結果が膨大になってしまいます。また、より大きなデータベースでは、out-of-memory として問題になることもあります。 条件を忘れているか緩すぎるせいで結果が大きくなっていますので、クエリを見直してみてください。

from 句では複数の変数を同時に宣言することができます。

```
from Function func1, Function func2, Function func3
```

例として、Function 型の値を 3 つ宣言しました。 このように書くと、func1, func2, func3 それぞれにすべての関数が入ります。

つまり、

$$(\text{func1}, \text{func2}, \text{func3}) \in \text{Function} \times \text{Function} \times \text{Function}$$

みたいなことになります。 使われていない変数や、無駄な計算のいくつかは最適化によって削られますが、いたずらに変数を増やすと思わぬ直積を生む原因になるので注意してください。

### where 句

次に、where 句を用いて具体的に値をフィルタする方法を説明します。

where では、論理式を and や or を使ってつなげ、複雑な条件を記述することができます。

論理式には、=, !=といった演算や、predicate を用いることができます。

冒頭の where 句をもう一度考えてみましょう。

```
where fc.getTarget().getName() = "printf"
```

fc は FunctionCall の実体でした。すると、getTarget は FunctionCall に生えているメソッドのように見えます。実際にドキュメントを当たってみましょう。

「CodeQL C++ FunctionCall」等で調べるとドキュメントが出てきます。CodeQL は C++以外にもたくさんのライブラリを公開しているので、参照しているドキュメントが適切な言語のものか確認しましょう。

https://codeql.github.com/codeql-standard-libraries/cpp/semmle/code/cpp/exprs/Call.qll/type.Call$FunctionCall.html

少し下の方にスクロールすると、“Gets the function called by this call.” という predicate を見つけることができます。 クリックして詳細を閲覧すると、次のようなシグネチャが確認できます。

```
Function getTarget()
```

`getTarget` は `Function` という実体を返すようです。`Function` は先程扱いましたね。

つまり、`fc.getTarget()`は、「`fc` が実際に呼んでいる関数の実体を手に入れる」という意味の式として解釈することができます。

続けて、`getName` も調べましょう。先程のページの帰り値の型の `Function` の部分をクリックすると `Function` のドキュメントを閲覧することができます。

`getName` の説明に、“Gets the name of this declaration.” とあります。また、同じようにシグニチャを確認すると、`string getName()` とあります。

ここに来てやっと身近な型が出てきました。`string` は皆さんの想像する文字列型だと考えていただいて良いです。どうやら、`getName` メソッドは `Function` の名前を文字列として返す predicate のようです。

これらを踏まえ、もう一度 `where` 句の条件を考察しましょう。

```
where fc.getTarget().getName() = "printf"
```

この `where` 句がどのような機能を持つのかなんとなく想像できるようになったと思います。つまり、「関数コール `fc` の内、`fc` が呼んでいる関数の名前が `"printf"`と一致するもの」を条件にしているのです。

我々が書いた `printf.c` の `main` では

```
printf("it works!");
```

とあり、この条件に合致した関数コールのはずです。`fc` には少なくともこの関数呼び出しが代入されるはずです（他の部分で `printf` を呼んでいる関数があった場合、それも条件に合致する）。

これで条件の絞り込みが終わりました。最後にこれを表示しましょう！あと少しです！

### `select` 句

`from` で定義し、`where` で絞り込んだ値を、実際にどのように表示するのかを指定する句です。

```
select fc, fc.getEnclosingFunction(), fc.getLocation()
```

ここで、`fc`, `fc.getEnclosingFunction()`, `fc.getLocation()` の 3 つを表示しています。ここで使える predicate も先ほど参照したドキュメントにある通りです。

getEnclosingFunction は、その関数コールが行われた関数を、getLocation はその関数コールが行われた場所を返します。 printf("it works!"); は、main 関数内の関数呼び出しであり、コードとしては 4 行目に存在します。

```
|       fc       | col1 |                             col2                             |
+----------------+------+--------------------------------------------------------------+
| call to printf | main | file:///home/iwancof/WorkSpace/CodeQL/article/printf.c:4:5:4:10 |
```

クエリの実行によって得られた結果は我々の期待するものと一致していそうです。

### まとめ

このように、ソースコードをデータベースライクに扱い、クエリを書くことによって情報を抽出することができました。

where では、指定した型の実体すべてを含む集合を計算し、where で型に付随している predicate を用いて条件の絞り込みを行い、最後に select によって値を表示する、といった感じです。

