CASE 09

A Reference to a Ghost

CLASSIFICATION: DANGLING VIEW (UB) | STATUS: CLOSED | WEAPON: a dangling borrow

A function that returned the right text, and a window onto where that text used to sit. By the time the caller looked through the window, the text had gone home — and whatever ran next had moved into the empty room.

1 THE CRIME SCENE

A logging helper turns a status code into a readable label and hands it back as a `std::string_view` — the lightweight, copy-free string type everyone reaches for. Most of the time the label printed correctly. Sometimes it printed blanks, or a fragment of some unrelated string, or a smear of bytes; once in a while it took the process down. The corruption tracked with nothing the team could pin — it came and went with unrelated code nearby, the kind of nondeterminism that whispers "race" or "uninitialised memory." The compiler said nothing, at any warning level. The function was three lines long and looked entirely correct.

2 THE BODY

The helper and its caller.

```
1  std::string_view status_label(int code) {
2      std::string label = "HTTP status " + std::to_string(code);  // a local string
3      return label;                                               // return a view into it
4  }
5
6  int main() {
7      std::string_view s = status_label(404);
8      std::cout << "label = " << s << "\n";                       // read through the view
9  }
```

It compiles without a single diagnostic: `-Wall -Wextra` say nothing, `-Wreturn-local-addr` says nothing, and even the static analyzer `-fanalyzer` stays silent. We run it.

FORENSIC READOUT

```
$ g++ -std=c++20 -Wall -Wextra -O2 label.cpp -o label
$ ./label
label = <- 15 bytes of dead stack (here, zeros)
```

The label is gone. The function plainly built the right string — yet by the time `main` reads it, the text has been replaced by whatever now occupies that memory. On this run, zeros; on another, garbage; on a third, a crash.

3 THE SUSPECTS

Three explanations. The first two are misdirection.

Suspect 1 — `std::to_string` and the concatenation. Perhaps the label was built wrong — a bad conversion, a botched join. The simplest theory, and testable: print the string *before* the function returns and see what it holds.

Suspect 2 — `string_view` over-reading. A view does not null-terminate; perhaps the printer runs off the end of the data and trails garbage. A genuine `string_view` hazard, and worth ruling out.

Suspect 3 — `return label;`. The function hands back a view into `label` — a string that is destroyed the instant the function returns. The killer.

4 THE FORENSIC LAB

First, Suspect 1. We print the string inside the function, while it is still alive.

FORENSIC READOUT

```
$ ./inside # the same builder, printing the string before it returns
  inside, while alive: HTTP status 404
```

Built perfectly. The bytes are correct as long as `label` exists, so the conversion is innocent — the text is right, then it is gone. Now AddressSanitizer, on the original.

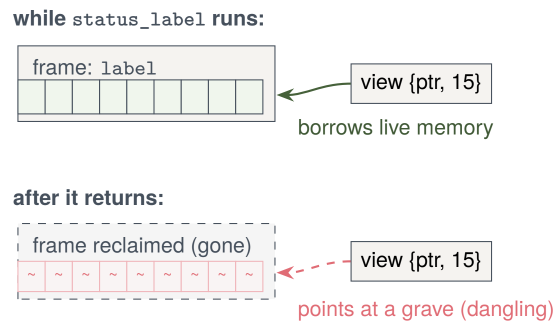
FORENSIC READOUT

```
$ g++ -std=c++20 -O1 -g -fsanitize=address label.cpp && ./a.out
==ERROR: AddressSanitizer: stack-use-after-return
READ of size 15 at 0x7f..080 thread T0
    #2 operator<<(ostream&, string_view) /usr/include/c++/13/string_view:763
Address 0x7f..080 is located in stack of thread T0 at offset 128 in frame
    #0 status_label(int) label.cpp:6
SUMMARY: AddressSanitizer: stack-use-after-return in fwrite
```

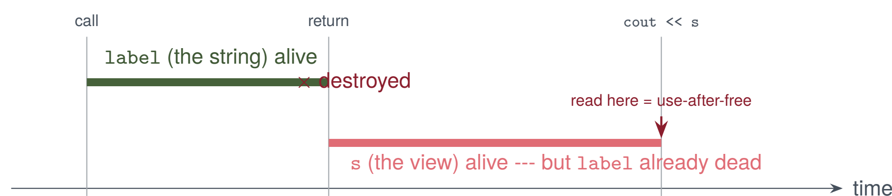
The verdict is exact. The read of 15 bytes — precisely the length of `"HTTP status 404"` — lands in `status_label`'s stack frame, a frame that ceased to exist when the function returned. That "READ of size 15" also clears Suspect 2: the view's length is intact and the read is perfectly bounded; it is not running off the end of live data, it is reading dead memory. The string was not built wrong and not over-read. It was *outlived*. Suspect 3 stands convicted — and the silence of every static tool, `-fanalyzer` included, is explained: the dangling pointer is wrapped inside a `string_view` object, and tracking it back to a dead local across the return is past what they attempt.

