Input-Driven Recursion: Ongoing Security Risks

White Paper

December 18, 2024

Written by: **Alexis Challande and Brad Swain**

Table of Contents

Introduction

For readers in a hurry, this white paper could be summarized in a single sentence:

Recursing on user input is bad; don't do it.

To elaborate a bit more, we found that recursion issues—which the security community has largely dismissed—can lead to stack overflows that quickly take down systems with little effort. Using a relatively simple CodeQL query—which we developed in stages, detailed at the end of this whitepaper—we uncovered several vulnerabilities in a variety of projects, all of which have a large user base and excellent secure development practices.

We disclosed these vulnerabilities to the respective project owners, who have fixed them:

- ElasticSearch (in PatternBank)
- OpenSearch (in FilterPath, parseGeometryCollection, and validatePatternBank)
- CVE-2024-7254 in Protocol Buffers (Google)
- Function rewrite in Guava (Google)
- CVE-2024-47072 in xstream (XStream)

This white paper presents several case studies that demonstrate the harms of recursive functions and reviews the effectiveness of available solutions for addressing them. Our takeaway: don't rely on `StackOverflowErrors` to catch recursive attacks; instead, avoid using recursion altogether, and rewrite recursive functions as iterative functions.

Background: Recursion 101

Recursion is one of the first “Eureka!” moments that new programmers experience. The humble `Fibonacci` is a classic introduction to the concept. A recursive function calls itself to solve a problem by breaking it down into smaller, similar subproblems.

```
public int fibonacci(int n)  {
   if(n == 0) return 0;
   else if(n == 1) return 1;
  else
     return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Figure 1: Recursive `Fibonacci` function from Stack Overflow

Recursion can be elegant, simple, and, most importantly, practical. Once understood, it is the go-to method for dealing with nested structures, whether traversing a tree, visiting nodes in a graph, or parsing JSON.

If you remember CS101 (actually CS106B), the three “musts” of recursions are:

1. Your code must have a case for all valid inputs.
2. You must have a *base case* that makes no recursive calls.
3. When you make a recursive call, it should be to a simpler instance of the same problem, and you should make forward progress towards the base case.

Although elegant, most recursive functions can be flawed if they process untrusted data.

The Dangers of Recursion

Computers have finite resources, including a finite number of stack frames (corresponding to function call depth) available during program execution. Large amounts of recursion can, therefore, exhaust the stack and generate a `StackOverflowError` (or similar ones in different languages). For example, the `fibonacci` example from Stack Overflow[1] will crash on a large input (like `0xabcdabcd`):

```
Exception in thread "main" java.lang.StackOverflowError
	at Fibonacci.fibonacci(Fibonacci.java:8)
	at Fibonacci.fibonacci(Fibonacci.java:8)
	at Fibonacci.fibonacci(Fibonacci.java:8)
```

Figure 2: `StackOverflowError` from Stack Overflow

Violating any of the three "musts" in the previous section can result in a stack exhaustion problem, which can be a security concern, as the case studies in this white paper highlight.

The limit of the bug

Some applications have availability requirements and must gracefully handle alien inputs. For instance, with all the latest and greatest DDoS protection, your newest web application should not be taken offline by a single deeply nested JSON POST request. In contexts where availability is crucial, using recursion to process **untrusted** input is a real risk with the potential for actual harm.

However, stack overflows result **in little or no real-world harm** in most contexts. Imagine compiling a Rust program that includes "1+" repeated ten thousand times. As expected, `rustc` overflows the stack and crashes. The only harm is to our ego, and we can fix the issue by rerunning the compiler, this time on a less audacious program.

[1] Yet another unintended pun

An Overview of the Different Solutions

Our favorite: Don’t use recursion

Don’t recurse over user inputs—instead, rewrite them as iterative functions. For instance, after we reported a recursion issue in Guava to Google, Google rewrote its function cycle detection function to change it from a recursive to an iterative one.

A recommended practice would be to add recursions to the forbidden list of actions to perform on user-tainted data (along with deserialization, code evaluation, and many others).

Second-best choice: Use a depth counter

If an iterative approach is not feasible, combining a recursive approach with a depth counter is possible, but this comes with drawbacks.

