DCL39-C. Avoid information leakage in structure padding

Created by Bhadrinath, last modified by Carol J. Lallier on Apr 10, 2014

The C Standard, 6.7.2.1, discusses the layout of structure fields. It specifies that non-bit-field members are aligned in an implementation-defined manner and that there may be padding within or at the end of a structure. Furthermore, initializing the members of the structure does not guarantee initialization of the padding bytes. The C Standard, 6.2.6.1, paragraph 6 [ISO/IEC 9899:2011], states:

> When a value is stored in an object of structure or union type, including in a member object, the bytes of the object representation that correspond to any padding bytes take unspecified values.

When passing a pointer to a structure across a trust boundary to a different trusted domain, programmers must ensure that the padding bytes of these structures do not contain sensitive information.

Noncompliant Code Example

This noncompliant code example runs in kernel space and copies data from `struct test` to user space. However, padding bytes may be used within the structure, for example, to ensure the proper alignment of the structure members. These padding bytes may contain sensitive information, which may then be leaked when the data is copied to user space.

```
#include <stddef.h>

struct test {
  int a;
  char b;
  int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
  struct test arg = {.a = 1, .b = 2, .c = 3};
  copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Noncompliant Code Example (`memset()`)

The padding bytes can be explicitly initialized by calling `memset()`:

```
#include <string.h>

struct test {
  int a;
  char b;
  int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
  struct test arg;

  /* Set all bytes (including padding bytes) to zero */
  memset(&arg, 0, sizeof(arg));

  arg.a = 1;
  arg.b = 2;
  arg.c = 3;

  copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

However, compilers are free to implement `arg.b = 2` by setting the low byte of a 32-bit register to 2, leaving the high bytes unchanged, and storing all 32 bits of the register into memory. This could leak the high-order bytes resident in the register to a user.

Compliant Solution

This compliant solution serializes the structure data before copying it to an untrusted context:

```
#include <stddef.h>

struct test {
  int a;
  char b;
  int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);
```

```
void do_stuff(void *usr_buf) {
  struct test arg = {.a = 1, .b = 2, .c = 3};
  /* May be larger than strictly needed */
  unsigned char buf[sizeof(arg)];
  size_t offset = 0;

  memcpy(buf + offset, &arg.a, sizeof(arg.a));
  offset += sizeof(arg.a);
  memcpy(buf + offset, &arg.b, sizeof(arg.b));
  offset += sizeof(arg.b);
  memcpy(buf + offset, &arg.c, sizeof(arg.c));
  offset += sizeof(arg.c);

  copy_to_user(usr_buf, buf, offset /* size of info copied */);
}
```

This code ensures that no uninitialized padding bytes are copied to unprivileged users. Note that the structure copied to user space is now a packed structure and that the `copy_to_user()` function would need to unpack it to re-create the original padded structure.

Compliant Solution (Padding Bytes)

Padding bytes can be explicitly declared as fields within the structure. This solution is not portable, however, because it depends on the implementation and target memory architecture. The following solution is specific to the x86-32 architecture:

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

struct test {
  int a;
  char b;
  char padding_1, padding_2, padding_3;
  int c;
};

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
  /* Ensure c is the next byte after the last padding byte */
  static_assert(offsetof(struct test, c) ==
                offsetof(struct test, padding_3) + 1,
                "Structure contains intermediate padding");
  /* Ensure there is no trailing padding */
  static_assert(sizeof(struct test) ==
                offsetof(struct test, c) + sizeof(int),
                "Structure contains trailing padding");
```

```
  struct test arg = {.a = 1, .b = 2, .c = 3};
  arg.padding_1 = 0;
  arg.padding_2 = 0;
  arg.padding_3 = 0;
  copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

The C Standard `static_assert()` macro accepts a constant expression and an error message. The expression is evaluated at compile time and, if false, the compilation is terminated and the error message is output (see DCL03-C. Use a static assertion to test the value of a constant expression for more details). The explicit insertion of the padding bytes into the `struct` should ensure that no additional padding bytes are added by the compiler, and consequently both static assertions should be true. However, it is necessary to validate these assumptions to ensure that the solution is correct for a particular implementation.

