C++ Safe Buffers

danakj@chromium.org (she/her)

Chrome Security Team at Google

RFC: C++ Buffer Hardening

- RFC posted on the LLVM discourse
  - https://discourse.llvm.org/t/rfc-c-buffer-hardening/65734

Two parts

1. C++ library hardening = **libc++ changes**
2. C++ safe buffers = **compiler changes**

C++ library hardening

- See the next talk!
- Work from this assumption:

Standard library **containers**

terminate the program if you

go **out of bounds.**

The proposal starts with a desire to have bounds checks in the standard library.

The talk after this one will be about this idea.

The comments on the proposal get a little bit spicy, but there were some interesting things, which I will come back to if I have time.

GCC's Fortification: Compiler hardening of libc

https://developers.redhat.com/articles/2022/09/17/gccs-new-fortification-level

- The compiler provides a builtin `__builtin_dynamic_object_size`
- Evaluates to an expression that is the exact size of an object.
- Using what would appear to be dataflow analysis (it considers branches).

When the resulting expression depends on runtime data:

- The **compiler inserts bounds checks** on libc calls like memset, passing this expression as the bounds.

One thing brought up in the comments was Fortification.

Fortification is a static analysis happening in the GCC compiler. Clang also supports the builtins that have been added for it, though I don't know that it uses them in the same way.

When GCC sees a call to a libc function like memset(), it performs static analysis to determine if the call will stay in bounds.

But it can't do that for a dynamic buffer like a vector. So the compiler determines that it can't tell at compile time, and inserts a bounds check before the memset() occurs.

The bounds check is against a compiler-generated expression that gives the actual size of the buffer at runtime (if possible).

GCC's Fortification in action

When memset(buf, 0, 22) occurs:

- Compiler sees that buf may only be 21 characters.
- Needs runtime data to know the bounds.

So the compiler inserts a bounds check:

CHECK(**22 < cond ? 42 : 21**)

```
char s[21];

void f(bool cond)
{
    char *buf = s;
    if (cond)
        buf = malloc(42);
    memset(buf, 0, 22);
}

int main()
{
    f(false);
}
```

This example is slightly simplified from the link on the previous slide.

The function f() will memset into a buffer, which is chosen at runtime. Either
- The static 21 character array `s`, or
- The malloced 42 character array.

The memset() is for 22 characters, so it will write out of bounds if the condition is false, and it writes to the static array.

With fortification, the compiler performs a dataflow static analysis (via the builtin we mentioned) to determine if the size of `buf` is known at compile time when memset() is called.
It will see that it is not, it depends on the value of `cond`.
Given that, the compiler will change the call to memset() to perform bounds checking, to verify that the write of 22 bytes will stay in bounds.

The compiler is able to do this without a container like span by generating an expression that returns the size of the buffer, if it is able to determine it.

The limitations are that if this malloc decision happened elsewhere, and the pointer was passed into f(), the builtin would be unable to figure out its size and just returns the max value for size_t. That's where std::span, with its paired length, can do better.

Libc++ Implements the C++ Standard

The authors stated that:

- Willing to change the ABI (behind a compile-time flag).
- **Not willing** to change the API.
- Will add checks only where Undefined Behaviour is the specification.

Quote:

"Libc++ implements the C++ Standard, and that is a crucial point that we do not plan on changing. None of these additional checks would change the API".

The authors were asked to clarify a bunch of things, and some highlights were:

- That they are willing to break ABI, but behind a flag. This is kind of a new take from C++ standard-adjacent land afaik.
- That they will not ship a library that differs from the standard.
- To that end, they will insert checks into places the spec states as Undefined Behaviour. This becomes "Just Fine" as an implementer since the spec has told you that you can do whatever you want in this space.

C++ safe buffers

- Compiler warning: -Wunsafe-buffer-usage
- Bans pointer **arithmetic**
- Will also ban std functions on pointers that are proxies for pointer arithmetic
  - std::next, std::prev, std::advance, ...

Clear target audience

- Big Tech companies trying to stop shipping CVEs to users.
- Huge teaching opportunity for folks learning to write C++.

