NAME

bzero, explicit_bzero − zero a byte string

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <strings.h>

void bzero(void *s***[.***n***], size_t** *n***);**

#include <string.h>

void explicit_bzero(void *s***[.***n***], size_t** *n***);**

DESCRIPTION

The **bzero**() function erases the data in the *n* bytes of the memory starting at the location pointed to by *s*, by writing zeros (bytes containing '\0') to that area.

The **explicit_bzero**() function performs the same task as **bzero**(). It differs from **bzero**() in that it guarantees that compiler optimizations will not remove the erase operation if the compiler deduces that the operation is "unnecessary".

RETURN VALUE

None.

VERSIONS

explicit_bzero() first appeared in glibc 2.25.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
bzero(), **explicit_bzero**()	Thread safety	MT-Safe

STANDARDS

The **bzero**() function is deprecated (marked as LEGACY in POSIX.1-2001); use **memset**(3) in new programs. POSIX.1-2008 removes the specification of **bzero**(). The**bzer o**() function first appeared in 4.3BSD.

The **explicit_bzero**() function is a nonstandard extension that is also present on some of the BSDs. Some other implementations have a similar function, such as **memset_explicit**() or **memset_s**().

NOTES

The **explicit_bzero**() function addresses a problem that security-conscious applications may run into when using **bzero**(): if the compiler can deduce that the location to be zeroed will never again be touched by a *correct* program, then it may remove the **bzero**() call altogether. This is a problem if the intent of the**bzero**() call was to erase sensitive data (e.g., passwords) to prevent the possibility that the data was leaked by an incorrect or compromised program. Calls to **explicit_bzero**() are never optimized away by the compiler.

The **explicit_bzero**() function does not solve all problems associated with erasing sensitive data:

- The **explicit_bzero**() function does *not* guarantee that sensitive data is completely erased from memory. (The same is true of **bzero**().) For example, there may be copies of the sensitive data in a register and in "scratch" stack areas. The **explicit_bzero**() function is not aware of these copies, and can’t erase them.
- In some circumstances, **explicit_bzero**() can *decrease* security. If the compiler determined that the variable containing the sensitive data could be optimized to be stored in a register (because it is small enough to fit in a register, and no operation other than the **explicit_bzero**() call would need to take the address of the variable), then the **explicit_bzero**() call will force the data to be copied from the register to a location in RAM that is then immediately erased (while the copy in the register remains unaffected). The problem here is that data in RAM is more likely to be exposed by a bug than data in a register, and thus the **explicit_bzero**() call creates a brief time window where the sensitive data is more vulnera-

ble than it would otherwise have been if no attempt had been made to erase the data.

Note that declaring the sensitive variable with the **volatile** qualifier does *not* eliminate the above problems. Indeed, it will make them worse, since, for example, it may force a variable that would otherwise have been optimized into a register to instead be maintained in (more vulnerable) RAM for its entire lifetime.

Notwithstanding the above details, for security-conscious applications, using **explicit_bzero**() is generally preferable to not using it. The developers of **explicit_bzero**() anticipate that future compilers will recognize calls to **explicit_bzero**() and take steps to ensure that all copies of the sensitive data are erased, including copies in registers or in "scratch" stack areas.

SEE ALSO

bstring(3), **memset**(3), **swab**(3)