Compliant Solution (Structure Packing—GCC)

GCC allows specifying declaration attributes using the keyword `__attribute__((__packed__))`. When this attribute is present, the compiler will not add padding bytes for memory alignment unless otherwise required to by the `_Alignas` alignment specifier, and it will attempt to place fields at adjacent memory offsets when possible.

```
#include <stddef.h>

struct test {
  int a;
  char b;
  int c;
} __attribute__((__packed__));

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
  struct test arg = {.a = 1, .b = 2, .c = 3};
  copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

Compliant Solution (Structure Packing—Microsoft Visual Studio)

Microsoft Visual Studio supports `#pragma pack()` to suppress padding bytes [MSDN]. The compiler adds padding bytes for memory alignment depending on the current packing mode but still honors alignment specified by `__declspec(align())`. In this compliant solution, the packing mode is set to 1 in an attempt to ensure all fields are given adjacent offsets:

```
#include <stddef.h>

#pragma pack(push, 1) /* 1 byte */
struct test {
```

```
  int a;
  char b;
  int c;
};
#pragma pack(pop)

/* Safely copy bytes to user space */
extern int copy_to_user(void *dest, void *src, size_t size);

void do_stuff(void *usr_buf) {
  struct test arg = {1, 2, 3};
  copy_to_user(usr_buf, &arg, sizeof(arg));
}
```

The `pack` pragma takes effect at the first `struct` declaration after the pragma is seen.

Risk Assessment

Padding bytes might contain sensitive data because the C Standard allows any padding bytes to take unspecified values. A pointer to such a structure could be passed to other functions, causing information leakage.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DCL39-C	Low	Unlikely	High	**P1**	**L3**

Related Vulnerabilities

Numerous vulnerabilities in the Linux Kernel have resulted from violations of this rule. CVE-2010-4083 describes a vulnerability in which the `semctl()` system call allows unprivileged users to read uninitialized kernel stack memory, because various fields of a `semid_ds struct` declared on the stack are not altered or zeroed before being copied back to the user. CVE-2010-3881 describes a vulnerability in which structure padding and reserved fields in certain data structures in `QEMU-KVM` were not initialized properly before being copied to user space. A privileged host user with access to `/dev/kvm` could use this flaw to leak kernel stack memory to user space. CVE-2010-3477 describes a kernel information leak in `act_police` where incorrectly initialized structures in the traffic-control dump code may allow the disclosure of kernel memory to user space applications.

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

Related Guidelines

CERT C Secure Coding Standard	DCL03-C. Use a static assertion to test the value of a constant expression

Bibliography

[ISO/IEC 9899:2011]	6.2.6.1, "General" 6.7.2.1, "Structure and Union Specifiers"
[Graff 2003]	

[Sun 1993]	

rule dcl review review-dms review-ajb review-rcs

3 Comments

Martin Sebor

A few suggestions:

1. It's a common complaint raised against the CERT Secure Coding Standards that code examples use shorthand or comments instead of demonstrating the problems they discuss and some don't even compile. To avoid these complaints it would be helpful if the code examples could be compiled with no syntax errors.
2. Since the CERT practices are intended to be generally applicable in portable programs, the first and most prominent compliant solutions should be the one that's portable and that avoids relying on implementation-specific extensions such as gcc `__attribute__` or Visual C `#pragma`.
3. To avoid distracting readers with code that's unrelated to the problem or solution being discussed, it's best to provide code examples that are concise, to the point, and free of any features or techniques that go beyond the scope of the subject (such as the crafty implementation of `static_assert`).

Robert Seacord (Manager)

Martin,

Problem 1 definitely needs to be addressed.

For problem 2, it has generally been the practice in the secure coding standard to start with compliant but possibly flawed solutions, point out their weaknesses, and then work towards the more general solutions. This assumes that these solutions are not dominated by the portable solution for all possible quality attributes. Your approach may be as good or better, but would require revisiting a number of existing rules.

For problem 3, I don't think I mind including static_assert() in the solution as it protects against this compliant solution failing. However, I do think defining the macro is distracting in the solution. Instead, I would just use the C1X static_assert and reference guideline:

DCL39-C. Avoid information leak in structure padding

following the example.

DCL39-C also needs to be updated to discuss static_assert in C1X.

rCs

Brendan Saulsbury

I'm pretty sure you made a copy/paste error and you mean DCL03-C. Use a static assertion to test the value of a constant expression.

I have taken out the macro definition and updated the text to reference that guideline.

I have also reworked the code examples so that they all compile with gcc using the command:
gcc -c -Wall -Wextra -std=c99 -pedantic -Werror [file.c]
with the exception of Compliant Solution 3 because the static_assert macro is undefined.