EXP34-C. Do not dereference null pointers

Dereferencing a null pointer is undefined behavior.

On many platforms, dereferencing a null pointer results in abnormal program termination, but this is not required by the standard. See "Clever Attack Exploits Fully-Patched Linux Kernel" [Goodin 2009] for an example of a code execution exploit that resulted from a null pointer dereference.

Noncompliant Code Example

This noncompliant code example is derived from a real-world example taken from a vulnerable version of the `libpng` library as deployed on a popular ARM-based cell phone [Jack 2007]. The `libpng` library allows applications to read, create, and manipulate PNG (Portable Network Graphics) raster image files. The `libpng` library implements its own wrapper to `malloc()` that returns a null pointer on error or on being passed a 0-byte-length argument.

This code also violates ERR33-C. Detect and handle standard library errors.

```
#include <png.h> /* From libpng */
#include <string.h>

void func(png_structp png_ptr, int length, const void *user_data) {
  png_charp chunkdata;
  chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
  /* ... */
  memcpy(chunkdata, user_data, length);
  /* ... */
 }
```

If `length` has the value `1`, the addition yields 0, and `png_malloc()` subsequently returns a null pointer, which is assigned to `chunkdata`. The `chunkdat` pointer is later used as a destination argument in a call to `memcpy()`, resulting in user-defined data overwriting memory starting at address 0. In the case of the ARM and XScale architectures, the `0x0` address is mapped in memory and serves as the exception vector table; consequently, dereferencing `0x0` did not cause an abnormal program termination.

Compliant Solution

This compliant solution ensures that the pointer returned by `png_malloc()` is not null. It also uses the unsigned type `size_t` to pass the `length` parameter, ensuring that negative values are not passed to `func()`.

This solution also ensures that the `user_data` pointer is not null. Passing a null pointer to memcpy() would produce undefined behavior, even if the number of bytes to copy were 0. The `user_data` pointer could be invalid in other ways, such as pointing to freed memory. However there is no portable way to verify that the pointer is valid, other than checking for null.

```
#include <png.h> /* From libpng */
#include <string.h>

 void func(png_structp png_ptr, size_t length, const void *user_data) {
  png_charp chunkdata;
  if (length == SIZE_MAX) {
    /* Handle error */
  }
  chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
  if (NULL == chunkdata) {
    /* Handle error */
  }
  if (NULL == user_data) {
    /* Handle error */
  }
  /* ... */
  memcpy(chunkdata, user_data, length);
  /* ... */

 }
```

Noncompliant Code Example

In this noncompliant code example, `input_str` is copied into dynamically allocated memory referenced by `c_str`. If `malloc()` fails, it returns a null pointer that is assigned to `c_str`. When `c_str` is dereferenced in `memcpy()`, the program exhibits undefined behavior. Additionally, if `input_str` is a null pointer, the call to `strlen()` dereferences a null pointer, also resulting in undefined behavior. This code also violates ERR33-C. Detect and handle standard library errors.

```
#include <string.h>
#include <stdlib.h>

void f(const char *input_str) {
  size_t size = strlen(input_str) + 1;
  char *c_str = (char *)malloc(size);
  memcpy(c_str, input_str, size);
  /* ... */
  free(c_str);
  c_str = NULL;
  /* ... */
}
```

Compliant Solution

This compliant solution ensures that both `input_str` and the pointer returned by `malloc()` are not null:

```
#include <string.h>
#include <stdlib.h>

void f(const char *input_str) {
  size_t size;
  char *c_str;

  if (NULL == input_str) {
    /* Handle error */
  }

  size = strlen(input_str) + 1;
  c_str = (char *)malloc(size);
  if (NULL == c_str) {
    /* Handle error */
  }
  memcpy(c_str, input_str, size);
  /* ... */
  free(c_str);
  c_str = NULL;
  /* ... */
}
```

Noncompliant Code Example

This noncompliant code example is from a version of `drivers/net/tun.c` and affects Linux kernel 2.6.30 [Goodin 2009]:

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait)  {
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk = tun->sk;
  unsigned int mask = 0;

  if (!tun)
    return POLLERR;

  DBG(KERN_INFO "%s: tun_chr_poll\n", tun->dev->name);

  poll_wait(file, &tun->socket.wait, wait);

  if (!skb_queue_empty(&tun->readq))
    mask |= POLLIN | POLLRDNORM;

  if (sock_writeable(sk) ||
     (!test_and_set_bit(SOCK_ASYNC_NOSPACE, &sk->sk_socket->flags) &&
     sock_writeable(sk)))
    mask |= POLLOUT | POLLWRNORM;

  if (tun->dev->reg_state != NETREG_REGISTERED)
    mask = POLLERR;

  tun_put(tun);
  return mask;
}
```

The `sk` pointer is initialized to `tun->sk` before checking if `tun` is a null pointer. Because null pointer dereferencing is undefined behavior, the compiler (GCC in this case) can optimize away the `if (!tun)` check because it is performed after `tun->sk` is accessed, implying that `tun` is non-null. As a result, this noncompliant code example is vulnerable to a null pointer dereference exploit, because null pointer dereferencing can be permitted on several platforms, for example, by using `mmap(2)` with the `MAP_FIXED` flag on Linux and Mac OS X, or by using the `shmat()` POSIX function with the `SHM_RND` flag [Liu 2009].

Compliant Solution

This compliant solution eliminates the null pointer deference by initializing `sk` to `tun->sk` following the null pointer check. It also adds assertions to document that certain other pointers must not be null.

