DCL30-C. Declare objects with appropriate storage durations

Every object has a storage duration that determines its lifetime: *static*, *thread*, *automatic*, or *allocated*.

According to the C Standard, 6.2.4, paragraph 2 [ISO/IEC 9899:2011],

> *The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.*

Do not attempt to access an object outside of its lifetime. Attempting to do so is undefined behavior and can lead to an exploitable vulnerability. (See also undefined behavior 9 in the C Standard, Annex J.)

Noncompliant Code Example (Differing Storage Durations)

In this noncompliant code example, the address of the variable `c_str` with automatic storage duration is assigned to the variable `p`, which has static storage duration. The assignment itself is valid, but it is invalid for `c_str` to go out of scope while `p` holds its address, as happens at the end of `dont_do_this()`.

```
#include <stdio.h>

const char *p;
void dont_do_this(void) {
  const char c_str[] = "This will change";
  p = c_str; /* Dangerous */
}

void innocuous(void) {
  printf("%s\n", p);
}

int main(void) {
  dont_do_this();
  innocuous();
  return 0;
}
```

Compliant Solution (Same Storage Durations)

In this compliant solution, `p` is declared with the same storage duration as `c_str`, preventing `p` from taking on an indeterminate value outside of `this_is_OK()`:

```
void this_is_OK(void) {
  const char c_str[] = "Everything OK";
  const char *p = c_str;
  /* ... */
}
/* p is inaccessible outside the scope of string c_str */
```

Alternatively, both `p` and `c_str` could be declared with static storage duration.

Compliant Solution (Differing Storage Durations)

If it is necessary for `p` to be defined with static storage duration but `c_str` with a more limited duration, then `p` can be set to `NULL` before `c_str` is destroyed. This practice prevents `p` from taking on an indeterminate value, although any references to `p` must check for `NULL`.

```
const char *p;
void is_this_OK(void) {
  const char c_str[] = "Everything OK?";
  p = c_str;
  /* ... */
  p = NULL;
}
```

Noncompliant Code Example (Return Values)

In this noncompliant code sample, the function `init_array()` returns a pointer to a character array with automatic storage duration, which is accessible to the caller:

```
char *init_array(void) {
  char array[10];
  /* Initialize array */
  return array;
}
```

Some compilers generate a diagnostic message when a pointer to an object with automatic storage duration is returned from a function, as in this example. Programmers should compile code at high warning levels and resolve any diagnostic messages. (See MSC00-C. Compile cleanly at high warning levels.)

Compliant Solution (Return Values)

The solution, in this case, depends on the intent of the programmer. If the intent is to modify the value of `array` and have that modification persist outside the scope of `init_array()`, the desired behavior can be achieved by declaring `array` elsewhere and passing it as an argument to `init_array()`:

```
#include <stddef.h>
void init_array(char *array, size_t len) {
  /* Initialize array */
  return;
}

int main(void) {
  char array[10];
  init_array(array, sizeof(array) / sizeof(array[0]));
  /* ... */
  return 0;
}
```

Noncompliant Code Example (Output Parameter)

In this noncompliant code example, the function `squirrel_away()` stores a pointer to local variable `local` into a location pointed to by function parameter `ptr_param`. Upon the return of `squirrel_away()`, the pointer `ptr_param` points to a variable that has an expired lifetime.

```
void squirrel_away(char **ptr_param) {
  char local[10];
  /* Initialize array */
  *ptr_param = local;
}

void rodent(void) {
  char *ptr;
  squirrel_away(&ptr);
  /* ptr is live but invalid here */
}
```

Compliant Solution (Output Parameter)

In this compliant solution, the variable `local` has static storage duration; consequently, `ptr` can be used to reference the `local` array within the `rodent()` function:

```
char local[10];

void squirrel_away(char **ptr_param) {
  /* Initialize array */
  *ptr_param = local;
}

void rodent(void) {
  char *ptr;
  squirrel_away(&ptr);
  /* ptr is valid in this scope */
}
```

Risk Assessment

Referencing an object outside of its lifetime can result in an attacker being able to execute arbitrary code.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL30-C	High	Probable	High	P6	L2

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**pointered-deallocation**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-DCL30**	Fully implemented
CodeSonar	5.2p0	**LANG.STRUCT.RPL**	Returns pointer to local
Compass/ROSE			Can detect violations of this rule. It automatically detects returning pointers to local variables. Detecting more general cases, such as examples where static pointers are set to local variables which then go out of scope, would be difficult
Coverity	2017.07	**RETURN_LOCAL**	Finds many instances where a function will return a pointer to a local stack variable. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary
Klocwork	2018	**LOCRET.ARG** **LOCRET.GLOB** **LOCRET.RET**	
LDRA tool suite	9.7.1	**42 D, 77 D, 71 S, 565 S**	Enhanced Enforcement
Parasoft C/C++test	10.4.2	**CERT_C-DCL30-a** **CERT_C-DCL30-b**	The address of an object with automatic storage shall not be returned from a function The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist
Polyspace Bug Finder	R2019b	CERT C: Rule DCL30-C	Checks for pointer or reference to stack variable leaving scope (rule fully covered)
PRQA QA-C	9.7	**3217, 3225, 3230, 4140**	Partially implemented
PRQA QA-C++	4.4	**2515, 2516, 2527, 2528, 4026, 4624, 4629**	
PVS-Studio	6.23	**V506**, **V507**, **V558**, **V623**, **V723**, **V738**	
Splint	3.1.1		
TrustInSoft Analyzer	1.38	**dangling_pointer**	Exhaustively detects undefined behavior (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 C Secure Coding Standard	MSC00-C. Compile cleanly at high warning levels	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C	EXP54-CPP. Do not access an object outside of its lifetime	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TS 17961	Escaping of the address of an automatic object [addrescape]	Prior to 2018-01-12: CERT: Unspecified Relationship
MISRA C:2012	Rule 18.6 (required)	Prior to 2018-01-12: CERT: Unspecified Relationship

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-562 and DCL30-C

DCL30-C = Union(CWE-562, list) where list =

- Assigning a stack pointer to an argument (thereby letting it outlive the current function

Bibliography

[Coverity 2007]	
[ISO/IEC 9899:2011]	6.2.4, "Storage Durations of Objects"