Variant Analysis with QL

Pavel Avgustinov, Kevin Backhouse, Man Yue Mo

Semmle Ltd publications@semmle.com

Abstract. As new security problems and innovative attacks continue to be discovered, program analysis remains a burgeoning area of research. QL builds on previous attempts to enable declarative program analysis through Datalog, but solves some of the traditional challenges: Its object-oriented nature enables the creation of extensive libraries, and the query optimizer minimizes the performance cost of the abstraction layers introduced in this way. QL enables agile security analysis, allowing security response teams to find all variants of a newly discovered vulnerability. Their work can then be leveraged to provide automated on-going checking, thus ensuring that the same mistake never makes it into the code base again. This paper demonstrates declarative variant analysis by example.

1 Introduction

QL is an object-oriented, declarative, logic language which has been successfully applied to the domain of program analysis. The idea is not new: Datalog-like formalisms are established as an effective way of implementing complex flow-sensitive analyses like points-to and dataflow [3, 6]. However, the expressiveness and modularity of QL allows us to do something interesting and novel: Analysis building blocks are packaged up in reusable libraries, which can then be easily instantiated to particular problems encountered in the wild.

This process is known as *variant analysis*. Given a newly discovered bug or vulnerability, how can we find more instances (variants) of the same issue? Security research teams around the world currently do this by laborious manual search, but formally encoding the concern in QL can make the task significantly more achievable. The end result is a democratization of program analysis: Everyone is empowered to propose and implement new checks, or to proactively eradicate bugs of a certain class from the codebase they work on.

2 QL for ad-hoc variant analysis

In order to make program analysis a standard part of the software developer's toolbox, we must make the barrier to getting started as low as possible. QL's approach of high-level, declarative queries, separate from the messy details of how to parse and compile the code, is a big step in that direction.

We will use a recent example of this use case for a whirlwind tour of QL (for which [1] provides a full introduction). A vulnerability had been caused by an invalid overflow check in C++. The code looked something like this:

```
if (cur + offset < cur)
  return false;
use(cur + offset);
```

In the above, `cur` and `offset` were unsigned 16-bit values, and the check was intended to detect arithmetic overflow and wraparound. Unfortunately, it was flawed: The C/C++ standard specifies that 16-bit values are promoted to 32-bits for the purposes of arithmetic, and in this wider type the addition cannot possibly overflow. The comparison then has a 32-bit value on its left and a 16-bit value on its right, and will promote its right-hand side to 32 bits. Thus, a potential overflow in the 16-bit range would never be detected.

Here is the QL query we wrote as the customer was describing the issue:

```
from Variable v, RelationalOperation cmp, AddExpr add
where v.getAnAccess() = add.getAnOperand()
  and add = cmp.getAnOperand()
  and v.getAnAccess() = cmp.getAnOperand()
  and v.getType().getSize() < 4
  and not add.getExplictlyConverted().getType().getSize() < 4
select cmp, ``Bad overflow check.''
```

This directly encodes the verbal definition of the problem into machine-checkable QL. We are going to reason about a variable `v`, a relational operation `cmp` and an addition operation `add`, where: (a) an access to `v` figures as an operand of `add`, (b) the addition `add` is an operand of the comparison `cmp`, (c) another access to `v` is an operand of `cmp`, (d) the type of `v` is smaller than 4 bytes (*i.e.* subject to arithmetic promotion), and (e) the addition `add` is not explicitly truncated to a type smaller than 4 bytes.

This flagged the original problem and a few other variants in the same code base that had escaped manual detection. Indeed, the check is of general utility — there is nothing codebase-specific in it! In running it across other code bases, we have discovered numerous instances of incorrect overflow guards, and we are aware of several that turned out to be security vulnerabilities.

3 Variant analysis from building blocks

Writing more sophisticated analyses is usually considered a highly specialist task. It is also extremely challenging to create general-purpose analyses that have high precision on arbitrary code bases. Both of these problems are addressed by creating easily reusable "analysis building blocks" in the standard QL libraries. We can provide a framework for doing dataflow analysis, or points-to analysis, or taint tracking, which encapsulates the complexities of the target language and allows end users to easily achieve remarkable bespoke results.

A recent spate of Java deserialization vulnerabilities suggests that numerous unknown variants might still be lurking in well-known code bases. The problem arises when untrusted (attacker-controlled) data is passed to some deserialization mechanism, which then creates attacker-determined objects. Merely this act can lead to arbitrary code execution [4]. Here is our QL query for such problems:

```
import java
import semmle.code.java.dataflow.FlowSources
import UnsafeDeserialization

from UnsafeDeserializationSink sink, RemoteUserInput source
where source.flowsTo(sink)
select sink.getMethodAccess(), "Unsafe deserialization of $@.",
      source, "user input"
```

Most of the logic is encapsulated in the imported libraries, which stand ready to be extended. The query itself checks that a value in the `RemoteUserInput` class flows to `sink`, which is an `UnsafeDeserializationSink`. The most common Java APIs are already modelled by the QL libraries; for example, `RemoteUserInput` will cover HTTP servlet request objects and data read over network sockets, while `UnsafeDeserializationSink` models Java serialization, but also frameworks like XStream and Kryo. Each of these concepts, as well as, if necessary, the dataflow computation itself may be extended separately.

