Clang Static Analyzer Documentation

Release 6

Analyzer Team

Aug 13, 2018

Contents

Contents:

CHAPTER 1

Debug Checks

The analyzer contains a number of checkers which can aid in debugging. Enable them by using the "-analyzer-checker=" flag, followed by the name of the checker.

1.1 General Analysis Dumpers

These checkers are used to dump the results of various infrastructural analyses to stderr. Some checkers also have "view" variants, which will display a graph using a 'dot' format viewer (such as Graphviz on OS X) instead.

- debug.DumpCallGraph, debug.ViewCallGraph: Show the call graph generated for the current translation unit. This is used to determine the order in which to analyze functions when inlining is enabled.
- debug.DumpCFG, debug.ViewCFG: Show the CFG generated for each top-level function being analyzed.
- debug.DumpDominators: Shows the dominance tree for the CFG of each top-level function.
- debug.DumpLiveVars: Show the results of live variable analysis for each top-level function being analyzed.
- debug.ViewExplodedGraph: Show the Exploded Graphs generated for the analysis of different functions in the input translation unit. When there are several functions analyzed, display one graph per function. Beware that these graphs may grow very large, even for small functions.

1.2 Path Tracking

These checkers print information about the path taken by the analyzer engine.

- debug.DumpCalls: Prints out every function or method call encountered during a path traversal. This is indented to show the call stack, but does NOT do any special handling of branches, meaning different paths could end up interleaved.
- debug.DumpTraversal: Prints the name of each branch statement encountered during a path traversal ("IfStmt", "WhileStmt", etc). Currently used to check whether the analysis engine is doing BFS or DFS.

1.3 State Checking

These checkers will print out information about the analyzer state in the form of analysis warnings. They are intended for use with the -verify functionality in regression tests.

- debug.TaintTest: Prints out the word "tainted" for every expression that carries taint. At the time of this writing, taint was only introduced by the checks under experimental.security.taint.TaintPropagation; this checker may eventually move to the security.taint package.
- debug.ExprInspection: Responds to certain function calls, which are modeled after builtins. These function calls should affect the program state other than the evaluation of their arguments; to use them, you will need to declare them within your test file. The available functions are described below.

(FIXME: debug.ExprInspection should probably be renamed, since it no longer only inspects expressions.)

1.3.1 ExprInspection checks

- `void clang_analyzer_eval(bool);`

  Prints TRUE if the argument is known to have a non-zero value, FALSE if the argument is known to have a zero or null value, and UNKNOWN if the argument isn't sufficiently constrained on this path. You can use this to test other values by using expressions like "x == 5". Note that this functionality is currently DISABLED in inlined functions, since different calls to the same inlined function could provide different information, making it difficult to write proper -verify directives.

  In C, the argument can be typed as 'int' or as '_Bool'.

  Example usage:

```
clang_analyzer_eval(x); // expected-warning{{UNKNOWN}}
if (!x) return;
clang_analyzer_eval(x); // expected-warning{{TRUE}}
```

- `void clang_analyzer_checkInlined(bool);`

  If a call occurs within an inlined function, prints TRUE or FALSE according to the value of its argument. If a call occurs outside an inlined function, nothing is printed.

  The intended use of this checker is to assert that a function is inlined at least once (by passing 'true' and expecting a warning), or to assert that a function is never inlined (by passing 'false' and expecting no warning). The argument is technically unnecessary but is intended to clarify intent.

  You might wonder why we can't print TRUE if a function is ever inlined and FALSE if it is not. The problem is that any inlined function could conceivably also be analyzed as a top-level function (in which case both TRUE and FALSE would be printed), depending on the value of the -analyzer-inlining option.

In C, the argument can be typed as 'int' or as '_Bool'.

Example usage:

```
int inlined() {
  clang_analyzer_checkInlined(true); // expected-warning{{TRUE}}
  return 42;
}

void topLevel() {
  clang_analyzer_checkInlined(false); // no-warning (not inlined)
  int value = inlined();
  // This assertion will not be valid if the previous call was not inlined.
  clang_analyzer_eval(value == 42); // expected-warning{{TRUE}}
}
```

- `void clang_analyzer_warnIfReached();`

  Generate a warning if this line of code gets reached by the analyzer.

