MEM01-C. Store a new value in pointers immediately after free()

Dangling pointers can lead to exploitable double-free and access-freed-memory vulnerabilities. A simple yet effective way to eliminate dangling pointers and avoid many memory-related vulnerabilities is to set pointers to `NULL` after they are freed or to set them to another valid object.

Noncompliant Code Example

In this noncompliant code example, the type of a message is used to determine how to process the message itself. It is assumed that `message_type` is an integer and `message` is a pointer to an array of characters that were allocated dynamically. If `message_type` equals `value_1`, the message is processed accordingly. A similar operation occurs when `message_type` equals `value_2`. However, if `message_type == value_1` evaluates to true and `message_type == value_2` also evaluates to true, then `message` is freed twice, resulting in a double-free vulnerability.

```
char *message;
int message_type;

/* Initialize message and message_type */

if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
}
/* ...*/
if (message_type == value_2) {
   /* Process message type 2 */
  free(message);
}
```

Compliant Solution

Calling `free()` on a null pointer results in no action being taken by `free()`. Setting `message` to `NULL` after it is freed eliminates the possibility that the `message` pointer can be used to free the same memory more than once.

```
char *message;
int message_type;

/* Initialize message and message_type */

if (message_type == value_1) {
  /* Process message type 1 */
  free(message);
  message = NULL;
}
/* ... */
if (message_type == value_2) {
  /* Process message type 2 */
  free(message);
  message = NULL;
}
```

Exceptions

MEM01-C-EX1: If a nonstatic variable goes out of scope immediately following the `free()`, it is not necessary to clear its value because it is no longer accessible.

```
void foo(void) {
  char *str;
  /* ... */
  free(str);
  return;
}
```

Risk Assessment

Setting pointers to `NULL` or to another valid value after memory is freed is a simple and easily implemented solution for reducing dangling pointers. Dangling pointers can result in freeing memory multiple times or in writing to memory that has already been freed. Both of these problems can lead to an attacker executing arbitrary code with the permissions of the vulnerable process.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM01-C	High	Unlikely	Low	P9	L2

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04		Supported: Astrée reports usage of invalid pointers.
Axivion Bauhaus Suite	6.9.0	**CertC-MEM01**	Fully implemented
CodeSonar	5.2p0	**ALLOC.DF** **ALLOC.UAF**	Double free Use after free
Compass/ROSE			
Coverity	2017.07	**USE_AFTER_FREE**	Can detect the specific instances where memory is deallocated more than once or read/written to the target of a freed pointer
LDRA tool suite	9.7.1	**484 S, 112 D**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-MEM01-a** **CERT_C-MEM01-b** **CERT_C-MEM01-c** **CERT_C-MEM01-d**	Do not use resources that have been freed Always assign a new value to an expression that points to deallocated memory Always assign a new value to global or member variable that points to deallocated memory Always assign a new value to parameter or local variable that points to deallocated memory
Parasoft Insure++			Detects dangling pointers at runtime
Polyspace Bug Finder	R2019b	CERT C: Rec. MEM01-C	Checks for missing reset of a freed pointer (rec. fully covered)

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	VOID MEM01-CPP. Store a valid value in pointers immediately after deallocation
ISO/IEC TR 24772:2013	Dangling References to Stack Frames [DCM] Dangling Reference to Heap [XYK] Off-by-one Error [XZH]
MITRE CWE	CWE-415, Double free CWE-416, Use after free

Bibliography

[Seacord 2013]	Chapter 4, "Dynamic Memory Management"
[Plakosh 2005]