EXP36-C. Do not cast pointers into more strictly aligned pointer types

Do not convert a pointer value to a pointer type that is more strictly aligned than the referenced type. Different alignments are possible for different types of objects. If the type-checking system is overridden by an explicit cast or the pointer is converted to a void pointer (`void *`) and then to a different type, the alignment of an object may be changed.

The C Standard, 6.3.2.3, paragraph 7 [ISO/IEC 9899:2011], states

> *A pointer to an object or incomplete type may be converted to a pointer to a different object or incomplete type. If the resulting pointer is not correctly aligned for the referenced type, the behavior is undefined.*

See undefined behavior 25.

If the misaligned pointer is dereferenced, the program may terminate abnormally. On some architectures, the cast alone may cause a loss of information even if the value is not dereferenced if the types involved have differing alignment requirements.

Noncompliant Code Example

In this noncompliant example, the `char` pointer `&c` is converted to the more strictly aligned `int` pointer `ip`. On some implementations, `cp` will not match `&c` . As a result, if a pointer to one object type is converted to a pointer to a different object type, the second object type must not require stricter alignment than the first.

```
#include <assert.h>

void func(void) {
  char c = 'x';
  int *ip = (int *)&c; /* This can lose information */
  char *cp = (char *)ip;

  /* Will fail on some conforming implementations */
  assert(cp == &c);
}
```

Compliant Solution (Intermediate Object)

In this compliant solution, the `char` value is stored into an object of type `int` so that the pointer's value will be properly aligned:

```
#include <assert.h>

void func(void) {
  char c = 'x';
  int i = c;
  int *ip = &i;

  assert(ip == &i);
}
```

Noncompliant Code Example

The C Standard allows any object pointer to be cast to and from `void *`. As a result, it is possible to silently convert from one pointer type to another without the compiler diagnosing the problem by storing or casting a pointer to `void *` and then storing or casting it to the final type. In this noncompliant code example, `loop_function()` is passed the `char` pointer `char_ptr` but returns an object of type `int` pointer:

```
int *loop_function(void *v_pointer) {
  /* ... */
  return v_pointer;
}

void func(char *char_ptr) {
  int *int_ptr = loop_function(char_ptr);

  /* ... */
}
```

This example compiles without warning using GCC 4.8 on Ubuntu Linux 14.04. However, `int_pointer` can be more strictly aligned than an object of type `char *`.

Compliant Solution

Because the input parameter directly influences the return value, and `loop_function()` returns an object of type `int *`, the formal parameter `v_pointer` is redeclared to accept only an object of type `int *`:

```
int *loop_function(int *v_pointer) {
  /* ... */
  return v_pointer;
}

void func(int *loop_ptr) {
  int *int_ptr = loop_function(loop_ptr);

  /* ... */
}
```

Noncompliant Code Example

Some architectures require that pointers are correctly aligned when accessing objects larger than a byte. However, it is common in system code that unaligned data (for example, the network stacks) must be copied to a properly aligned memory location, such as in this noncompliant code example:

```
#include <string.h>

struct foo_header {
  int len;
  /* ... */
};

void func(char *data, size_t offset) {
  struct foo_header *tmp;
  struct foo_header header;

  tmp = (struct foo_header *)(data + offset);
  memcpy(&header, tmp, sizeof(header));

  /* ... */
}
```

Assigning an unaligned value to a pointer that references a type that needs to be aligned is undefined behavior. An implementation may notice, for example, that `tmp` and `header` must be aligned and use an inline `memcpy()` that uses instructions that assume aligned data.

Compliant Solution

This compliant solution avoids the use of the `foo_header` pointer:

```
#include <string.h>

struct foo_header {
  int len;
  /* ... */
};

void func(char *data, size_t offset) {
  struct foo_header header;
  memcpy(&header, data + offset, sizeof(header));

  /* ... */
}
```

Exceptions

EXP36-C-EX1: Some hardware architectures have relaxed requirements with regard to pointer alignment. Using a pointer that is not properly aligned is correctly handled by the architecture, although there might be a performance penalty. On such an architecture, improper pointer alignment is permitted but remains an efficiency problem.