```
public static final int MAX_DEPTH = 100;

public static int fibonacci(int n) throws InputTooBigException {
   return _fibonacci(n, 0);
}

public static int _fibonacci(int n, int depth) throws InputTooBigException  {
   if (depth >= MAX_DEPTH)
       throw new InputTooBigException();

   if(n == 0) return 0;
   else if(n == 1) return 1;
   else
       return _fibonacci(n-1, depth+1) + _fibonacci(n-2, depth+1);
}
```

Figure 3: Fibonacci updated with a depth counter

Manually adding depth to each call is both tedious and error-prone. The extra boilerplate reduces readability and removes some of the elegance that makes recursion an attractive choice. As such, few recursive functions include a depth limit.

Not a solution: Catch StackOverflow errors

A common but flawed approach is to catch `StackOverflowError` to handle excessive recursion. The reasoning might sound correct: since StackOverflowError is essentially just a built-in counter for how many recursive calls are allowed, why not use it as a natural limit and catch the error when it occurs?

However, this approach is problematic. When a `StackOverflowError` is thrown, the system is already under pressure: the stack space has been largely exhausted, and the resources (memory allocations, function calls …) have already been used.

A system under heavy load or resource pressure rarely fails predictably. While the ideal outcome may be that the system simply refuses to handle new requests, the more likely outcome is a cascade of other problems. It may be a failed database connection, timeout, logging failure, failed background task, or any number of seemingly unrelated issues. Unless the system was designed from the ground up with resource constraints in mind and built with graceful degradation and failure handling at every level, resource pressure will likely cause unpredictable failures throughout the entire system

As Tatu Saloranta, the maintainer of jackson-databind, perfectly summarized: trying to catch and handle a StackOverflowError as a countermeasure is not practical - it is already too late.

Exploring the Vulnerabilities

While the theory is nice, we wanted to see if real-world projects supported it. We wrote a CodeQL query for recursion cases on potentially untrusted user input in popular Java projects to do this. Indeed, we found this problematic pattern alarmingly common; below, we highlight some examples.

Protobuf Java Lite

According to Protobuf's official documentation:

> Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler. (source)

Parsing untrusted data is notoriously tricky, and security researchers have targeted parsers for every format. Protocol buffers are a solution developed by Google to provide a serialized exchange format with automatically generated parsers in various languages. They are used extensively both within Google and in the greater ecosystem.

However, they are also vulnerable to recursion error attacks.

For instance, one could crash a Java application parsing an external message using the `protobuf-lite` library by simply sending this one message.

```
with open("recursive.data", "wb") as f:
    f.write(bytearray([19] * 5_000_000))
```

Figure 4: A malicious message in Protobuf

It will throw a `StackOverflowError`. The problem lies in how Protobuf parses `Unknown Fields`.

> Unknown fields are well-formed protocol buffer serialized data representing fields that the parser does not recognize. For example, when an old binary parses data sent by a new binary with new fields, those new fields become unknown fields in the old binary. (source)

When this issue is combined with Groups—a deprecated feature that is still parsed because of backward compatibility—you get an explosive mix:

1. A group can contain another group.
2. The new group is parsed as an unknown field if the attacked schema does not contain a group.
3. An unknown group can contain another group.
4. Goto 2

Below is an excerpt of the code responsible for the parsing:

```
final boolean mergeOneFieldFrom(B unknownFields, Reader reader) throws IOException {
  int tag = reader.getTag();
  /* ... */
  switch (WireFormat.getTagWireType(tag)) {
    /* ... */
    case WireFormat.WIRETYPE_START_GROUP:
      final B subFields = newBuilder();
      /* ... */
      mergeFrom(subFields, reader);
      /* ... */
      return true;
    /* ... */
  }
}

final void mergeFrom(B unknownFields, Reader reader) throws IOException {
  while (true) {
    if (reader.getFieldNumber() == Reader.READ_DONE
        || !mergeOneFieldFrom(unknownFields, reader)) {
      break;
    }
  }
}
```

Figure 5: `mergeFrom` function in Protobuf (`protobuf/java/core/src/main/java/com/google/protobuf/UnknownFieldSchema.java#59–98`)

The exciting thing about this vulnerability is that it has one precondition on the attacked target: it must use the Java lite version of the Protocol Buffer library. There are no requirements for the scheme used by the targeted application.

While C++ API's official documentation advises discarding `Unknown Fields` for security reasons, it advises doing it *after* parsing the message. At this point, it is already too late.

While Protobuf parsing is usually resilient against recursion attacks (using depth counters), Google forgot about this one code path during development. We responsibly disclosed this issue to Google, and it was assigned CVE-2024-7254.

While investigating this problem, we found it also applied to other Protobuf implementations, including:

- Rust-protobuf, an unofficial implementation of Protocol buffers in Rust
- The pure Python implementation of Protobuf (by Google)

Elasticsearch

Elasticsearch is an open-source search and analytics engine built on Apache Lucene. It enables users to store, search, and analyze large volumes of data quickly and in near-real time. However, several of its features are also vulnerable to recursion attacks.
First, Elasticsearch supports a feature called Glob Patterns. The documentation states the following:

> Match a String against the given pattern, supporting the following simple pattern styles: "xxx*", "*xxx", "*xxx*" and "xxx*yyy" matches (with an arbitrary number of pattern parts), as well as direct equality.

The figure below shows an excerpt of the `globMatch` function:

