Bounds Checking with the Clang Static Analyzer

Improvements and Insights

Donát Nagy

donat.nagy@ericsson.com

Bounds Checking

ERICSSON

Out-of-bounds memory access is a very common and severe problem in many programming languages, including C and C++. (I don't think this needs more explanation.)

Bounds Checking

ERICSSON

Out-of-bounds memory access is a very common and severe problem in many programming languages, including C and C++. (I don't think this needs more explanation.)

Clang Static Analyzer

Clang Static Analyzer is a tool included in clang (available as `clang --analyze`) which uses symbolic execution to detect bugs in C, C++ or Objective-C code.

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:

  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:

  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:

  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production
- (There are still dozens of checkers that are stuck in alpha stage for 10+ years!)

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:

  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production
- (There are still dozens of checkers that are stuck in alpha stage for 10+ years!)
- I started to work on `alpha.security.ArrayBoundV2` a few years ago

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:
  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production
- (There are still dozens of checkers that are stuck in alpha stage for 10+ years!)
- I started to work on `alpha.security.ArrayBoundV2` a few years ago
- Now it is ready for use under the name `security.ArrayBound`

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:
  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production
- (There are still dozens of checkers that are stuck in alpha stage for 10+ years!)
- I started to work on `alpha.security.ArrayBoundV2` a few years ago
- Now it is ready for use under the name `security.ArrayBound`
- I improved three main areas:

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:
  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production
- (There are still dozens of checkers that are stuck in alpha stage for 10+ years!)
- I started to work on `alpha.security.ArrayBoundV2` a few years ago
- Now it is ready for use under the name `security.ArrayBound`
- I improved three main areas:
  - Reporting details in the diagnostic messages

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:
  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production
- (There are still dozens of checkers that are stuck in alpha stage for 10+ years!)
- I started to work on `alpha.security.ArrayBoundV2` a few years ago
- Now it is ready for use under the name `security.ArrayBound`
- I improved three main areas:
  - Reporting details in the diagnostic messages
  - Eliminating false positives (one example at the end of the talk)

Improvements in bounds checking

ERICSSON

- The static analyzer consists of a core engine and 100+ checkers
- Since 2010 there were **two** bounds checking checkers:
  `alpha.security.ArrayBound` and `alpha.security.ArrayBoundV2`
- But they were both prototype (alpha) quality, not ready for use in production
- (There are still dozens of checkers that are stuck in alpha stage for 10+ years!)
- I started to work on `alpha.security.ArrayBoundV2` a few years ago
- Now it is ready for use under the name `security.ArrayBound`
- I improved three main areas:
  - Reporting details in the diagnostic messages
  - Eliminating false positives (one example at the end of the talk)
  - Changes in the engine (loop handling)

Diagnostic message improvements

ERICSSON

Before my changes there were only three possible diagnostic messages:

- Out of bound memory access (accessed memory precedes memory block)
- Out of bound memory access (accessed exceeds upper limit of memory block)
- Out of bound memory access (index is tainted)

Diagnostic message improvements

Before my changes there were only three possible diagnostic messages:

- Out of bound memory access (accessed memory precedes memory block)
- Out of bound memory access (accessed exceeds upper limit of memory block)
- Out of bound memory access (index is tainted)

Now it also reports the relevant details, e.g. (taken from the tests)

- Access of 'int' element in 'TenElements' at negative index -172

Diagnostic message improvements

Before my changes there were only three possible diagnostic messages:

- Out of bound memory access (accessed memory precedes memory block)
- Out of bound memory access (accessed exceeds upper limit of memory block)
- Out of bound memory access (index is tainted)

Now it also reports the relevant details, e.g. (taken from the tests)

- Access of 'int' element in 'TenElements' at negative index -172
- Access of the heap area at index 3, while it holds only 2 'int' elements

Diagnostic message improvements

Before my changes there were only three possible diagnostic messages:

- Out of bound memory access (accessed memory precedes memory block)
- Out of bound memory access (accessed exceeds upper limit of memory block)
- Out of bound memory access (index is tainted)

Now it also reports the relevant details, e.g. (taken from the tests)

- Access of 'int' element in 'TenElements' at negative index -172
- Access of the heap area at index 3, while it holds only 2 'int' elements
- Access of 'TenElements' with a tainted index that may be negative or too large