The x86 32- and 64-bit architectures usually impose only a performance penalty for violations of this rule, but under some circumstances, noncompliant code can still exhibit undefined behavior. Consider the following program:

```
#include <stdio.h>
#include <stdint.h>

#define READ_UINT16(ptr)       (*(uint16_t *)(ptr))
#define WRITE_UINT16(ptr, val) (*(uint16_t *)(ptr) = (val))

void compute(unsigned char *b1, unsigned char *b2,
             int value, int range) {
  int i;
  for (i = 0; i < range; i++) {
    int newval = (int)READ_UINT16(b1) + value;
    WRITE_UINT16(b2, newval);
    b1 += 2;
    b2 += 2;
  }
}

int main() {
  unsigned char buffer1[1024];
  unsigned char buffer2[1024];
  printf("Compute something\n");
  compute(buffer1 + 3, buffer2 + 1, 42, 500);
  return 0;
}
```

This code tries to read short ints (which are 16 bits long) from odd pairs in a character array, which violates this rule. On 32- and 64-bit x86 platforms, this program should run to completion without incident. However, the program aborts with a SIGSEGV due to the unaligned reads on a 64-bit platform running Debian Linux, when compiled with GCC 4.9.4 using the flags `-O3` or `-O2 -ftree-loop-vectorize -fvect-cost-model`.

If a developer wishes to violate this rule and use undefined behavior, they must not only ensure that the hardware guarantees the behavior of the object code, but they must also ensure that their compiler, along with its optimizer, also respect these guarantees.

EXP36-C-EX2: If a pointer is known to be correctly aligned to the target type, then a cast to that type is permitted. There are several cases where a pointer is known to be correctly aligned to the target type. The pointer could point to an object declared with a suitable alignment specifier. It could point to an object returned by `aligned_alloc()`, `calloc()`, `malloc()`, or `realloc()`, as per the C standard, section 7.22.3, paragraph 1 [ISO/IEC 9899:2011].

This compliant solution uses the alignment specifier, which is new to C11, to declare the `char` object `c` with the same alignment as that of an object of type `int`. As a result, the two pointers reference equally aligned pointer types:

```
#include <stdalign.h>
#include <assert.h>

void func(void) {
  /* Align c to the alignment of an int */
  alignas(int) char c = 'x';
  int *ip = (int *)&c;
  char *cp = (char *)ip;
  /* Both cp and &c point to equally aligned objects */
  assert(cp == &c);
}
```

Risk Assessment

Accessing a pointer or an object that is not properly aligned can cause a program to crash or give erroneous information, or it can cause slow pointer accesses (if the architecture allows misaligned accesses).

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP36-C	Low	Probable	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**pointer-cast-alignment**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-EXP36**	
Compass/ROSE			Can detect violations of this rule. However, it does not flag explicit casts to `void *` and then back to another pointer type
Coverity	2017.07	**MISRA C 2004 Rule 11.4** **MISRA C 2012 Rule 11.1** **MISRA C 2012 Rule 11.2** **MISRA C 2012 Rule 11.5** **MISRA C 2012 Rule 11.7**	Implemented
ECLAIR	1.2	**CC2.EXP36**	Fully implemented
EDG			
GCC	4.3.5		Can detect some violations of this rule when the `-Wcast-align` flag is used
Klocwork	2018	**MISRA.CAST.PTR.UNRELATED** **MISRA.CAST.PTR_TO_INT** **PORTING.CAST.PTR.FLTPNT** **PORTING.CAST.PTR** **PORTING.CAST.PTR.SIZE** **PORTING.CAST.SIZE**	
LDRA tool suite	9.7.1	**94 S, 606 S**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-EXP36-a**	A cast should not be performed between a pointer to object type and a different pointer to object type
Polyspace Bug Finder	R2019b	CERT C: Rule EXP36-C	Checks for wrong allocated object size for cast (rule fully covered)
PRQA QA-C	9.5	**0326, 3305**	Fully implemented
PRQA QA-C++	4.3	**3033, 3038**	
PVS-Studio	6.23	**V548**, **V641**	
RuleChecker	19.04	**pointer-cast-alignment**	Fully checked

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 C	VOID EXP56-CPP. Do not cast pointers into more strictly aligned pointer types	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 TS 17961	Converting pointer values to more strictly aligned pointer types [alignconv]	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 11.1 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 11.2 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 11.5 (advisory)	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 11.7 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship

Bibliography

[Bryant 2003]	
[ISO/IEC 9899:2011]	6.3.2.3, "Pointers"
[Walfridsson 2003]	Aliasing, Pointer Casts and GCC 3.3