Why does inadvertently passing a std::string instead of a char const* to a variadic function crash on x86-32 but not x86-64?

devblogs.microsoft.com/oldnewthing/20250110-00

January 10, 2025

A customer tracked down a crash to inadvertently passing a `std::string` instead of a `char const*` to a variadic function.

```
extern void Log(const char* format, ...);

std::string name("apple");

Log("%z: %d", name, 42);
// oops, should be Log("%z: %d", name.c_str(), 42);
```

The `Log` function takes a format string similar to `printf`, but with slightly different insertions. In this case, it uses `%z` to represent a null-terminated string. (Other formats include `%v` for `std::string_view`.)

As noted in the comment, the `name` parameter should have been passed as `name.c_str()` so that a `const char*` is passed, which is what `%z` expects.

The customer found something unusual in the failure profile: Only the x86-32 systems were crashing. The x86-64 systems were working fine. What's going on?

The C++ language specifies that implementations are permitted but not required to support passing class types to satisfy a classic C-style `...` argument.[1] The clang and gcc compilers do not allow it, but msvc does. What is it about x86-32 that makes the code crash, and what is it about x86-64 that makes it work?

The calling convention for x86-32 passes parameters on the stack by value, so there is a `std::string` on the stack after the format string. On the other hand, the calling convention for x86-64 passes large structures by address. The 64-bit `std::string` is 32 bytes in msvc, so it will be passed by address.

The other piece of information you need to know to understand what's going on is the internal layout of a `std::string` in msvc, which we studied some time ago.

Now let's put things together.

In the case above, the string `"apple"` fits inside the 16-character short string buffer, so in the 32-bit case, the values on the stack are the characters of the string, followed by the size and capacity.

	42				
`std::string`	capacity				
	size				
	?	?	?	?	
	?	?	?	?	
	`e`	`\0`	?	?	
	`a`	`p`	`p`	`l`	
	pointer				→ `"%z: %d"`

Now we can see why the code crashes on x86-32: The `Log` function interprets the stack like this:

	42				
	capacity				
	size				
	?	?	?	?	
	?	?	?	?	
`int`	`e`	`\0`	?	?	
`const char*`	`a`	`p`	`p`	`l`	
`const char*`	pointer				→ `"%z: %d"`

The first four characters of the string are misinterpreted as a pointer, resulting in an invalid pointer that crashes when the `Log` function tries to treat it as a pointer to a string. (It crashes before getting around to the garbage integer parameter coming next.)

On the other hand, on x86-64, the `std::string` is passed by address, so you get this:

`std::string`	capacity	
	size	
	? ? ? ? ? ? ? ?	
	a p p l e \0 ? ?	
	⋮	
	(home space 3)	`r9` = ?
	(home space 2)	`r8` = 42
	(home space 1)	`rdx` ⟶ `std::string` on stack
	(home space 0)	`rcx` ⟶ `"%z: %d"`

In the x86-64 case, the value in `rdx` is treated as a pointer to a null-terminated string, and how about that, it is! It points to the `"apple"` stored in the short string buffer. Next is the value in `r8`, which is treated as the integer parameter, and that's what it is too!

So that's why the problem doesn't appear on x86-64: The short string optimization, the policy of passing large structures by address, and the specific layout of a `std::string` in msvc collectively mean that a pointer to a short `std::string` also works as a pointer to a null-terminated string because the null-terminated string is at the start of the `std::string`.

The customer explained that in all the cases they know about, the name is a relatively short string, just 5 to 10 characters. This means that it will always fit in the short string buffer, and the above diagrams will apply.

But what if the name doesn't fit in the short string buffer?

In the x86-32 case, we get this:

	42
`std::string`	capacity
	size
	? ? ? ?
	? ? ? ?

	? ? ? ?	
	pointer	→ `"relatively long name"`
	pointer	→ `"%z: %d"`

If we relabel the parameters by how the `Log` function sees them:

	42	
	capacity	
	size	
	? ? ? ?	
	? ? ? ?	
`int`	? ? ? ?	
`const char*`	pointer	→ `"relatively long name"`
`const char*`	pointer	→ `"%z: %d"`

This time, the `Log` function reads the first four bytes of the `std::string`, and they are a pointer to the string data, so the string gets logged succssfully. However, the integer value is not the expected value of 42, but rather four garbage bytes from the unused portion of the small string buffer inside the `std::string`.

If you have a large string on x86-64, you get this:

`std::string`	capacity	
	size	
	? ? ? ? ? ? ? ?	
	pointer	→ `"relatively long name"`
	⋮	
	(home space 3)	`r9` = ?
	(home space 2)	`r8` = 42
	(home space 1)	`rdx` → `std::string` on stack

(home space 0) `rcx` ⟶ `"%z: %d"`

This time, the `Log` function interprets `rdx` as a pointer to a null-terminated string, and it ends up interpreting the bytes of the pointer at the start of the `std::string` as characters to be logged, so you will get garbage. If you assume that even a long name won't be 4 billion characters long, then you know that the garbage will find a null byte within 20 characters, because it'll eventually run into the zeroes that form the most significant bytes of the `size`. The integer value in `r8` is interpreted as intended.

Okay, so here's what we have:

	x86-32	**x86-64**
Short name	(crash)	`"name: value"`
Long name	`"name: garbage"`	`"garbage: value"`

The customer used these results to prioritize how urgently a fix is needed for the two platforms. They decided that the x86-32 version required an immediate fix, but the x86-64 version could wait, since the only consequence is garbage in a log file. They'll just make a note in their debugging documents that the name in the 64-bit log file is not trustworthy for this particular version of their program.

[1] Not to be confused with a C++-style template parameter pack.