In the work that led to the discovery of CVE-2017-9805 [5], we were auditing Apache Struts and noticed that it had a somewhat convoluted code pattern for deserialization: implementors of the interface `ContentTypeHandler` would be responsible for deserializing data passed in via the first parameter of their `toObject` method. Adding support for this[1] was a simple matter of providing an additional kind of `UserInput` by adding this class definition:

```
/** Mark the first argument of `toObject` as user input source **/
class ContentTypeHandlerInput extends UserInput {
  ContentTypeHandlerInput() {
    exists(Method m | m.asParameter() = m.getParameter(0) |
      m.getSignature() = "toObject(java.io.Reader,java.lang.Object)" and
      m.getDeclaringType().getASupertype+().hasQualifiedName(
            "org.apache.struts2.rest.handler", "ContentTypeHandler")
    )
  }
}
```

With this definition in scope, the remainder of the library will kick in and perform the additional tracking, leading us to the result.

A more general customization hook for variant analysis is provided by the libraries in the form of so-called *dataflow configurations*. The user can specify a set of sources, a set of sinks and (optionally) a set of sanitizers that should prevent flow. The libraries then take care of the rest.

An example of this in action can be seen in our work on CVE-2017-13782 [2], inspired by past issues with the `dtrace` component of Apple's MacOS kernel. This subsystem allows user-supplied dtrace scripts to perform various operations, with their data stored in an array of registers. Such data should not be used in sensitive operations like pointer arithmetic or array subscripting, at least without

[1] More recent versions of the QL libraries are able to track this pattern out of the box, but the same customization mechanisms are still available.

careful validation. The dataflow configuration that identified the vulnerability was defined as follows:

```
class DtraceRegisterFlow extends DataFlow::Configuration {
  DtraceRegisterFlow() { this = "DtraceRegisterFlow" }
  /** Our sources are register reads like `regs[i]`. */
  override predicate isSource(DataFlow::Node node) {
    exists(ArrayExpr regRead | regRead = node.asExpr() |
      regRead.getArrayBase().(VariableAccess).getTarget().hasName("regs") and
      regRead.getEnclosingFunction().hasName("dtrace_dif_emulate")
    )
  }
  /** Our sinks are array index or pointer arithmetic operations. */
  override predicate isSink(DataFlow::Node node) {
    exists(Expr use | use = node.asExpr() |
      use = any(ArrayExpr ae).getArrayOffset() or
      use = any(PointerAddExpr add).getAnOperand()
    )
  }
}
```

This concern is much too codebase-specific to be flagged by a standard analysis, but the vulnerability leaks arbitrary kernel memory and is, therefore, of high severity. QL's approach of providing analysis building blocks makes it feasible to create bespoke checks in such cases.

4 Conclusion

We have discussed the idea of *variant analysis*: Given a bug or vulnerability, how can we find other variants of the same problem? QL makes it very easy to write simple analyses, and allows users to bring state-of-the-art flow analysis to bear when necessary, all while abstracting away the complexity. The underlying libraries are continuously evolving, and as they become more powerful existing queries written against them automatically increase in power as well.

In our experience, this approach resonates strongly with both security researchers and developers, who embrace the idea that program analysis should be everyone's concern. The analyses created during this process tend to distribute fairly evenly into three categories:

Codebase-specific Concerns specific to a particular code base, its APIs and invariants, like our "bad use of `dtrace` data" example.

Domain-specific Analyses that apply to code written for a particular domain, possibly while requiring some customization. A good example is unsafe deserialization, which is applicable to any code that serializes data.

General Checks that are applicable to any code base, usually concerning common pitfalls of the target language, like the "bad overflow guard" query.

QL is used to analyze over 50,000 open-source projects on the https://lgtm.com portal, and a query console is available to run custom analyses. The default libraries are available as open-source at https://github.com/lgtmhq/lgtm-queries.

References

1. Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: object-oriented queries on relational data. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, volume 56 of *LIPIcs*, pages 2:1–2:25. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.
2. Kevin Backhouse. Using QL to find a memory exposure vulnerability in Apple's macOS XNU kernel. In *lgtm.com blog*, 2017. `https://lgtm.com/blog/apple_xnu_dtrace_CVE-2017-13782`.
3. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, 2009.
4. Chris Frohoff and Gabriel Lawrence. Deserialize My Shorts, Or How I Learned to Start Worrying and Hate Java Object Deserialization. In *AppSec California*, 2015.
5. Man Yue Mo. Using QL to find a remote code execution vulnerability in Apache Struts. In *lgtm.com blog*, 2017. `https://lgtm.com/blog/apache_struts_CVE-2017-9805`.
6. John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, 2005.