WG14 N1080
Sept 27, 2004

Limited `size_t`

Randy Meyers

Introduction

During committee discussion at the Sydney meeting, a request was made to limit `size_t` for the new functions. Basically, it was observed that a common programming error is to accidentally calculate a negative size value that when placed in a `size_t` (an unsigned type) looks like a very big positive number. When such values are used for source sizes, far too much data is accessed. When such values are used for destination sizes, the destination appears far larger than it really is, and functions might write past the end of the object. This last case can be insidious since the expected protection of using the new secure functions has been defeated, but until the function might not fail or cause any sort of any invalid memory reference until confronted with atypically large data. This can also be yet another source for buffer overrun security attacks despite using the secure library.

Several committee members spoke in favor of limiting `size_t` for the functions in the secure library. I don't recall any opposition. But, the idea surfaced in committee discussion without any written proposal to back it up. I agreed to look into this. Clearly, the idea discussed at the meeting was too half-baked to be folded into a draft of the Security TR. So, I produced this proposal to allow for proper committee review.

Proposal in a nutshell

A new typedef is introduced named (for the sake of presentation) `size_t_s`. This typedef has exactly the same type as `size_t`. However, when `size_t_s` is used to define a function parameter, it is a warning to the programmer that only a limited range of the possible values of a `size_t` is allowed. Furthermore, functions that have `size_t_s` parameters have a responsibility to check that the values of those parameters are reasonable. If those parameters have unreasonable values, the function will treat that as an error condition, not perform any further work, and return some sort of error indicator to the caller. (The details of returning the error indicator to the caller are likely to vary from function to function.)

The rest of this paper discusses the details of this proposal.

The typedef

No good catchy name for the limited typedef proposal occurred to me. Two possibilities are:

- `size_t_s,` which has the advantage of following the naming pattern established for functions in the Secure TR

- `size_s`, which has the advantage of being a shorter name and more or less fitting the pattern, but has the disadvantage that it is only one character different from `size_t`.

For the purpose of this document, `size_t_s` is used. If the committee likes this proposal, suggestions for a better name are welcome.

Many headers in the library define the `size_t` typedef. Likewise, `size_t_s` will be defined in any header that uses it, and in <stddef.h>, because that header would be an expected place for the typedef. Thus, the following headers:

- `<stddef.h>`
- `<stdio.h>`
- `<stdlib.h>`
- `<string.h>`
- `<time.h>`
- `<wchar.h>`

will contain the additional declaration:

```
typedef size_t size_t_s;
```

That declaration will have to be properly protected to avoid redeclarations in case multiple headers defining the typedef are included, just as the declarations of `size_t` in those headers is protected for the same situation. The declaration will also have to honor the `__STDC_WANT_SECURE_LIB__` macro.

Note that because `size_t_s` is the same type as `size_t`, the two types are not really distinct and can be punned in various ways, and that there is no issue with binary compatibility between the two types. This is appropriate since `size_t_s` is just `size_t` with additional conceptual rules for the programmer and the library implementer. This type equivalence makes it proper for the `sizeof` operator to be used to produce the value for a `size_t_s` function argument.

Reasonable Values

The set of reasonable values for `size_t_s` needs to meet two slightly opposing requirements:

1. It needs to be small enough that the implementation can identify suspect values indicative of programming errors.
2. It needs to be large enough to be useful to programmers under most conditions.

In previous committee discussion, the proposal was the maximum value of `size_t_s` should be `SIZE_MAX/2`. (`SIZE_MAX`, defined by C99 7.18.3, is the maximum value of a `size_t`.)

That value does eliminate the "negative" values that might be in a `size_t`, but might still be bigger than desirable. Consider an implementation that only supports 32K sized objects, but uses a 32-bit `int` as `size_t`. Such in implementation would probably want to reject any `size_t_s` larger than 32K, not 2Gig.

Therefore, a more reasonable definition of the limit on `size_t_s` is the smaller of (`SIZE_MAX/2`) or (the maximum size of an object supported by the supported by the implementation).

Some consequences of this definition of the limit:

If an implementation allows as large of objects as `size_t` can represent, then `size_t_s` has the same limit originally proposed, which merely disallows the "negative" values for a size.

If an implementation has a smaller limit on the size of objects than `SIZE_MAX`, then there are two cases:

1. The limit on object sizes is smaller than (`SIZE_MAX/2`). In which case, the programmer can represent the size of any object in `size_t_s`.
2. The limit on object sizes is greater than (`SIZE_MAX/2`). In which case, the programmer can represent the size of any object in `size_t_s` except for those values that might appear "negative."

In all cases, the "negative" appearing values are disallowed.

In all cases, the programmer can represent either the size of any object, or objects up to `SIZE_MAX/2` in size.

Note that hosted implementations must support objects of at least 64K-1 in size, which implies that the smallest `size_t` is a 16-bit `unsigned short`. Thus, the tightest restriction on a hosted environment is 32K-1. That is not an unreasonable limit for minimalist systems.

More typical workstation and PC environments will have limits of 2G-1.

Freestanding environments sometimes have very tiny objects: 256 byte objects and `size_t` is an unsigned char. Such environments may find any limit on the range of `size_t_s` burdensome. However, such environments are not required to implement a library, and the Security TR is not really aimed at them. I would propose a footnote in the TR stating that freestanding environments choosing the implement the TR may ignore the limit on `size_t_s`.

Knowing the limit

Programmers will want to know the limit on `size_t_s` so that they can enforce the limit in their own functions (and perhaps to verify that the limit is big enough for their purposes).

The limit depends upon implementation-defined numbers, so it is reasonable that the limit be provided by the implementation.

The macro `SIZE_MAX_S` should be defined in `<stdint.h>` as this limit. The definition should be conditional on `__STDC_WANT_SECURE_LIB__`.

Note that `<stdint.h>` is the definition point for `SIZE_MAX`, the limit macro for `size_t`.

It might also be useful to have a function defined in <stdlib.h> that has the definition:

```
bool issize_t_s(size_t_s value) {return value <= SIZE_MAX;}
```

Checking `size_t_s` parameters

Functions in the Secure Library that take `size_t` parameters will be changed to take `size_t_s` parameters. The specification of the function will be changed to require checking the limit of the `size_t_s` parameters.

The edit will be very similar to the edits made to N1078 (the latest draft of the Security TR) to check for and handle null pointers.

For example, the prototype for the memcpy_s function would become:

```
errno_t memcpy_s(void * restrict s1, size_t_s s1max,
                 const void * restrict s2, size_t_s n);
```

Paragraph 2 of the function's description would be changed to:

> If `s1` or `s2` is a null pointer or if `s1max` or `n` is greater than `SIZE_MAX_S`, then no copying is performed.

The "Returns" section would be changed to:

> The `memcpy_s` function returns zero if `n` is less than or equal to `s1max` and `s1` and `s2` are not null pointers and `s1max` and `n` are less than or equal to `SIZE_MAX_S`. Otherwise, ERANGE is returned.

However, these wording changes are related to the issue around the edits in N1078 regarding the null pointer handling. If the committee wishes to keep the approach in

N1078 of stating explicitly the behavior of functions with null pointers (or in the general case, an invalid argument of some kind), then the edits for checking the range of `size_t_s` parameters are just more of the same.

If the wording for null pointer parameters in N1078 changes, a similar approach would be required for range testing `size_t_s` parameters.

In Conclusion

I believe that this proposal fleshes out the direction discussed at the Sydney meeting. I believe that it does hang together, and that the edits to the TR are not burdensome.

The biggest question is whether the committee thinks that this work is worth doing.