```java
public static boolean globMatch(String pattern, String str) {
    /* ... */
    int firstIndex = pattern.indexOf('*');
    if (firstIndex == -1) {
        return /* ... */
    }
    if (firstIndex == 0) {
        if (pattern.length() == 1) {
            return true;
        }
        int nextIndex = pattern.indexOf('*', firstIndex + 1);
        if (nextIndex == -1) {
            return /* ... */
        } else if (nextIndex == 1) {
            // Double wildcard "**" - skipping the first "*"
            return globMatch(pattern.substring(1), str);
        }
        /* ... */
        return false;
    }
    return /* ... */
}
```

Figure 6: `globMatch` function (`elasticsearch/libs/core/src/main/java/org/elasticsearch/core/Glob.java#27–59`)

The issue arises when handling patterns with repeated wildcard symbols (*). Logically, multiple consecutive wildcards are equivalent to a single wildcard. This implementation handles consecutive wildcards by removing the first wildcard and making a recursive call to match against the remaining pattern. The algorithm works beautifully for a pattern like `**foo`. It removes the first wildcard and performs a recursive call to allow the matching to continue. On the other hand, this function does not handle many consecutive wildcards as beautifully—a pattern with thousands of consecutive wildcards results in thousands of recursive calls.

A malicious user who can submit an arbitrary glob pattern could cause a stack overflow. Because Elastic's developers extensively use the recursion pattern in their codebase, the `globMatch` issue was not an isolated case. For example, Elasticsearch has a custom regular expression dialect called Grok that supports named patterns:

```
DIGIT [0-9]
CHAR [a-zA-Z]
DIGITORCHAR %{DIGIT} | %{CHAR}
```

Each line contains a name followed by the pattern associated with that name. Patterns can be regular expressions, references to other patterns, or a mix of both. This design leads to the potential for patterns to create unresolvable cycles.

```
SELF %{SELF}
A %{B}
B %{A}
```

Because Grok patterns are user-defined and cannot be trusted to be cycle-free, the Grok parser includes a check for cycles, implemented with recursion. An abbreviated version is included below, with comments added by us.

```
private static void innerForbidCircularReferences(
   Map<String, String> bank,
   String patternName,
   List<String> path,
   String pattern) {
   // Check for patterns referencing themselves e.g. "A %{A}"
   if (patternReferencesItself(pattern, patternName)) {
       /* throw */
   }

   // check all sub-pattern names found in the pattern
   for (int i = pattern.indexOf("%{"); i != -1; i = pattern.indexOf("%{", i + 1)) {
       String otherPatternName = /* get the sub-pattern */
       path.add(otherPatternName);
       String otherPattern = bank.get(otherPatternName);
       if (otherPattern == null) {
           /* throw */
       }

       // check if any sub-patterns reference patternName
       innerForbidCircularReferences(bank, patternName, path, otherPattern);
   }
}
```

Figure 7: Abbreviated source (`elasticsearch/libs/grok/src/main/java/org/elasticsearch/grok/PatternBank.java#84–103`)

The bank maps pattern names to the corresponding pattern: `PatternName` is the pattern being checked, `path` contains all patterns seen on this recursive chain, and `pattern` is the current pattern being checked. To illustrate, the figure below shows a pattern bank:

```
A %{B}
B %{C}
C %{A}
```

Figure 8: Pattern bank definition

The call stack at the innermost call chain to check “A” will look like this:

```
// %{B} does not reference A
innerForbidCircularReferences(bank@123, “A”, [], “%{B}”)
// %{C} does not reference A
innerForbidCircularReferences(bank@123, “A”, [“B”], “%{C}”)
// %{B} does not reference A
innerForbidCircularReferences(bank@123, “A”, [], “%{B}”)
// %{C} does not reference A
innerForbidCircularReferences(bank@123, “A”, [“B”], “%{C}”)
//  error! %{A} references A
innerForbidCircularReferences(bank@123, “A”, [“B”, “C”], “%{A}”)
```

Figure 9: Execution trace for `innerForbidCircularReferences`

Unfortunately, the recursion depth depends on the number of nested patterns. A malicious user could create many deeply nested patterns, causing the circular reference check to overflow.

