What is Memory Safety

Version 0.9

Date 10/09/2026

1 Introduction

1.1 Purpose

The purpose of this document is to give an introduction to memory safety. Two important aspects of memory safety are spatial safety and temporal safety. This document will discuss both. It aims to help engineers and SoC architects understand the issues, and be in a good position to start thinking about how to build systems that are memory safe.

This document is one of a series, each providing more detail on specific aspects. For anyone learning about CHERI, we suggest that all the documents in the series are reviewed. Please don't hesitate to share suggestions or improvements - we see this as a collaborative effort to advance CHERI technology together.

Memory safety has long been one of the most persistent and costly challenges in systems programming. While spatial safety failures (like buffer overflows) tend to attract the most attention, temporal safety violations are every bit as dangerous and often far more subtle.

If you're comfortable with how CPUs can use pointers to read and write system memory, you've got all the prerequisites you need for the rest of the document.

1.2 Memory safety in standards and regulation

Memory safety is no longer only a software engineering concern. The prevalence of memory-safety vulnerabilities in deployed products has led governments, standards bodies, and industry groups to examine how such weaknesses can be reduced or eliminated.

To support a common understanding of the problem, ETSI is developing **TS 104 198, Cyber Security (CYBER); Memory safety requirements**. This is a technical specification that provides a standard framework for discussing memory-safety weaknesses and the technologies used to address them. The specification is intended to help technology developers, product manufacturers, evaluators, and policymakers understand the strengths and limitations of different memory-safety approaches. The current document is not intended to reproduce or replace that specification. Instead, it aims to provide an introduction to the topic.

1.3 Terminology

Allocation of memory is when software requests an amount of memory to be allocated for use.

Deallocation or **freeing** of memory is when software no longer needs some previously allocated memory, and releases it back to the pool so it can be used for something else.

Reallocation of memory is when software requests an increase in size of some memory that has been previously allocated, or when memory that has previously been freed is allocated again.

A **pointer** is basically a memory address stored as a few bytes of data in memory. The value of that data can be used to reference (or point to) some other part of memory for reading or writing. A **dangling pointer** is a pointer that references memory that has been freed or repurposed, making any access unsafe and unpredictable.

Spatial safety concerns what parts of memory can be accessed. It is the property that a program only accesses memory within the bounds of the objects it is meant to use. When spatial safety holds, every pointer refers to a specific region of memory, and the program never reads or writes outside that region. When it doesn't hold, the program can stray into memory that was never intended to be touched.

Temporal safety concerns when a memory object is accessed rather than where. Even if a pointer refers to the correct region of memory, it may still be used at the wrong time. Two of the most common examples are use-after-free and use-after-reallocation hazards.

1.4 Content

1.5 Illustrations

2 What Is Memory Safety All About?

Before we look at memory safety and the difference between temporal and spatial safety, it is useful to have a look at the memory life cycle, and understand how memory is normally used.

2.1 Basic Memory Life Cycle

Memory within a system is a finite resource. Therefore, in almost all systems, memory will be allocated, deallocated and, at some point in the future, allocated again. This life cycle is shown in Figure 1.

