m4 Dangling Pointer Bug

John Brzustowski

January 18, 2006

(jump to "Current problematic behaviour" for a quick example)

1 Overview

It appears that the GNU documentation for m4 does not specify how m4 should behave when a macro whose arguments are being collected is redefined or deleted. In the latter case, there is a dangling pointer bug. Below are details of the misbehaviour, and the rationale and implementation of a proposed solution. A set of patches to m4-1.4.4 is provided separately.

2 Current unspecified but non-problematic behaviour

Redefining a macro with `define()` causes the new definition to be used for pending expansions:

```
define(‘f’,‘one’)f(define(‘f’,‘two’))
=> two
```

Redefining a macro with "pushdef" causes the original definition to be used for pending expansions:

```
define(‘f’,‘one’)f(pushdef(‘f’,‘two’)) f()
=> one two
```

`pushdef()` protects pushed definitions from `define()`, so that the original definition is used for pending expansions:

```
define(‘f’,‘one’)f(pushdef(‘f’,‘two’)f(define(‘f’,‘three’)))
=> one
```

If the definition of a macro with a pending expansion is changed with `define()`, its ultimate expansion is not affected by any subsequent `pushdef()`s:

```
define(‘f’,‘one $1’)f(define(‘f’,‘two $1’)f(pushdef(‘f’,‘three $1’)‘four’))
=> two two four
```

3 Current problematic behaviour

In what follows, the string `***JUNK***` represents non-printable or nonsense characters. Invoke m4 with ”-dqeat” to see the behaviour more clearly.

Undefining a macro with pending expansion exposes the dangling pointer bug:

```
define(‘f’,‘one’)f(undefine(‘f’)‘two’)
=> ***JUNK***
```

proposal: result should be `f(two)`

This also occurs when more than one definition for the macro has been made using pushdef:

```
define(‘f’,‘one’)f(pushdef(‘f’,‘two’)f(undefine(‘f’)‘three’))
=> ***JUNK***
```

proposal: result should be `f(f(three))`

Popping the definition of a pending expansion, even if there remains a definition for a symbol with the same name as the pending macro, exposes the bug. Here, the inner (pushed) definition of f becomes invalid, while the outer one remains correct.

```
define(‘f’,‘one $1’)f(pushdef(‘f’,‘two $1’)f(popdef(‘f’)three))
=> one ***JUNK***
```

proposal: result should be `one one three`

4 Source of the problem

Consider this example:

```
$ m4 -dqeat
define(`f', `level1 $1')f(f(f(undefine(`f')stuff)))
=> m4trace: -1- define(`f', `level1 $1')
=> m4trace: -4- undefine(`f')
=> m4trace: -3- ***JUNK***(`stuff') -> `***JUNK***'
=> m4trace: -2- ***JUNK***(`***JUNK***') -> `***JUNK***'
=> m4trace: -1- ***JUNK***(`***JUNK***') -> `***JUNK***'
```

There are three pending expansions of `f` when `undefine(`f')` is called. Simply deleting the definition of `f` from the symbol table at that point leaves three dangling pointers on the stack (in the local variable `sym` of function `expand_macro()`). Depending on how the memory allocation/deallocation routines work, this example might not be enough to expose the bug, since all `free()`d storage might be intact. A more complicated example that might expose the bug in such cases by preventing re-use of identical deallocated memory is this:

```
$ m4 -dqeat
define(`f', `level1 $1')f(f(f(undefine(`f')undefine(`include'))))
=> m4trace: -1- define(`f', `level1 $1')
=> m4trace: -4- undefine(`f')
=> m4trace: -4- undefine(`include')
=> m4trace: -3- ***JUNK***(`') -> `***JUNK***'
=> m4trace: -2- ***JUNK***(`***JUNK***') -> `***JUNK***'
=> m4trace: -1- ***JUNK***(`***JUNK***') -> `***JUNK***'
=> ***JUNK***
```

5 Goals of proposed fix

- maintain the existing behaviour in non-problematic cases: if, while its arguments are being expanded, a macro's definition is changed (by a `define()` not preceeded by any `pushdef()`s), use the changed definition (which might not be the most recent definition, since `pushdef()`s might have occurred since the redefinition) for expanding the macro.
- change the existing behaviour in problematic cases: if, while its arguments are being expanded, a macro's definition is deleted either by `popdef()` or `undefine()`, then use the *current* definition (i.e. the most recent non-deleted definition of the macro) when expanding the macro. If there is no non-deleted definition of the macro, then expand it as `$0($@@)`, which is identical behaviour to that when the macro is undefined, except that leading whitespace is stripped from its arguments. (A *non-deleted* definition is one still in the symbol table and for which `SYMBOL_DELETED` is false. See below.)

  QUESTION: is it worth the effort of maintaining or retrieving the original argument text to remove this leading whitespace discrepancy?
- do this with minimal impact on code and performance.

6 Note on m4 vs cpp

This fix makes explicit a difference between how m4 and cpp expand macros. According to GNU documentation for cpp, macro expansion always uses the definition of the macro at the time collection of its arguments begins, so that if an argument redefines the macro, this does not affect any pending expansions:

Current non-problematic m4 behaviour is already incompatible:

```
$ cpp -P                        $ m4
#define f(x) one         define(`f',`one')
f(                       f(define(`f',`two'))
#undef f                 => two
#define f(x) two
)
=> one
```

By specifying the following behaviour for m4 under currently problematic conditions, this fix widens the discrepancy:

```
$ cpp -P                            $ m4
#define f(x) one x                  define(`f',`one $1')
f(                                  f(undefine(`f')two)
#undef f                            => f(two)
two
)
=> one two
```

I am not suggesting semantic compatibility with cpp should be a goal for m4, but just pointing out where existing differences will be solidified, as this may be a source of confusion for people using both programs.