Diagnostic message improvements

Before my changes there were only three possible diagnostic messages:

- Out of bound memory access (accessed memory precedes memory block)
- Out of bound memory access (accessed exceeds upper limit of memory block)
- Out of bound memory access (index is tainted)

Now it also reports the relevant details, e.g. (taken from the tests)

- Access of 'int' element in 'TenElements' at negative index -172
- Access of the heap area at index 3, while it holds only 2 'int' elements
- Access of 'TenElements' with a tainted index that may be negative or too large
- Access of 'TenElements' with a tainted index that may be too large

Capabilities of the analyzer

ERICSSON

- I will demonstrate the capabilities and logic of the analyzer through a series of examples

Capabilities of the analyzer

ERICSSON

- I will demonstrate the capabilities and logic of the analyzer through a series of examples
- These examples will be all very simple – for the convenience of the audience

Capabilities of the analyzer

ERICSSON

- I will demonstrate the capabilities and logic of the analyzer through a series of examples
- These examples will be all very simple – for the convenience of the audience
  - Even a chatbot would notice that these code fragments "look fishy"

Capabilities of the analyzer

ERICSSON

- I will demonstrate the capabilities and logic of the analyzer through a series of examples
- These examples will be all very simple – for the convenience of the audience
  - Even a chatbot would notice that these code fragments "look fishy"
- But the analyzer can combine these principles and apply them rigorously

Capabilities of the analyzer

ERICSSON

- I will demonstrate the capabilities and logic of the analyzer through a series of examples
- These examples will be all very simple – for the convenience of the audience
  - Even a chatbot would notice that these code fragments "look fishy"
- But the analyzer can combine these principles and apply them rigorously
- It relies on clang tooling to "see" connections that are not spelled out in the code

Capabilities of the analyzer

ERICSSON

- I will demonstrate the capabilities and logic of the analyzer through a series of examples
- These examples will be all very simple – for the convenience of the audience
  - Even a chatbot would notice that these code fragments "look fishy"
- But the analyzer can combine these principles and apply them rigorously
- It relies on clang tooling to "see" connections that are not spelled out in the code
- A human (or a chatbot) wouldn't spot out-of-bounds access that happens in a parent class destructor of an object whose lifetime was extended by a lambda capture...

Example 1: literal

```
int Array[10];
int foo(void) {
  return Array[11];
}
```

The clang compiler reports this by default:

> array index 11 is past the end of the array (that has type 'int[10]') [-Warray-bounds]

Example 1: literal

```
int Array[10];
int foo(void) {
  return Array[11];
}
```

The clang compiler reports this by default:

> array index 11 is past the end of the array (that has type 'int[10]') [-Warray-bounds]

And the static analyzer can also easily report this:

> Out of bound access to memory after the end of 'Array' [security.ArrayBound]
> Access of 'Array' at index 11, while it holds only 10 'int' elements

Example 2: following calls

```
int Array[10];
int twice(int x) { return x * 2; }
int foo(void) {
  return Array[twice(twice(3))];
}
```

The compiler warning doesn't report this, but the analyzer can follow function calls:

> Out of bound access to memory after the end of 'Array' [security.ArrayBound]
> Access of 'Array' at index 12, while it holds only 10 'int' elements

Example 2: following calls

```
int Array[10];
int twice(int x) { return x * 2; }
int foo(void) {
  return Array[twice(twice(3))];
}
```

The compiler warning doesn't report this, but the analyzer can follow function calls:

> Out of bound access to memory after the end of 'Array' [security.ArrayBound]
> Access of 'Array' at index 12, while it holds only 10 'int' elements

The analyzer can be configured to load definitions from different translation units (CTU = cross translation unit analysis mode).

Example 3: memory manipulation

```
int Array[10];
int foo(void) {
  int x = 9;
  x++;
  return Array[x];
}
```

Store (memory model) of the analyzer
after the initialization of `x`:
memory area of variable "`int x`" → concrete integer 9
after the increment operation:
memory area of variable "`int x`" → concrete integer 10

The analyzer tracks the contents of the memory as it evaluates the code step by step:

Out of bound access to memory after the end of 'Array' [security.ArrayBound]
Access of 'Array' at index 10, while it holds only 10 'int' elements

Notice that this is not main() – symbolic execution can start anywhere!