The compiler changes for Safe buffers are a simple idea at a high level. A new compiler warning that fires when you do pointer arithmetic (but not pointer dereferencing).

Of course, this is C++, which means even simple ideas require very complex solutions to handle both the language and the pre-existing ecosystem.

The target audience does not appear to include hobbyists, students, or C++ devs outside of Big Tech.
But I see this as a huge teaching opportunity for folks learning to write C++, in order to steer them into more predictable behaviour, and help them avoid spending years of their lives debugging UB and memory bugs.

Incremental Application

- Critical for any C++ memory safety tool

Files and Scopes

- Pragma **unsafe_buffer_usage** allows pointer arithmetic
- Pragma **only_safe_buffers** disallows pointer arithmetic

Think Globally, Act Locally

- **[[unsafe_buffer_usage]]** function attribute indidates buffer overflow is possible within.

Because the target of this proposal is Big Tech, and Big Tech has a LOT of C++ code, it's critical that you can either:

- Mechanically apply the change to all of your code at once, or
- Incrementally opt parts of your codebase into the new thing.

Since this requires rewriting code from native pointers to using containers or functions that encapsulate the pointers, it can't be applied mechanically.

So the authors have provided ways to incrementally apply the safe-buffers restrictions to a codebase, and grow adoption in a viral-but-explicit way.

Secondly, incremental adoption implies the necessary escape hatch to use pointer arithmetic inside an encapsulated API, such as inside std::span.

The first tool provided is pragmas to allow or disallow pointer arithmetic.

- The first one, with "unsafe" in its name, is the moral equivalent of the Rust unsafe keyword, but just for pointer arithmetic.
- The second allows smaller parts of code to opt into being "safe" wrt buffer overflows.

The second tool is the [[unsafe_buffer_usage]] function attribute. It serves two purposes:

- It allows the static analysis of the warning to be done locally within each function, which is critical for the performance of the check.

- Pointers are disallowed from being passed from opted-in-safe code to a function marked with the attribute.
- This allows the static analysis to escape from individual functions and be applied virally across a codebase.

The RFC authors expect the function attribute to be used by tools to suggest a safer alternative function.

Suggesting Fixes

- A key pillar of this proposal.
- The majority of the code in the implementation.
- Will allow the warning to be applied to **far more code** than it would otherwise.

Not everything can be determined by the compiler, may be some fill-in-the-blanks:

- `std::span<int> myVar(buffer, #size#)`

One highly emphasized point in this proposal is the ability for the compiler to generate suggested fixes.

These fixes are the most involved suggested fixes we have seen from a C++ compiler to date (I think). This is where a lot of the novel technology lies, and it will enable the warnings to be applied much quicker to large existing codebases.

They fixit suggestions must involve changing types in other locations in the code, and choosing what type would be appropriate based on the usage of it.
Then it must suggest migrations to the new type's API as needed.

The authors recognize they can't provide perfect suggestions always, some things will need to be filled in by the developer with non-local knowledge.

How does this look: Writing APIs

```
// Safe function.
int index(std::span<int> array, size_t index) {
    return array[index];
}

// Scary function.
int [[unsafe_buffer_usage]] index_unchecked(int* array, size_t index) {
    return array[index];
}
```

Let's take a look at some code to get a feel for this.

Here we have a little function called `index_unchecked()` that will happily walk off the end (or beginning) of your array.

But we don't want most users to use that. We want C++ developers to pay for only what they need, and they don't need to pay in CVEs for their array access unless they are doing some serious optimization work.

So the scary function is annotated with the [[unsafe_buffer_usage]] attribute. This will prevent it from being used in code that opted into safety from pointer-arithmetic bugs.

The function above, called `index()`, our glorious default path, is no different from the scary function today. However _if_ the function can rely on std::span to abort when an OOB access happens, then the developer knows this function can not invoke OOB read/write. So they don't annotate this function.

How does this look: Using APIs

```
#pragma push(only_safe_buffers)
int main() {
   int array[] = {1, 2, 3, 4, 5};

   std::cout << index(std::span(array), 2) << '\n';  // No problem.

}
#pragma pop
```

Here we have a call to the friendly index() function.