```
A0 %{A1}
A1 %{A2}
A2 %{A3}
…
An %{An+1}
```

Figure 10: A deeply nested pattern bank

```
innerForbidCircularReferences(bank@123, “A0”, [], “%{A1}”)
innerForbidCircularReferences(bank@123, “A1”, [“A0”], “%{A2}”)
innerForbidCircularReferences(bank@123, “A2”, [“A0”, “A1”], “%{A3}”)
innerForbidCircularReferences(bank@123, “A3”, [“A0”, “A1”, “A2”], “%{A4}”)
…
innerForbidCircularReferences(bank@123, “An”, [“A0”, “A1”, “A2”, …, “An”],
“%{An+1}”)
```

Figure 11: Execution trace for `innerForbidCircularReferences` with a malicious pattern

The previous example underlines another unfortunate side effect of recursive processing. If each recursive call allocates a new stack frame, it also performs allocations for the newly created stack frame. In this case, it will allocate a new string at each iteration. Replacing this string allocation with a vast array can lead to an `Out-of-memory` error, even before exhausting the stack size. The key takeaway is that catching `StackOverflowErrors` gracefully is not enough to catch *recursive attacks*.

Jenkins

Jenkins is a popular CI/CD server designed to be highly extensible through its plugin architecture. As each plugin can have dependencies, Jenkins implants a `CyclicGraphDetector` to detect cycles between them.

The `visit` method presented below is responsible for cycle detection. As you have probably figured out, the function uses a recursive pattern.

```
private void visit(N p) throws CycleDetectedException {
    /* ... */
    visiting.add(p);
    for (N q : getEdges(p)) {
        /* ... */
        if (visiting.contains(q))
            detectedCycle(q);
        visit(q);
    }
    visiting.remove(p);
    /* ... */
}
```

Figure 12: `visit` function of the `CyclicGraphDetector`
(`jenkins/core/src/main/java/hudson/util/CyclicGraphDetector.java#44–58`)

The Cycle detection works flawlessly: if a node has already been marked as visited before the end of the recursion, it will return a throw a `CycleDetectedException`. However, this will not be enough for deeply nested graphs because the cycle will overflow.

Creating hundreds of plugins to trigger the recursion is not a realistic attack vector because you already have code execution on the targeted system. Nonetheless, the `CyclicGraphDetector` is also used to detect cycles in `Environment` variables. Adding multiple environment variables, as defined below, also triggers the same recursion.

```
VAR1=${VAR0}
VAR2=${VAR1}
VAR3=${VAR2}
…
VAR10000=${VAR9999}
```

Figure 13: Generating a huge number of env variables

It is worth mentioning that we **don't believe this bug to be a security vulnerability**. It requires an attacker to be able to influence the Jenkins building environment. If an attacker already has such capabilities, they probably have other options now than just crashing the build server.

Jackson-core

Jackson-core is the foundational library for Jackson, providing base low-level parser and generator abstractions for various data formats, such as JSON, XML, CSV, CBOR, and others.

Jackson-core provides a `JsonPointer` class. A `JsonPointer` is a string that identifies a specific value within a JSON object. In the JSON object below, the JSON Pointer `/a/b/c/target` identifies the key `target`, where each of `a`, `b`, `c`, and `target` are segments of the `JsonPointer`.

```
{ "a": { "b": { "c": "target": {} } } }
```

Figure 14: A sample JSON object

The `JsonPointer` class provides a `head` function, which constructs a new `JsonPointer` with the *last* segment dropped. In our previous example, this would construct the `/a/b/c` pointer.

The `head` method implementation is challenging because a JSON pointer object contains a `_nextSegment` attribute, which holds the remainder of the segment (in our example,

/b/c/target). Indeed, computing the head for /a/b/c/target means that pointers /c, /b/c, and finally /a/b/c need to be built in that order. This pattern of traversing to the bottom and walking back up the segments is a perfect fit for recursion, and that is precisely how the head function is implemented.

This method calls the _constructHead function (displayed below), which recursively calls itself to construct a new JsonPointer object without the dropped segment.

```
protected JsonPointer _constructHead(int suffixLength, JsonPointer last)
{
    if (this == last) {
        return EMPTY;
    }
    JsonPointer next = _nextSegment;
    String str = toString();
    // !!! TODO 07-Oct-2022, tatu: change to iterative, not recursive
    return new JsonPointer(str.substring(0, str.length() - suffixLength), 0,
            _matchingPropertyName,
            _matchingElementIndex, next._constructHead(suffixLength, last));
}
```

Figure 15: Head construction in JsonPointer (jackson-core/src/main/java/com/fasterxml/jackson/core/JsonPointer.java#832–843)

If head() is called on a user-controlled pointer, a malicious user could create a pointer with many sections and cause a stack overflow. The test case below shows how to trigger the overflow.