Example 4: multiple paths

```
int Array[10];
int foo(void) {
  int x = 9;
  if (!Array[x])
    x++;
  return Array[x];
}
```

Store (memory model) of the analyzer

after the initialization of `x`:

memory area of variable "`int x`" → concrete integer 9

after the increment operation on the branch that reaches it:

memory area of variable "`int x`" → concrete integer 10

`return Array[x];` is checked separately with each of the two states

The analyzer follows each execution path separately:

> Out of bound access to memory after the end of 'Array' [security.ArrayBound]
> Access of 'Array' at index 10, while it holds only 10 'int' elements

The path is also reported ("Taking true branch" note on the "if" statement).

Example 5: symbolic values

```
int Array[10];
int foo(int x) {
  if (x >= 10)
    return Array[x];
  return x;
}
```

Store (memory model) of the analyzer
during the whole analysis:
memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints (information recorded about symbols)
(at the beginning: nothing interesting)
on the path that enters the if:
$S_1 \in [10, \texttt{INT\_MAX}]$

The analyzer can follow this:

```
Out of bound access to memory after the end of 'Array' [security.ArrayBound]
Assuming 'x' is >= 10
Taking true branch
Access of 'Array' at an overflowing index, while it holds only 10 'int' elements
```

Example 6: separation between paths

```
int Array[10];
int foo(int x) {
  if (x >= 10)
    printf("too large!\n");
  return Array[x];
}
```

Store (memory model) of the analyzer

during the whole analysis:

memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints (information recorded about symbols)

(at the beginning: nothing interesting)

on the path that has entered the if (and stays separated):

$S_1 \in [10, \texttt{INT\_MAX}]$

Paths are not merged if their state (e.g. constraint set) differs:

```
Out of bound access to memory after the end of 'Array' [security.ArrayBound]
Assuming 'x' is >= 10
Taking true branch
Access of 'Array' at an overflowing index, while it holds only 10 'int' elements
```

Example 7: conservative logic

```
int Array[10];
int foo(int x) {
  return Array[x];
}
```

Store (memory model) of the analyzer
during the whole analysis:
memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints (information recorded about symbols)
during the whole analysis: nothing interesting

The analyzer doesn't report anything – it optimistically assumes that this function is only called in situations where the argument is in bounds.

Example 7: conservative logic

```
int Array[10];
int foo(int x) {
  return Array[x];
}
```

Store (memory model) of the analyzer
during the whole analysis:
memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints (information recorded about symbols)
during the whole analysis: nothing interesting

The analyzer doesn't report anything – it optimistically assumes that this function is only called in situations where the argument is in bounds.

It is often useful to perform the validation of arguments in the "outer" functions, where it is easy to report the errors. After this, the small subroutines can trust that their arguments are valid – so the analyzer shouldn't report them!

Example 7: conservative logic

```
int Array[10];
int foo(int x) {
  return Array[x];
}
```

Store (memory model) of the analyzer
during the whole analysis:
memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints (information recorded about symbols)
during the whole analysis: nothing interesting

The analyzer doesn't report anything – it optimistically assumes that this function is only called in situations where the argument is in bounds.

It is often useful to perform the validation of arguments in the "outer" functions, where it is easy to report the errors. After this, the small subroutines can trust that their arguments are valid – so the analyzer shouldn't report them!

This is not main() – the analyzer doesn't see the context!

Example 8: untrusted value

```
int Array[10];
int foo(void) {
  int x;
  scanf("%d", &x);
  return Array[x];
}
```

Store (memory model) of the analyzer
after the call to `scanf`:
memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints: nothing interesting

Untrusted values:
after the call to `scanf`: S_1 is tainted (untrusted)

If taint analysis (`optin.taint.GenericTaint`) is enabled, the analyzer reports this:

> Potential out of bound access to 'Array' with tainted index [security.ArrayBound]
> Access of 'Array' with a tainted index that may be negative or too large

When the analyzer sees that the value originates from an untrusted source, it can and will report that it **may be** out of bounds.