5 THE CONVICTION

Here is the mechanism. A `std::string_view` is not a string; it is a borrowed window — a pointer and a length, owning nothing. `status_label` builds `label`, whose 15 characters live in a buffer inside the string object, on the function's stack frame. `return label;` constructs a view — { pointer into that buffer, length 15 } — and copies *it* out to the caller. Then the frame unwinds: `label` is destroyed and its storage reclaimed. The pointer the caller now holds aims at memory that is no longer anyone's.



The window is unchanged; only the room behind it emptied. When `main` prints the view, it dereferences that pointer and reads 15 bytes of whatever moved in — here, the zeros left by the stream machinery that ran next. The lifetimes never overlapped the way the code assumed.



The killer line is `return label;`: it returns a non-owning view of an object whose lifetime ends at that very `return`. Reading through the view afterward is access outside the object's lifetime — undefined behavior, and on this run it cashed out as a silent blank.

Why didn't C++ warn me? — the question this book exists to answer.

- **Why does the language allow it?** Because `string_view` is, by design, a non-owning borrow — a pointer plus a length. The language lets you build one from any character source and lets a function return one, because a view is meant to be a cheap window onto data that lives elsewhere. It trusts you, exactly as it trusts a raw pointer or reference, to keep that data alive for as long as the view is used.
- **Why is it designed this way?** For speed. A view passes and slices strings with no copy and no allocation; that is its entire reason to exist. Make it own its data and it becomes `std::string`. Non-ownership is the feature, and the cost of the feature is that the borrower must not outlive the lender — a contract the type cannot enforce without becoming something else.
- **Why can't the compiler catch it?** Because proving that a pointer escapes its object's lifetime is, across function boundaries, undecidable; the pointer can be stored in a view, returned, and carried through arbitrary layers. A few narrow shapes are caught (`-Wreturn-local-addr` for a bare returned reference), but here the pointer hides inside a class, and the data-flow from local string to view to caller is past what the warnings or even `-fanalyzer` pursue. No diagnostic is required. Only a runtime tool that tags reclaimed memory — AddressSanitizer — catches the read.

6 THE LESSON

> **The rule, read from an empty room:**
> A `std::string_view` is a borrowed window, not a copy — a pointer and a length. It must never outlive the object it borrows. Return a view into a local, or bind one to a temporary, and you keep an address after the tenant has moved out. What it shows you is whatever moved in.

How the detective keeps a borrow honest:

- **Return owning types.** A function that *builds* its result should return `std::string` (which owns its data), not a `string_view`. The move is cheap; the safety is absolute.
- **Use `string_view` for parameters, not for outliving results.** As an argument it borrows the caller's live string — exactly its purpose. As a stored or returned value it must point at something that will outlive it.
- **Never bind a view to a temporary.** `std::string_view sv = a + b;` dangles at the semicolon; keep the owner alive first — `std::string s = a + b;` then view `s`.
- **The rule generalizes.** `std::span`, iterators, references, raw pointers — every non-owning handle requires its owner to outlive it. Same grave, different headstone.
- **Catch it with AddressSanitizer.** It flags stack-use-after-return and heap-use-after-free precisely; static checks, `-fanalyzer` included, miss views through class wrappers. Run your tests under ASan.

Crime reconstruction. We change the return type from `std::string_view` to `std::string` — the function now owns what it builds — and nothing else.

```
FORENSIC READOUT
$ g++ -std=c++20 -Wall -Wextra -O2 label_fixed.cpp && ./a.out
label = HTTP status 404
$ g++ -O1 -g -fsanitize=address label_fixed.cpp && ./a.out # ASan: clean
label = HTTP status 404
```

The caller now holds its own copy, alive for as long as it is needed; the sanitizer is silent because there is no dead memory to read. One owning return in place of one borrowed view.

`string_view` never promised to keep the text — only to remember where it sat. You let the text go home, kept the address, and read it aloud to the caller. The window was honest. There was simply nothing behind it but whatever moved in.

CASE 09 --- CLOSED