  Example usage:

```
if (true) {
  clang_analyzer_warnIfReached();  // expected-warning{{REACHABLE}}
}
else {
  clang_analyzer_warnIfReached();  // no-warning
}
```

- `void clang_analyzer_numTimesReached();`

  Same as above, but include the number of times this call expression gets reached by the analyzer during the current analysis.

  Example usage:

```
for (int x = 0; x < 3; ++x) {
  clang_analyzer_numTimesReached(); // expected-warning{{3}}
}
```

- `void clang_analyzer_warnOnDeadSymbol(int);`

  Subscribe for a delayed warning when the symbol that represents the value of the argument is garbage-collected by the analyzer.

  When calling 'clang_analyzer_warnOnDeadSymbol(x)', if value of 'x' is a symbol, then this symbol is marked by the ExprInspection checker. Then, during each garbage collection run, the checker sees if the marked symbol is being collected and issues the 'SYMBOL DEAD' warning if it does. This way you know where exactly, up to the line of code, the symbol dies.

  It is unlikely that you call this function after the symbol is already dead, because the very reference to it as the function argument prevents it from dying. However, if the argument is not a symbol but a concrete value, no warning would be issued.

  Example usage:

```
do {
  int x = generate_some_integer();
  clang_analyzer_warnOnDeadSymbol(x);
} while(0);  // expected-warning{{SYMBOL DEAD}}
```

- `void clang_analyzer_explain(a single argument of any type);`

  This function explains the value of its argument in a human-readable manner in the warning message. You can make as many overrides of its prototype in the test code as necessary to explain various integral, pointer, or even record-type values. To simplify usage in C code (where overloading the function declaration is not allowed), you may append an arbitrary suffix to the function name, without affecting functionality.

  Example usage:

```
void clang_analyzer_explain(int);
void clang_analyzer_explain(void *);

// Useful in C code
void clang_analyzer_explain_int(int);

void foo(int param, void *ptr) {
  clang_analyzer_explain(param); // expected-warning{{argument 'param'}}
  clang_analyzer_explain_int(param); // expected-warning{{argument 'param'}}
  if (!ptr)
    clang_analyzer_explain(ptr); // expected-warning{{memory address '0'}}
}
```

- `void clang_analyzer_dump( /* a single argument of any type */);`

  Similar to clang_analyzer_explain, but produces a raw dump of the value, same as SVal::dump().

  Example usage:

```
void clang_analyzer_dump(int);
void foo(int x) {
  clang_analyzer_dump(x); // expected-warning{{reg_$0<x>}}
}
```

- `size_t clang_analyzer_getExtent(void *);`

  This function returns the value that represents the extent of a memory region pointed to by the argument. This value is often difficult to obtain otherwise, because no valid code that produces this value. However, it may be useful for testing purposes, to see how well does the analyzer model region extents.

  Example usage:

```
void foo() {
  int x, *y;
  size_t xs = clang_analyzer_getExtent(&x);
  clang_analyzer_explain(xs); // expected-warning{{'4'}}
  size_t ys = clang_analyzer_getExtent(&y);
  clang_analyzer_explain(ys); // expected-warning{{'8'}}
}
```

- `void clang_analyzer_printState();`

  Dumps the current ProgramState to the stderr. Quickly lookup the program state at any execution point without ViewExplodedGraph or re-compiling the program. This is not very useful for writing tests (apart from testing how ProgramState gets printed), but useful for debugging tests. Also, this method doesn't produce a warning, so it gets printed on the console before all other ExprInspection warnings.

  Example usage:

```
void foo() {
  int x = 1;
```

(continues on next page)

(continued from previous page)

```
  clang_analyzer_printState(); // Read the stderr!
}
```

- `void clang_analyzer_hashDump(int);`

  The analyzer can generate a hash to identify reports. To debug what information is used to calculate this hash it is possible to dump the hashed string as a warning of an arbitrary expression using the function above.

  Example usage:

```
void foo() {
  int x = 1;
  clang_analyzer_hashDump(x); // expected-warning{{hashed string for x}}
}
```

1.4 Statistics

The debug.Stats checker collects various information about the analysis of each function, such as how many blocks were reached and if the analyzer timed out.

There is also an additional -analyzer-stats flag, which enables various statistics within the analyzer engine. Note the Stats checker (which produces at least one bug report per function) may actually change the values reported by -analyzer-stats.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search