Example 9: checked value

```
int Array[10];
int foo(void) {
  int x;
  scanf("%d", &x);
  if (0 <= x && x < 10)
    return Array[x];
  return -1;
}
```

Store (memory model) of the analyzer
after the call to `scanf`:
memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints:
on the path that enters the `if`: $S_1 \in [0, 9]$

Untrusted values:
after the call to `scanf`: S_1 is tainted (untrusted)

The analyzer doesn't report this – the value is untrusted, but it cannot be out of bounds.

Example 10: recording assumptions

```
int Array[10];
int foo(int x) {
  return Array[x] + Array[x+10];
}
```

Store (memory model) of the analyzer
during the whole analysis:
memory area of variable "`int x`" $\rightarrow$ symbol S_1

Constraints:

after the first subscript operator: $S_1 \in [0, 9]$
$\Rightarrow$ the analyzer deduces that $(S_1 + 10) \in [10, 19]$

Out of bound access to memory after the end of 'Array' [security.ArrayBound]
Access of 'Array' at an overflowing index, while it holds only 10 'int' elements

... and the first subscript operator is marked with an explanatory note:

Assuming index is non-negative and less than 10, the number of 'int' elements in 'Array'

Example 11: dynamic memory

```
int *foo(void) {
  int *Array =
    (int*)calloc(10, sizeof(int));
  Array[11] = 123;
  return Array;
}
```

Store (memory model) of the analyzer
after the call to `calloc` and assignment:
memory area of variable "`int *Array`" $\rightarrow$ symbol S_1
(memory area pointed by $S_1 \rightarrow$ filled with zero)

Constraints: nothing interesting

Dynamic extent information:
after the call to `calloc`:
$S_1 \rightarrow$ concrete integer 40 (if `sizeof(int)` $= 4$)

`security.ArrayBound` is not limited to static arrays, it handles any memory region:

> Out of bound access to memory after the end of the heap area [security.ArrayBound]
> Access of the heap area at index 11, while it holds only 10 'int' elements

(Memory leak, use-after-free etc. is also reported by other checkers.)

Example 12: symbolic comparison

```
char *foo(int x, int y) {
  char *Buf =
    (char*)malloc(x);
  if (x < y)
    Buf[y] = 100;
  return Buf;
}
```

Store (memory model) of the analyzer

memory area of variable "`int x`" $\rightarrow$ symbol S_1

memory area of variable "`int y`" $\rightarrow$ symbol S_2

after the call to `malloc` and assignment:

memory area of variable "`int *Buf`" $\rightarrow$ symbol S_3

Constraints:

on the path that enters the `if`: $(S_1 < S_2) = 1$

Dynamic extent information:

after the call to `malloc`: $S_3 \rightarrow$ symbol S_1

Out of bound access to memory around the heap area [security.ArrayBound]
Access of the heap area at a negative or overflowing index

Example 13: symbolic comparison II

```
int *foo(int x, int y) {
  int *Buf =
    (int*)malloc(x * sizeof(int));
  if (x < y)
    Buf[y] = 100;
  return Buf;
}
```

Store (memory model) of the analyzer

memory area of variable "`int x`" $\rightarrow$ symbol S_1

memory area of variable "`int y`" $\rightarrow$ symbol S_2

after the call to `malloc` and assignment:

memory area of variable "`int *Buf`" $\rightarrow$ symbol S_3

Constraints:

on the path that enters the `if`: $(S_1 < S_2) = 1$ (=true)

Dynamic extent information:

after the call to `malloc`: $S_3 \rightarrow$ symbol $(4 \times S_1)$

TODO: As of now, this is not reported: the checker tests (byte offset < extent), which is $(4 \times S_2 < 4 \times S_1)$ and cannot recognize that this contradicts the constraint $(S_1 < S_2)$.

Comparing numbers

ERICSSON

As seen in the previous example, when `security.ArrayBound` performs a bounds check, it uses the same subroutine (`evalBinOp`) which evaluates a comparison (e.g. `x < y`) spelled in the source!

Comparing numbers

ERICSSON

As seen in the previous example, when `security.ArrayBound` performs a bounds check, it uses the same subroutine (`evalBinOp`) which evaluates a comparison (e.g. `x < y`) spelled in the source!

This is necessary because the operands may be symbolic and we may have relevant constraints.

Comparing numbers is simple...