## クエリの応用

更に複雑なクエリを書くため、もう少し応用的な文法を学びましょう。 ですが、printf.c ではコードが小さく面白くないため、もう少し大きなコードベースを対象に解析を行います。

対象は何でも良いです。自分の興味のあるコードベースを選び、ビルドする時に CodeQL のデータベースを作成するようにしてください。

ここでは Linux Kernel の commit_creds 関数を対象に解析を行います。この関数に対する知識は必須ではありませんが、正確性を無視しざっくりと説明をすると「現在実行中のタスクの権限を更新する」という機能を持ちます。主に CTF 等で権限昇格に使われる馴染み深い関数であるため解析対象にしましたが、深い意味はありません。

カーネルのビルド方法に関してはここでは詳細に説明しません。いくつかの注意点として、デバッグ用の関数や KASAN などを有効にしている場合、抽出されるデータベースにも含まれるため、それらがノイズになる方は適宜 config を編集すると良いです。

また先程も少し言及しましたが、GCC でビルドを行うと一部ファイルが認識されなくなるバグが存在します。これは記事作成時の最新バージョン(2.21.3)でも再現します。 少し試してみるだけであれば何でも良いですが、本格的に使いたい方は clang-17 を入れたうえで次のコマンドを実行してください。

```
codeql database create \
  --language=cpp \
  --command="make -j4 LLVM=1 LLVM_IAS=1" \
  kerneldb
```

それ以外は、先ほどの環境構築と同じことをすればオーケーです。

さて、作成したデータベースに対して早速クエリを書いていきましょう。

まず、先程学習したことを使って、「commit_creds が呼び出している関数一覧」を取得してみます。

```
import cpp

from Function commit_creds, FunctionCall callee_call, Function callee
where
  commit_creds.getName() = "commit_creds" and
  callee_call.getEnclosingFunction() = commit_creds and
  callee_call.getTarget() = callee
select callee, callee.getLocation()
```

最初に、commit_creds という関数を探し、FunctionCall の中で commit_creds に含まれているものを取得します。最後に、その実体を callee として記録するという流れです。

```
|         callee          |                                col1                                |
+-------------------------+--------------------------------------------------------------------+
| __builtin_constant_p    | file://:0:0:0:0                                                    |
| __builtin_expect        | file://:0:0:0:0                                                    |
| atomic_long_read        | file:///linux/include/linux/atomic/atomic-instrumented.h:3186:1:3186:16 |
| get_current             | file:///linux/arch/x86/include/asm/current.h:44:44:44:54           |
（籠の扣わ篠簫）
```

おそらく大量の関数がリストアップされると思います。一つ一つクエリを改善していきましょう。

まず、from にて定義している callee_call ですが、これは最終結果には現れない、いわば「一時変数」のようなものです。この定義の規模なら問題になりませんが、より大きなクエリを書く準備のため、from を整理することから始めましょう。

CodeQL には、from 以外にも変数を宣言できる場所があります。その代表例が exists をはじめとする、量化子で変数を束縛する文法です。

これを使うと、先程のクエリを次のように書き換えることができます。

```
import cpp

from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  exists(FunctionCall fc |
    fc.getEnclosingFunction() = commit_creds and
    callee = fc.getTarget()
  )
select callee, callee.getLocation()
```

exists の中で一時的に FunctionCall 型の fc を宣言しています。数学的な言い回しをするなら、「そのような fc が存在する」的な感じです。どのような fc か、それを記述しているのが | で区切られたあとの式です。内容は一つ前のものと変わりませんね。

これによって from による定義を一つ減らすことができました。exists 以外にも、forall や sum, concat など、表現力を上げるための文法や便利機能が沢山あります。 ただ、名前から機能を容易に想像できる上、公式ドキュメントも充実しています。これらを使う頃には相応に理解度が深まっていると思うので、今回の記事では省略させていただきます。

次に、クエリの結果を見てみましょう。大量に出力されたエントリは、そのほとんどがコンパイラビルトインです。呼び出されている関数であることに変わりはないですが、解析対象としては不適です。これを弾く条件を書きましょう。 コンパイラビルトインはソースコード中に定義を持たないという性質を利用し、論理式を構築します。

```
import cpp

from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  exists(FunctionCall fc |
    fc.getEnclosingFunction() = commit_creds and
    callee = fc.getTarget()
  ) and
+  callee.hasDefinition()
select callee, callee.getLocation()
```

callee.hasDefinition() という部分を追加しました。これにより、定義を持たない関数群をまとめて除外することができます。このクエリを実行すると、ちょうど 10 個程度の結果が帰ってきます。 余力がある方は、kernel/fork.c に存在する実装と見比べてみて、ちゃんと漏れなくリストアップされているか確認してみてください。