```
@Test
void maliciousPath() throws Exception
{
    final String INPUT = "/a".repeat(10000);
    JsonPointer ptr = JsonPointer.compile(INPUT);
    assertEquals("/a".repeat(9999), ptr.head().toString());
}
```

Figure 16: Proof of concept demonstrating the recursion overflow

However, it should be noted that Jackson's maintainer considers that attackers should not be able to use methods invoking head(), preventing this issue from being a security risk. They nonetheless fixed the issue in a recent commit, following our report.

The Broader Impact of Recursions

If denial of service is a part of your threat model, we recommend adding this additional item to your Secure Coding Practices Checklist:

- **Never perform unbounded recursions on user tainted data.**

With relatively little effort, we skimmed through the results from a straightforward CodeQL query and found a handful of denial-of-services vulnerabilities. Although we highlighted specific projects here, the patterns are common. With more time, we will likely continue to find issues in other projects.

Our approach limited itself to recursive chains of length 3 (A calls B calls C calls A), but longer chains are even more dangerous because they exhaust the stack more quickly.

We only looked at a handful of applications, and only in Java. Similar issues exist in other languages, as highlighted by the issues affecting several language-specific implementations in Protobuf and *safety-first* languages like Rust. This suggests an extensive landscape of potential threats waiting to be discovered and mitigated.

Background: How We Developed the Query

This section describes our iterative process of writing increasingly sophisticated CodeQL queries—from our first basic attempts to our final optimized version that detected real vulnerabilities.

Why we used CodeQL

CodeQL is like SQL for source code—you write queries to find patterns in code, such as security bugs or questionable coding practices. (We provide much more detail on CodeQL in our Testing Handbook.) That makes it a useful tool for building queries to find dangerous recursive functions.

You can find recursive functions without CodeQL—one approach is to use Tree Sitter to build an Abstract Syntax Tree of your code and search for cycles in the call graph. We tried this with Rust, where CodeQL support is not available yet. But this approach quickly breaks down with real codebases; you'd need complex logic to handle language features like polymorphism and macro expansions. These challenges led us to focus on Java instead, where we could leverage CodeQL's robust analysis capabilities.

First query: Learning from mistakes

Here's our first attempt at finding recursive functions:

```
import java

// Warning, do-not-use, extremely slow
from MethodCall ma, MethodCall mb
where
     ma.getMethod() = mb.getEnclosingCallable()
     and ma.getEnclosingCallable() = mb.getMethod()
select ma, "Mutual recursion between $@ and $@", ma.getMethod(),
ma.getMethod().getName(), mb.getMethod(), mb.getMethod().getName()
```

Figure 17: Initial CodeQL query

This query worked fine on our test cases but hit a wall with real codebases. The problem? It performs a cartesian product between every method call in the codebase. With 1,000 method calls, you're looking at 1,000,000 comparisons. For example, this request does not finish in a reasonable time on Elasticsearch codebase.

A quick check of the CodeQL documentation revealed our mistake—avoiding cartesian products is literally their first performance tip. It's time to optimize.

Subsequent tentatives: Speed, precision, and depth

Our optimized query checks for direct recursion (functions calling themselves) and second-order recursion (function A calls B, which calls back to A). With just a few lines of CodeQL, we've already surpassed what we could do with Tree Sitter:

```
import java

from MethodCall ma
where
    // Check for self-recursive functions
    ma.getMethod() = ma.getEnclosingCallable()
    // Or check for second-order recursion
    or exists(
        Method m |
        m = ma.getMethod() and m.getACallee() = ma.getEnclosingCallable()
    )
select ma, "Recursion starting in $@", ma.getMethod(), ma.getMethod().getName()
```

Figure 18: Improved CodeQL query

The performance difference when we ran both queries against the Elasticsearch codebase was dramatic: whereas the first query didn't finish after 10 minutes, the second found 825 results in just five seconds.

Still, we saw opportunities to improve it. Although our query was finding hundreds of recursive functions, many were in test and benchmark files—code that would never run in production. Here's how we filtered them out:

```
predicate isTestPackage(RefType referenceType) {
   exists(
       string packageName |
       packageName = referenceType.getPackage().getName() |
       packageName.matches("%test%")
       or packageName.matches("%benchmark%")
   )
   or referenceType.getName().toLowerCase().matches("%test%")
}

from MethodCall ma
where
   (
       ma.getMethod() = ma.getEnclosingCallable()
       or exists(
           Method m |
           m = ma.getMethod() and m.getACallee() = ma.getEnclosingCallable()
       )
   )
   and not isTestPackage(ma.getMethod().getDeclaringType())

select ma, "Recursion starting in $@", ma.getMethod(), ma.getMethod().getName()
```

Figure 19: Improved CodeQL query filtering out results in test files

This filter reduced our results from 825 to 683 functions—all in production code. While the CodeQL Java library offers a classify predicate for similar filtering, we found that our custom solution worked better for our needs.

At this point, we were catching functions that call themselves (order 1) and functions that call back to themselves through one intermediary (order 2). But what about longer chains? Recursion chains of order 3 or higher can be even more dangerous—they consume more stack frames relative to the input size needed to trigger them.

Here's how we extended our query to detect recursion chains up to order 4:

```
...

predicate isBefore(Method a, Method b) {
    a.getLocation().getStartLine() < b.getLocation().getStartLine()
  }

abstract class RecursiveCall extends MethodCall {

    Method m1;
    Method m2;
    Method m3;
    Method m4;

    abstract string asString();

    string toString(Method m) {
        result = m.getName() + "(" + m.getLocation().getStartLine()  + ")" + " -> "
    }
}



...

class RecursiveCallOrder2 extends RecursiveCall {

    RecursiveCallOrder2() {
        m1 = this.getMethod()
        and m2 = m1.getACallee()
        and m2.getACallee() = m1
        and isBefore(m1, m2)
    }

    override string asString() {
        result = "Recursion(2): " + toString(m1) + toString(m2)  + m1.getName()
    }
}

...

from RecursiveCall recursiveCall
where
 not isTestPackage(recursiveCall.getMethod().getDeclaringType())
select recursiveCall, "$@", recursiveCall, recursiveCall.asString()
```

Figure 20: Recursion detection until level 4

We built this detection system by:

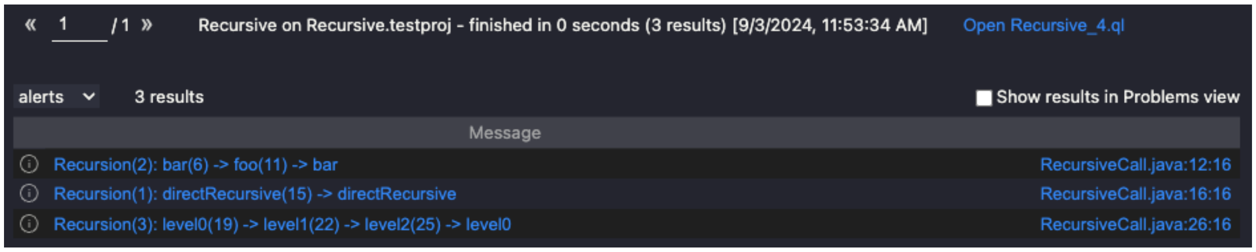
1. Creating an abstract base class that handles common recursion chain logic
2. Extending it with specialized classes for each recursion order (1 through 4)
3. Adding deduplication to avoid reporting the same chain multiple times
4. Including visualization logic to show the complete call chain

First, we created an abstract `RecursiveCall` class to handle the common logic all recursion chains share. This base class manages the methods in the chain and provides a framework for visualizing the call sequence.

We then built specialized classes for each recursion order, from direct self-recursion through complex four-function chains. Each class implements its logic for detecting specific recursion patterns.

Since recursion chains are loops, they could be detected starting from any function in the chain. For example, if A calls B calls C calls A, we could report it starting from A, B, or C. To avoid this redundancy, we implemented deduplication by always using the function with the lowest line number as the starting point[2].

Finally, we added visualization logic that transforms these chains into human-readable output. Instead of just listing function names, our output shows the complete path of recursive calls with line numbers, making it much easier to understand how functions call each other in complex scenarios.



```
« 1 /1 »   Recursive on Recursive.testproj - finished in 0 seconds (3 results) [9/3/2024, 11:53:34 AM]   Open Recursive_4.ql

alerts   3 results                                                           Show results in Problems view
                                        Message
Recursion(2): bar(6) -> foo(11) -> bar                                       RecursiveCall.java:12:16
Recursion(1): directRecursive(15) -> directRecursive                         RecursiveCall.java:16:16
Recursion(3): level0(19) -> level1(22) -> level2(25) -> level0               RecursiveCall.java:26:16
```

Figure 21: Result visualization on a test database

The final tentative: Exploring path queries

So far, we've only used alert queries in CodeQL. However, another type that seemed promising for our use case was path queries. The CodeQL documentation states that "you

[2] This is obviously not perfect, as chains can span multiple files, but a good enough approximation for our use case.

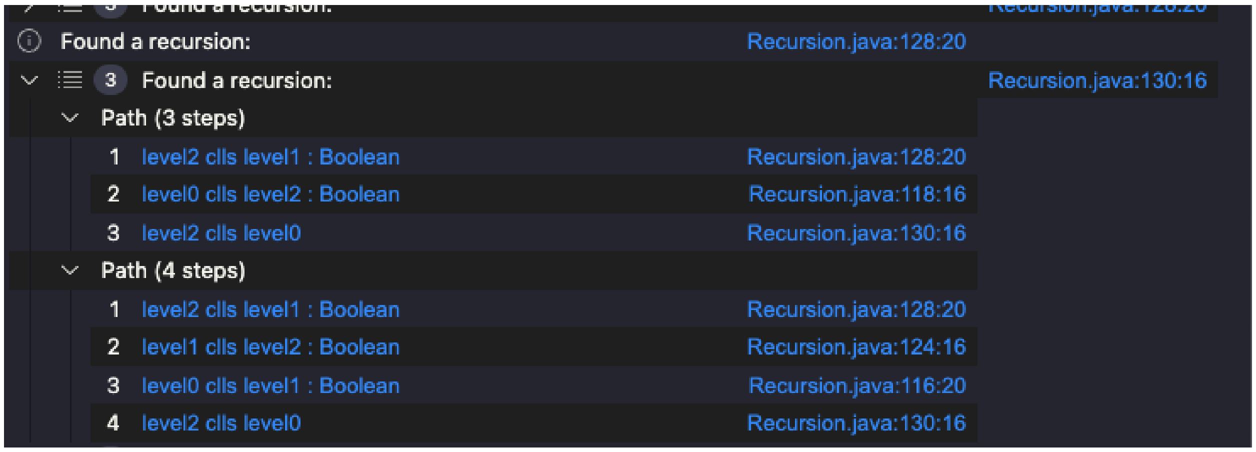
can create path queries to visualize the flow of information through a codebase." The initial version of this white paper stopped here because we did not find a way to detect recursive functions using path queries, but a reply to our question on the CodeQL Slack channel unblocked us. The figure below shows our final query to detect recursions:

```
import java
import semmle.code.java.dataflow.DataFlow

class RecursionSource extends MethodCall {
  RecursionSource() { not isTestPackage(this.getCaller().getDeclaringType()) }
}

module RecursiveConfig implements DataFlow::StateConfigSig {
  class FlowState = Method;

  predicate isSource(DataFlow::Node node, FlowState state) {
    node.asExpr() instanceof RecursionSource and
    state = node.asExpr().(MethodCall).getCaller()
  }

  predicate isSink(DataFlow::Node node, FlowState state) {
    node.asExpr() instanceof RecursionSource and
    state.calls+(node.asExpr().(MethodCall).getCaller()) and
    node.asExpr().(MethodCall).getCallee().calls(state)
  }
}

module RecursiveFlow = DataFlow::GlobalWithState<RecursiveConfig>;

import RecursiveFlow::PathGraph

from RecursiveFlow::PathNode source, RecursiveFlow::PathNode sink
where RecursiveFlow::flowPath(source, sink)
select sink.getNode(), source, sink, "Found a recursion: "
```

Figure 22: Recursion detection using a path query

While writing this query, the main problem was to find a solution to keep track of the nodes already seen in the current path. This is elegantly solved using a `State` variable to keep track of the initial source of the current recursion chain.

To keep the figure lighter, we removed any refinements (such as deduplication logic) from the query in the excerpt, but the full query is available in the repository. Unlike the other queries, the new one flag chains with bigger lengths. It also yields a much nicer visualization of the query results.



```
Found a recursion:                                   Recursion.java:128:20
3  Found a recursion:                                         Recursion.java:130:16
   Path (3 steps)
     1  level2 clls level1 : Boolean                 Recursion.java:128:20
     2  level0 clls level2 : Boolean                 Recursion.java:118:16
     3  level2 clls level0                           Recursion.java:130:16
   Path (4 steps)
     1  level2 clls level1 : Boolean                 Recursion.java:128:20
     2  level1 clls level2 : Boolean                 Recursion.java:124:16
     3  level0 clls level1 : Boolean                 Recursion.java:116:20
     4  level2 clls level0                           Recursion.java:130:16
```

Figure 23: Result visualization using the path query

Query limitations and further improvements

Our query can still be improved in several ways:

Filtering out false positives: Our query flags all recursive functions, including in some safe cases where it shouldn't. Consider this function from Elasticsearch:

```
public double q(double k, double compression, double n) {
    if (k <= 0) {
        return Math.exp(k * Z(compression, n) / compression) / 2;
    } else {
        return 1 - q(-k, compression, n);
    }
}
```

Figure 24: Scale function (`elasticsearch/libs/tdigest/src/main/java/org/elasticsearch/tdigest/ScaleFunction.java#322–328`)

While technically recursive, this function is safe—its recursion depth is always limited to 2 calls because the else branch inverts k from positive to negative, triggering the base case on the next call.

Here's a more complex example from Elasticsearch's geohash implementation:

```java
public static final String getNeighbor(String geohash, int level, int dx, int dy) {
    // ... skipped
    if (level == 1) {
        // ... skipped ...
    } else {
        // ... skipped ...
        if (nx >= 0 && nx <= 7 && ny >= 0 && ny <= 3) {
            return geohash.substring(0, level - 1) + encodeBase32(nx, ny);
        } else {
            String neighbor = getNeighbor(geohash, level - 1, dx, dy);
            return (neighbor != null) ? neighbor + encodeBase32(nx, ny) : neighbor;
        }
    }
}
```

Figure 25: Computation of neighbors using geo hashes (`elasticsearch/libs/geo/src/main/java/org/elasticsearch/geometry/utils/Geohash.java#201–244`)

At first glance, this looks dangerous: the function recursively calls itself while decrementing the level parameter. With a sufficiently high initial level (say 10,000), this could cause a stack overflow. However, Elasticsearch's codebase limits this level parameter to a maximum of 12.

Our query can't easily detect these kinds of safety checks. One solution would be to add heuristics that recognize when recursive functions use strictly decreasing parameters and then verify their bounds through manual review.

Identifying attacker control: Our query's biggest limitation to our query is its inability to determine whether attackers can control the recursive input. This is crucial for distinguishing between genuine security vulnerabilities and harmless recursive functions.

Consider this parser from Elasticsearch:

```
public static CollectorResult fromXContent(XContentParser parser) throws IOException
{
    // ... skipped
    while ((token = parser.nextToken()) != XContentParser.Token.END_OBJECT) {
        // ... skipped
        } else if (token == XContentParser.Token.START_ARRAY) {
            if (CHILDREN.match(currentFieldName, parser.getDeprecationHandler())) {
                while ((token = parser.nextToken()) !=
XContentParser.Token.END_ARRAY) {
                    children.add(CollectorResult.fromXContent(parser));
        // ... skipped
    }
    return new CollectorResult(name, reason, time, children);
}
```

Figure 26: fromXContent (`elasticsearch/server/src/main/java/org/elasticsearch/search/profile/query/CollectorResult.java#121–157`)

The code above depicts a standard recursive parser implementation—it handles nested data structures by calling itself on each nested element. In theory, deeply nested input could crash the parser:

```
[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[...[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[
```

Figure 27: Example of an invalid array definition

But context matters. In Elasticsearch, this parser (`XContentParser`) only processes data from trusted sources within the cluster. An attacker can't provide malicious input, so the recursion isn't a security risk.

This kind of contextual analysis—determining whether untrusted input can reach a recursive function—is beyond what our current query can detect. That's why results still require manual review before being classified as security issues.

Support for other languages: While this post focuses on Java, CodeQL supports Python, C/C++, Go, and other languages. The concepts we've covered—detecting recursive functions, filtering test code, and handling complex call chains—can be adapted to find similar vulnerabilities across these languages. The main differences will be in the syntax and language-specific features you'll need to account for.

Key Takeaways and Next Steps

These CodeQL queries dramatically improved our ability to find recursive patterns in large codebases. What previously required extensive manual review can now be automated—though you still need expertise to evaluate which findings are security issues.

Most importantly, remember that recursive functions are dangerous when they process untrusted input. While our queries can find recursive patterns, determining whether an attacker can control the input requires manual analysis. Always validate and limit recursive depth when handling data that could come from untrusted sources.

We're grateful to GitHub for making CodeQL accessible through excellent documentation and example queries from their Security Labs team. These resources helped us develop our approach, and we hope sharing our experience here helps others in the security community leverage CodeQL for their own analysis needs.

Want help securing your codebase with custom CodeQL queries? Our team can adapt these techniques to find recursion vulnerabilities and other security patterns specific to your code.

Contact us to learn more about our CodeQL development services.

About the Authors

Brad Swain, security engineer at Trail of Bits

Brad Swain is a security engineer on Trail of Bits Research and Engineering team with a background in program analysis and experience with compilers and operating systems. Brad has worked on engineering projects and DARPA funded research programs aimed at developing tools for analyzing and securing software systems. Prior to joining Trail of Bits, Brad previously served as Principal Investigator on a Small Business Innovation Research (SBIR) grant at a startup where he worked on static analysis tools for high performance computing (HPC) programs. Brad has a Master of Computer Science degree from Texas A&M University. He has authored and co-authored papers at the International Conference for High Performance Computing, Networking, Storage and Analysis (SC) and the International Conference on Software Engineering (ICSE).

Alexis Challande, security researcher at Trail of Bits

Alexis Challande (@DarkaMaul) is a security researcher working at Trail of Bits, where he is part of the Ecosystem group. While his work mostly focuses on open-source supply chain security, he is also deeply interested in applying new static analysis tools and techniques to uncover vulnerabilities at scale.

Prior to joining Trail of Bits, Alexis earned his PhD while working at a reverse engineering company, where his research centered on automated vulnerability patch detection within the Android ecosystem. His doctoral work contributed to advancing automated methods for identifying and analyzing security patches across large-scale mobile operating systems.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
https://www.trailofbits.com
info@trailofbits.com