Comparing numbers

ERICSSON

As seen in the previous example, when `security.ArrayBound` performs a bounds check, it uses the same subroutine (`evalBinOp`) which evaluates a comparison (e.g. `x < y`) spelled in the source!

This is necessary because the operands may be symbolic and we may have relevant constraints.

Comparing numbers ~~is simple~~ can trigger type conversions!

Comparing numbers

ERICSSON

As seen in the previous example, when `security.ArrayBound` performs a bounds check, it uses the same subroutine (`evalBinOp`) which evaluates a comparison (e.g. `x < y`) spelled in the source!

This is necessary because the operands may be symbolic and we may have relevant constraints.

Comparing numbers ~~is simple~~ can trigger type conversions!

In C/C++ when the operands of an arithmetic operator have different integer types, they are implicitly converted to the larger type, or if the sizes are equal (e.g. `int` vs `unsigned int`), then to the unsigned type. (This is just a simplified picture – there is also integer promotion etc.)

Comparing numbers

ERICSSON

As seen in the previous example, when `security.ArrayBound` performs a bounds check, it uses the same subroutine (`evalBinOp`) which evaluates a comparison (e.g. `x < y`) spelled in the source!

This is necessary because the operands may be symbolic and we may have relevant constraints.

Comparing numbers ~~is simple~~ can trigger type conversions!

In C/C++ when the operands of an arithmetic operator have different integer types, they are implicitly converted to the larger type, or if the sizes are equal (e.g. `int` vs `unsigned int`), then to the unsigned type. (This is just a simplified picture – there is also integer promotion etc.)

In particular if one operand is of type `size_t`, the comparison is unsigned!

Comparing numbers

ERICSSON

As seen in the previous example, when `security.ArrayBound` performs a bounds check, it uses the same subroutine (`evalBinOp`) which evaluates a comparison (e.g. `x < y`) spelled in the source!

This is necessary because the operands may be symbolic and we may have relevant constraints.

Comparing numbers ~~is simple~~ can trigger type conversions!

In C/C++ when the operands of an arithmetic operator have different integer types, they are implicitly converted to the larger type, or if the sizes are equal (e.g. `int` vs `unsigned int`), then to the unsigned type. (This is just a simplified picture — there is also integer promotion etc.)

In particular if one operand is of type `size_t`, the comparison is unsigned!

For example, `evalBinOp` evaluates `-1 < sizeof(int)` to false!

An old false positive

```
char foo(void *p) {
  return ((char *)p)[-1];
}
```

Store (memory model) of the analyzer

memory area of variable "`void *p`" $\rightarrow$ symbol S_1

Constraints: nothing interesting

Dynamic extent information: $S_1 \rightarrow$ symbol S_2

as S_1 is unknown, the engine introduces a symbol S_2 (of type `size_t`) to represent its extent

Before I fixed this bug, the checker was reporting an **overflow** error:

> Out of bound access to memory after the end of the region [alpha.security.ArrayBoundV2]
> Access of 'char' element in the region at index -1

Testing (byte offset < extent) meant $(-1 < S_2)$, but -1 was converted to `SIZE_MAX` and the analyzer concluded that the completely unknown extent cannot be greater than `SIZE_MAX`...

An old false positive

```
char foo(void *p) {
  return ((char *)p)[-1];
}
```

Store (memory model) of the analyzer

memory area of variable "`void *p`" $\rightarrow$ symbol S_1

Constraints: nothing interesting

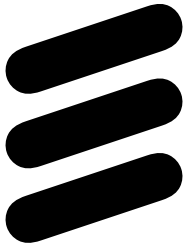
Dynamic extent information: $S_1 \rightarrow$ symbol S_2

as S_1 is unknown, the engine introduces a symbol S_2 (of type `size_t`) to represent its extent

Before I fixed this bug, the checker was reporting an **overflow** error:

> Out of bound access to memory after the end of the region [alpha.security.ArrayBoundV2]
> Access of 'char' element in the region at index -1

Testing (byte offset < extent) meant $(-1 < S_2)$, but -1 was converted to `SIZE_MAX` and the analyzer concluded that the completely unknown extent cannot be greater than `SIZE_MAX`...
This was causing hundreds of false positives on our reference set of open source projects...



ERICSSON

donat.nagy@ericsson.com