さて、ここから更に次のステップに進みたいところですが、その前段階として今作った「ある関数から呼ばれている関数一覧を取る」という操作を切り出して、再利用可能な形にしたいと思います。 普通のプログラミングでいうところの、関数呼び出しみたいなものです。

先程のクエリを次のように書き換えて見ましょう。

```
import cpp

predicate calls(Function a, Function b) {
  exists(FunctionCall fc |
    fc.getEnclosingFunction() = a and
    fc.getTarget() = b and
    b.hasDefinition()
  )
}

from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  calls(commit_creds, callee)
select callee, callee.getLocation()
```

predicate という単語が出てきました。これによって先程のロジックを calls として切り出しています。注意しなければならないのは、calls はあくまでも predicate、つまり述語であるという点です。

これも正確性を多少犠牲にした表現になりますが、最初は bool を返す関数のようなものだと考えると納得できます。where 句のフィルタは、各 predicate がすべて真のときに全体を真にするという認識で十分です。

from 句で Function を 2 つ定義しているため、カーネル内のすべての関数のペアが commit_creds と callee に入ります。まず commit_creds のほうは名前によってフィルタされ、次に callee の方は calls によってフィルタされます。「次に」という日本語を使いましたが、これらは逐次的に処理されているわけではなく、これらの条件を入れ替えても意味は変わりません。

```
from Function commit_creds, Function callee
where
  calls(commit_creds, callee) and
  commit_creds.getName() = "commit_creds"
select callee, callee.getLocation()
```

愚直に解釈すると、まずすべての caller callee ペアを見つけ、**その後**に commit_creds をフィルタしているように見えますが、それでは現実的な時間で応答するのは難しいでしょう。 これを可能にしているのは、単純に最適化の精度が高いからです。クエリをどの順番で書いてもその性能が変わることは（おそらく）ありません。安心してわかりやすい順番で書けばよいです。

では、気を取り直してクエリの実装を続けましょう。次の目標は先ほどのクエリを応用し、「commit_creds から到達できる関数一覧」とします。つまりさっきのクエリを再帰的に適用してみよう！という感じです。

ちょうど、ここまで学習した文法と機能だけで、そのような機能を持つクエリを書けるような構成にしました。挑戦してみたい方は挑戦してみてください！

筆者の答えを次に示します。

```
predicate calls(Function a, Function b) {
  exists(FunctionCall fc |
    fc.getEnclosingFunction() = a and
    fc.getTarget() = b and
    b.hasDefinition()
  )
}

predicate calls_rec(Function a, Function b) {
  calls(a, b)
  or
  exists(Function transit | calls_rec(a, transit) and calls(transit, b))
}

from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  calls_rec(commit_creds, callee)
select callee, callee.getLocation()
```

transit という経由地点が**存在する**(exists)なら、そこから更に b に到達できるか、みたいな感じです。

実行すると、おおよそ 8000 エントリほどの結果が返ってくると思います。結果を見てみると、一見全然関係ない関数が含まれていますが、カーネルは複雑なので、どこかに panic でもあったのでしょう。そこから広がっているのだと思います。 コンパイラビルトインを除外したときのように関係ないファイルや関数を一つ一つ取り除いても良いですし、これらをフィルタしていく作業は実際の解析において非常に重要なことですが、この記事の趣旨とは外れるためこれも割愛させていただきます。

その代わりに、ここでは賢く maxdepth を設定しようと思います。 幸いにして、文字列比較に string を使ったように、int というプリミティブな型があるので、これを使って実装します。 しかし、先程も強調しましたが、predicate は関数ではありません。「引数として int maxdepth を渡す」という認識でいると正しく実装できなくなってしまいます。

先程の説明の繰り返しになってしまいますが、predicate はあくまでも、引数の型を見て取りうる値を全部代入してみて結果が真ならば true を返す、と認識するべきです（正確にはこれも間違っていますが、詳細は後述します）。

それを踏まえ、次のようなクエリを考えてみます。

```
predicate calls_rec(Function a, Function b,int depth) {
  (calls(a, b) and depth = 1)
  or
  (exists(Function transit | calls_rec(a, transit, depth - 1) and calls(transit, b)))
}
from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  calls_rec(commit_creds, callee, 2)
select callee, callee.getLocation()
```