- We construct a span, which knows the length of the array.
- The index() function works on the span which performs bounds checks.
- No OOB bug can occur.

How does this look: Using APIs

```
#pragma push(only_safe_buffers)
int main() {
   int array[] = {1, 2, 3, 4, 5};

   std::cout << index(std::span(array), 2) << '\n';  // No problem.

   std::cout << index_unchecked(array, 2) << '\n';   // Compiler warning.


}
#pragma pop
```

Then we pass the same array as a pointer to the index_unchecked() function.

This function was annotated as [[unsafe_buffer_usage]] since it does pointer arithmetic/indexing.
But the caller here has opted into not having OOB bugs. So it's not allowed to call the [[unsafe_buffer_usage]] function, and the compiler generates a warning.

How does this look: Using APIs

```
#pragma push(only_safe_buffers)
int main() {
   int array[] = {1, 2, 3, 4, 5};

   std::cout << index(std::span(array), 2) << '\n';  // No problem.

   std::cout << index_unchecked(array, 2) << '\n';   // Compiler warning.

#pragma push(unsafe_buffer_usage)
   std::cout << index_unchecked(array, 2) << '\n';   // OK.
#pragma pop
}
#pragma pop
```

But maybe we have bounds-checked ourselves and we're doing some intense for looping, and we really don't want the bounds checks. We want to opt into OOB bugs, and have our reviewers suffer to verifying they don't actually happen.

So we mark the callee as allowed to perform pointer arithmetic, through the unsafe_buffer_usage pragma. The compiler accepts this, your code reviewer may not.

The Good and The Bad

```
#pragma push(only_safe_buffers)
int main() {
   int array[] = {1, 2, 3, 4, 5};

#pragma push(unsafe_buffer_usage)
   std::cout << index_unchecked(array, 2) << '\n';   // OK.
#pragma pop

}
#pragma pop
```

Good things about the approach:

- You can limit pointer arithmetic transitively, which is critical.
- You can choose to opt in to not having OOB bugs. For your whole codebase, or just specific files, or blocks of code.
- You can escape to pointer arithmetic in a way that is easily observable in code review, and can be easily found and banned by systems like PRESUBMIT scripts or git commit hooks.

Bad things:

- Because this is done outside of the language itself, it has to rely on pragmas.
- While they can be stuck behind macros, they are super noisy and nothing like an `unsafe` block for readability. I find it all rather noisy.

[>>]

- Pragmas require a full line, making it difficult or awkward to use inside an expression, such as allowing a function call that is used as a function argument.
- It's the awkwardness of #ifdefs all over again.

The Good and The Bad

```
#pragma push(only_safe_buffers)
int main() {
   int array[] = {1, 2, 3, 4, 5};

#pragma push(unsafe_buffer_usage)
   std::cout << index_unchecked(array, 2) << '\n';   // OK.
#pragma pop

}
#pragma pop
```

```
use_the_integer(
#pragma push(unsafe_buffer_usage)
  index_unchecked(array, 2)
#pragma pop
);
```

- Pragmas require a full line, making it difficult or awkward to use inside an expression, such as allowing a function call that is used as a function argument.
- It's the awkwardness of #ifdefs all over again.

Conclusion

The proposal talks about **pointer arithmetic**, but I think there's a slightly simpler way to describe it.

Provide a mechanism to **encapsulate the use of native arrays** in C++,
in order to **prevent Out-of-Bounds accesses**.

Full proposal in Review: https://reviews.llvm.org/D136811

So to sum it all up, I would say this proposal is to

Provide a mechanism to encapsulate the use of native arrays in C++, in order to prevent Out-of-Bounds accesses.

C++ simply doesn't give us the tools to write these types of mechanisms on a per-expression basis, which makes it all a bit awkward.
I really wish that we could see the language provide more extensibility in this direction in the future.

But working with the language we have, I think it's a really great proposal, and I look forward to it landing in Clang.

The proposal text is currently being reviewed, and if you have thoughts, wishes, or interest go check it out.