7 Mechanics of proposed fix

Add two fields to `struct symbol`:

- `int expansions_pending`: The number of pending expansions of this symbol definition. This will equal the number of pointers to this symbol table entry stored in the `sym` local variable of stack frames for `expand_macro()`. The initial value for a newly-defined symbol (whether or not it shadows an existing symbol) is 0. Accessed by macro `SYMBOL_EXPANSIONS_PENDING (sym)`.
- `boolean deleted`: `TRUE` if and only if this symbol represents a definition that has been deleted by `popdef()` or `undefine()`, but not yet removed from the symbol table because of pending expansions (i.e. to avoid dangling pointers). The initial value for a newly-defined symbol is `FALSE`. Accessed by macro `SYMBOL_DELETED (sym)`.

8 Changes to functions

- `builtin.c: expand_user_macro()` If `SYMBOL_DELETED (sym)` is `TRUE`, then expand this macro as `$0($@@)`.
- `macro.c: expand_macro()`

  Increase `SYMBOL_EXPANSIONS_PENDING (sym)` before arguments to this macro are collected.

  Decrease `SYMBOL_EXPANSIONS_PENDING (sym)` after arguments to this macro are collected, but before the macro is expanded. If argument collection has made `SYMBOL_DELETED (sym)` true, lookup the most recent non-deleted symbol of the same name before expanding the macro, if one exists. Use the `SYMBOL_INTERNAL_NAME` flag for `lookup_symbol()` because we already have the symbol table pointer to the name. Otherwise, since all symbols of the same name have been deleted, just use `sym`, and the expansion code will expand this `SYMBOL_DELETED` macro as `$0($@@)`.

  After expanding the macro, if `SYMBOL_EXPANSIONS_PENDING (sym)` has reached zero, and `SYMBOL_DELETED (sym) == TRUE`, then mark symbol by setting `SYMBOL_EXPANSIONS_PENDING(sym) = -1` and delete `sym` from the symbol table using `lookup_sym()` with `mode = SYMBOL_FINALIZE`.
- `macro.c: call_macro()`

  If `SYMBOL_TYPE (sym) == TOKEN_FUNC` and `SYMBOL_DELETED (sym) == TRUE`, call `expand_user_macro()` instead of calling the builtin function for this symbol. This corresponds to the case of a deleted builtin function with expansion pending. As noted above, this will expand the macro as `$0($@@)`.

- `symtab.c: lookup_symbol()` Use pointer comparisons instead of `strcmp()` for string equality when the target name is known to be internal.
  - case `mode == SYMBOL_LOOKUP`:
    Find the first symbol with name for which `SYMBOL_DELETED == FALSE`. If there is no such symbol (either because none match the name or because all those with matching name have `SYMBOL_DELETED == TRUE`), return NULL for undefined.
  - case `mode == SYMBOL_INSERT`:
    * case a) the symbol exists in the symbol table, with one or more definitions, and at least one has `SYMBOL_DELETED == FALSE`
      1. find the first symbol of that name with `SYMBOL_DELETED == FALSE`
      2. preserve the value of `EXPANSIONS_PENDING`
      3. set `SYMBOL_TRACED = FALSE`
    * case b) the symbol does not exist in the symbol table, or if it does, then all occurences have `SYMBOL_DELETED == TRUE`
      1. allocate a new symbol
      2. allocate a new copy of the symbol name if no symbol of this name exists
      3. set `SYMBOL_DELETED = FALSE`
      4. set `EXPANSIONS_PENDING = 0`
  - case `mode == SYMBOL_PUSHDEF`:
    1. allocate a new symbol
    2. allocate a new copy of the symbol name, if no symbol of this name exists
    3. set `SYMBOL_DELETED = FALSE`
    4. set `EXPANSIONS_PENDING = 0`
    5. mark the next non-DELETED symbol of this name as `SYMBOL_SHADOWED`, and copy its value of `SYMBOL_TRACED` status to the new symbol.
  - case `mode == SYMBOL_DELETE`:
    For every symbol, `sym`, matching `name`, if `SYMBOL_EXPANSIONS_PENDING`

    `(sym) > 0`, set `SYMBOL_DELETED (sym) = TRUE` and do not free it; otherwise, free `sym`.
  - case `mode == SYMBOL_POPDEF`:
    Find the first symbol, `sym`, matching name for which `SYMBOL_DELETED == FALSE`. If `SYMBOL_EXPANSIONS_PENDING (sym) > 0`, mark it as set `SYMBOL_DELETED (sym) = TRUE` and do not free it. Otherwise, free it.

  - case `mode == SYMBOL_FINALIZE`:

    Find and remove the `sym` matching `name` and flagged with `EXPANSIONS_PENDING == -1` from the symbol table and free its storage.

- `symtab.c: free_symbol()`

  If `sym` is the last remaining symbol with its name, free the storage for that name. This can be checked by examining the `SYMBOL_NAME` pointers of the preceding symbol (now passed as a parameter) and next symbol. Otherwise, do not free the symbol name. Free the symbol table entry.