「深さ 2 の関数一覧を取る」という意図でこのクエリを書きました。直接的な関係がある時の depth は 1 であり、それ以外の場合は depth-1 で transit まで到達できるなら該当、という感じです。

しかし、このクエリが終了することはありません。predicate は全部代入して確かめるということを意識すると、int として存在するすべての値を入れているため、膨大な時間がかかってしまうためです（CodeQL の int は 32bit 整数なので、無限ではない）。

これを判別する目安としては、実行に一分以上の時間がかかるか否かをチェックすればよいです。一分で終わらない場合、基本的にクエリが正しくないと考えて良いと思います。

この問題を解決するにはどうすればよいでしょうか？ 方法の一つとしては、int をそのまま使わないという手があります。

CodeQL には型の継承のような機能が存在し、コンストラクタ等に自身に対する条件を書くことで、そのクラスが持ちうる値の可能性を制御することができます。

```
class DepthLimit2 extends int {
  DepthLimit2() { this = [0 .. 2] }
}

predicate calls_rec(Function a, Function b, DepthLimit2 depth) {
  calls(a, b) and depth = 1
  or
  exists(Function transit | calls_rec(a, transit, depth - 1) and calls(transit, b))
}

from Function commit_creds, Function callee
where
  commit_creds.getName() = "commit_creds" and
  calls_rec(commit_creds, callee, 2)
select callee, callee.getLocation()
```

このようにすることで、現実的な時間で depth を指定できるようになります。指定する深さを 2 としてハードコードしていますが、CodeQL に Generics のような機能がないため、不可避の制限だと考えてください。

また、_ を使うことで、距離 2 以下で到達できる関数一覧を得ることもできます。

```
calls_rec(commit_creds, callee, _)
```

_ は don't care を意味し、何でも良いみたいな意味を持ちます。型として指定している DepthLimit2 は、0,1,2 のどれかを取るため、2 以下で到達できる関数一覧を取ることができます。

最後に、ここで学んだことを利用して、ちょっとしたクエリを書く練習をしてみましょう。回答には、count という言語機能を使っても構いません。 count は、条件に合う要素を数え上げることができます。例えば、num_func = count(Function f | f.getDefinition()) とすると、num_func には、定義をもつ関数の数が入ることになります。

問題: 次の仕様を満たすクエリを書いてください。「commit_creds から呼び出される関数の内、そこから深さ 2 以下で到達できる関数の数を集計するクエリ」

結果はおおよそ次のようになると思います。

```
|       callee       | num_child |
+--------------------+-----------+
| atomic_long_read   |         5 |
| get_current        |         0 |
| key_fsgid_changed  |         5 |
| key_fsuid_changed  |         5 |
| put_cred_many      |         9 |
| get_cred           |         2 |
| dec_rlimit_ucounts |         3 |
| inc_rlimit_ucounts |         4 |
| _printk            |         2 |
| gid_eq             |         1 |
| uid_eq             |         1 |
| set_dumpable       |         4 |
| proc_id_connector  |        29 |
| cred_cap_issubset  |         3 |
```

答え:

```
from Function commit_creds, Function callee, int num_child
where
  commit_creds.getName() = "commit_creds" and
  calls(commit_creds, callee) and
  num_child = count(Function child | calls_rec(callee, child, _))
select callee, num_child
```

### まとめ

これにて、クエリの応用は終了です。まだまだ紹介しきれていない機能ばかりなのですが、すべてを紹介することはできないため、ドキュメントを理解する上で重要な思想や考え方を中心に説明してきました。

これ以上高度なことをしようするなら必ずドキュメントを読むことになるでしょう。しかし、ここで学んだ考え方を応用すれば容易に理解できるはずです。みなさんが面白いクエリを書き、楽しい CodeQL ライフを送れることを期待しています！

## 最小不動点

おまけパートです。

最後に、今まで紹介してきたクエリが内部でどのように処理されているのか紹介します。

クエリの作成に役立つかもしれないし、役立たないかもしれません。興味がある方だけ読んでいただければ結構です。

さて、本文中にも登場した predicate に関して、もう少し掘り下げてみましょう。より単純な例を考えます。

```
predicate getANumber(int x) {
  x = 0
  or
  x <= 100 and getANumber(x - 1)
}

from int v
where
  getANumber(v)
select v
```

これを実行すると、0 から 100 までの整数が列挙されると思います。0 から 100 の整数を getANumber に渡した場合、中の論理式が真になる、と言い換えることもできます。

getANumber の内容をよく見てみましょう。まず、getANumber(0)は真になります。getNumber(1)はどうでしょうか？ x=0 の方は満たしませんが、x <= 100 and getANumber(x - 1)は満たします。 なぜなら、getANumber(0)はすでに集合に含まれることを確認したからです。