```
static unsigned int tun_chr_poll(struct file *file, poll_table *wait)  {
  assert(file);
  struct tun_file *tfile = file->private_data;
  struct tun_struct *tun = __tun_get(tfile);
  struct sock *sk;
  unsigned int mask = 0;

  if (!tun)
    return POLLERR;
  assert(tun->dev);
  sk = tun->sk;
  assert(sk);
  assert(sk->socket);
  /* The remaining code is omitted because it is unchanged... */
}
```

Risk Assessment

Dereferencing a null pointer is undefined behavior, typically abnormal program termination. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code [Jack 2007, van Sprundel 2006]. The indicated severity is for this more severe case; on platforms where it is not possible to exploit a null pointer dereference to execute arbitrary code, the actual severity is low.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP34-C	High	Likely	Medium	**P18**	**L1**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**null-dereferencing**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-EXP34**	
CodeSonar	5.2p0	**LANG.MEM.NPD** **LANG.STRUCT.NTAD** **LANG.STRUCT.UPD**	Null pointer dereference Null test after dereference Unchecked parameter dereference
Compass/ROSE			Can detect violations of this rule. In particular, ROSE ensures that any pointer returned by `malloc()`, `calloc()`, or `realloc()` is first checked for `NULL` before being used (otherwise, it is `free()`-ed). ROSE does not handle cases where an allocation is assigned to an lvalue that is not a variable (such as a `struct` member or C++ function call returning a reference)
Coverity	2017.07	**CHECKED_RETURN** **NULL_RETURNS** **REVERSE_INULL** **FORWARD_NULL**	Finds instances where a pointer is checked against `NULL` and then later dereferenced Identifies functions that can return a null pointer but are not checked Identifies code that dereferences a pointer and then checks the pointer against `NULL` Can find the instances where `NULL` is explicitly dereferenced or a pointer is checked against `NULL` but then dereferenced anyway. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary
Cppcheck	1.66	**nullPointer, nullPointerDefaultArg, nullPointerRedundantCheck**	Context sensitive analysis Detects when NULL is dereferenced (Array of pointers is not checked. Pointer members in structs are not checked.) Finds instances where a pointer is checked against `NULL` and then later dereferenced Identifies code that dereferences a pointer and then checks the pointer against `NULL` Does not guess that return values from `malloc()`, `strchr()`, etc., can be `NULL` (The return value from `malloc()` is `NULL` only if there is OOMo and the dev might not care to handle that. The return value from `strchr()` is often `NULL`, but the dev might know that a specific `strchr()` function call will not return `NULL`.)
Klocwork	2018	**NPD.CHECK.CALL.MIGHT** **NPD.CHECK.CALL.MUST** **NPD.CHECK.MIGHT** **NPD.CHECK.MUST** **NPD.CONST.CALL** **NPD.CONST.DEREF** **NPD.FUNC.CALL.MIGHT** **NPD.FUNC.CALL.MUST** **NPD.FUNC.MIGHT** **NPD.FUNC.MUST** **NPD.GEN.CALL.MIGHT** **NPD.GEN.CALL.MUST** **NPD.GEN.MIGHT** **NPD.GEN.MUST** **RNPD.CALL** **RNPD.DEREF**	
LDRA tool suite	9.7.1	**45 D, 123 D, 128 D, 129 D, 130 D, 131 D, 652 S**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_C-EXP34-a**	Avoid null pointer dereferencing
Parasoft Insure++			Runtime analysis
Polyspace Bug Finder	R2019b	CERT C: Rule EXP34-C	Checks for use of null pointers (rule partially covered)

PRQA QA-C	9.7	**2810, 2811, 2812, 2813**	Fully implemented
PRQA QA-C++	4.4	**2810, 2811, 2812, 2813**	
PVS-Studio	6.23	**V522**, **V595**, **V664**, **V713**, **V1004**	
SonarQube C/C++ Plugin	3.11	**S2259**	
Splint	3.1.1		
TrustInSoft Analyzer	1.38	**mem_access**	Exhaustively verified (see one compliant and one non-compliant example).

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT Oracle Secure Coding Standard for Java	EXP01-J. Do not use a null in a case where an object is required	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Pointer Casting and Pointer Type Changes [HFC]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Null Pointer Dereference [XYH]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TS 17961	Dereferencing an out-of-domain pointer [nullref]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-476, NULL Pointer Dereference	2017-07-06: CERT: Exact

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-690 and EXP34-C

EXP34-C = Union(CWE-690, list) where list =

- Dereferencing null pointers that were not returned by a function

CWE-252 and EXP34-C

Intersection(CWE-252, EXP34-C) = Ø

EXP34-C is a common consequence of ignoring function return values, but it is a distinct error, and can occur in other scenarios too.

Bibliography

[Goodin 2009]	
[Jack 2007]	
[Liu 2009]	

[van Sprundel 2006]	
[Viega 2005]	Section 5.2.18, "Null-Pointer Dereference"