同様に x=2 も条件を満たします。これを繰り返していくと x=101 までこの操作を続けることができ、そのタイミングで条件を満たさなくなります。実際の結果から、CodeQL も同じように考えていることがわかります。

では、内部のクエリエンジンはどのようにしてこれを計算しているのでしょうか？そのキーになるのが、**最小不動点**です。

クエリエンジンはまず、getANumber を満たすことができる引数の集合を空集合として考えます。次に、引数の型である int の中から適当な値を代入してみて、それが条件を満たすかチェックします。条件を満たすなら、集合に加えます。 次に、拡大した集合を参照しつつ、もう一度適当な値を getANumber に与えてみて、それが条件を満たすかどうかチェックします。新しく条件に合致する値が見つかった場合、それを集合に加えます。

この操作を繰り返していき、どこかのタイミングで集合の成長が止まる瞬間が来たとします。そのタイミングの集合を「getANumber(x)が真になる集合」として扱うことで、クエリを処理しています。

これを踏まえると、クエリの応用セクションで、評価が終了しないクエリの原因を推測することができます。

```
predicate calls_rec(Function a, Function b,int depth) {
  (calls(a, b) and depth = 1)
  or
  (exists(Function transit | calls_rec(a, transit, depth - 1) and calls(transit, b)))
}
```

まず適当な a, b, depth を与えてみて、それが条件を満たすなら追加、という操作を繰り返していきます。先程の例では、クエリエンジンの最適化によって x として与える値を適切に選び、現実的な時間でクエリを処理できました。

しかし、どうやら calls_rec は**複雑すぎる**ようで、int としてあり得るすべての値を代入してみて集合の構成を試みます。その結果、全く終わる気配がないクエリが誕生した、というカラクリです。

この事実を念頭に置くと、生の int や string をそのまま扱うのは危険な気がしてきます。この直感は正しく、DepthLimit2 という制限した型を定義したように、クエリエンジンが無

駄な探索を行わないように工夫するのが大切です。 見識が広い方は、解集合プログラミングとのつながりに気がついたかもしれません。

ここで、ちょっと意地悪なクエリを考えてみましょう。先程のアルゴリズムによる集合の構成では、**集合の要素が減る**ような場合は考慮されていませんでした。そもそもそのようなクエリを書くことは可能でしょうか？

実は工夫をするとそんなクエリを書くことができます。考えてみたい読者の方は、ここで止めて考えてみてください。同様に、ここまで学んだ機能で書くことができます。

答え:

```
predicate p(int x) {
  x = count(int e | p(e))
}
```

まず最初に、`0` を与えてみます。最初 `p` を満たす集合は空集合なので、`0` は条件に合致します。すると集合の個数が一つ増え、`count(int e | p(e))` は `1` になります。 すると今度は `0` が条件を満たさなくなってしまいます。

このクエリを実際に処理するとどのような結果が得られるでしょうか？

CodeQL は賢く、このような意地悪なクエリの処理を拒否してきます。

```
ERROR: Non-monotonic recursion: target -!-> target (no_poset.ql:6,21-27)
```

つまり、内部でクエリを処理する際に、集合が縮まないことを要求しているのです。( 詳しい人に指摘されるのが怖いので及び腰になりますが ) 数学的に表現するなら、集合間に包含関係によって半順序性を入れ、その順序に基づいた単調性を要求していると言い換えることもできます。

束論によると、半順序性を持った集合は、必ず最小不動点を持ちます ( 詳しくは、Knaster–Tarski の定理 )。CodeQL においては、処理をする前段階で単調性を確かめることで、その後の最小不動点計算の結果が存在することを使っているようです。もちろん、それが現実的な時間で終わるかは判別してくれませんが。

これらは、ドメイン理論などとも関わりがありますが、筆者はそもそも数学に明るくないため、これ以上の言及は避けます。怖いので。

## おわりに

ここで紹介した機能以外にも、たくさんの便利機能があり、マスターすることで日々のソースコードリーティングがより捗ることでしょう！

特に、CodeQL の目玉機能である、`taint tracking` はぜひ皆さんに紹介したかったのですが、内容が難解になりすぎてしまったので割愛しました。無念。

質問や誤植等がありましたら、気軽に連絡していただけると嬉しいです。

最後まで読んでいただきありがとうございました！