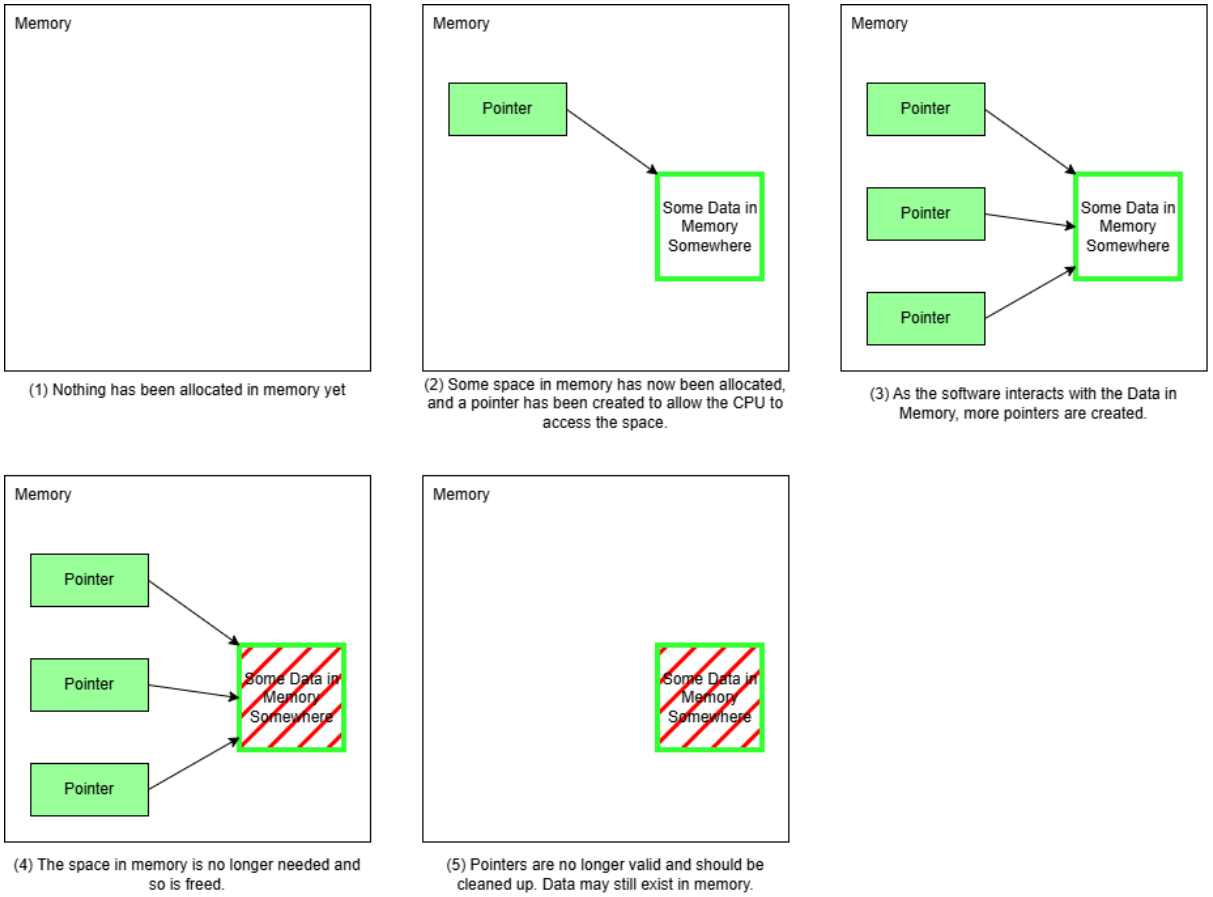

Figure 1 – Basic Memory Life Cycle

Phase 1: Initially the memory will be empty. Nothing has been allocated in the memory space and there are no pointers. At some point, some memory allocation will occur. If writing code in "C" this could mean calling a memory allocation function (such as "malloc"), effectively asking for a pointer to a number of bytes of space in the memory.

Phase 2: If enough space exists in the memory system, the memory allocator will allocate some space and return a pointer to that allocated space. That pointer will also exist in memory.

Phase 3: The software will use the allocated space for a period of time. During this time more pointers to the space may be created and shared with other bits of software running in the system. Some pointers may point to the start of the allocation, while others may point to specific regions within.

Phase 4: The software no longer needs the space and so requests that the space be deallocated. If coding in "C" this could mean calling the "free" function. When the deallocation is done, it means that the space is now available to be allocated to software that requests some space.

Phase 5: It is good practice to clean up any pointers that still exist and point to the deallocated region in memory. If these pointers were not cleaned up and remained in the system after the memory they pointed to was freed, they would be called dangling pointers. As we will see later, dangling pointers can cause system issues including instability and vulnerability.

2.2 Memory Life Cycle with Free and Reallocation

Over time the memory that was freed in Figure 1 will end up being recycled and used again. This is shown in Figure 2.

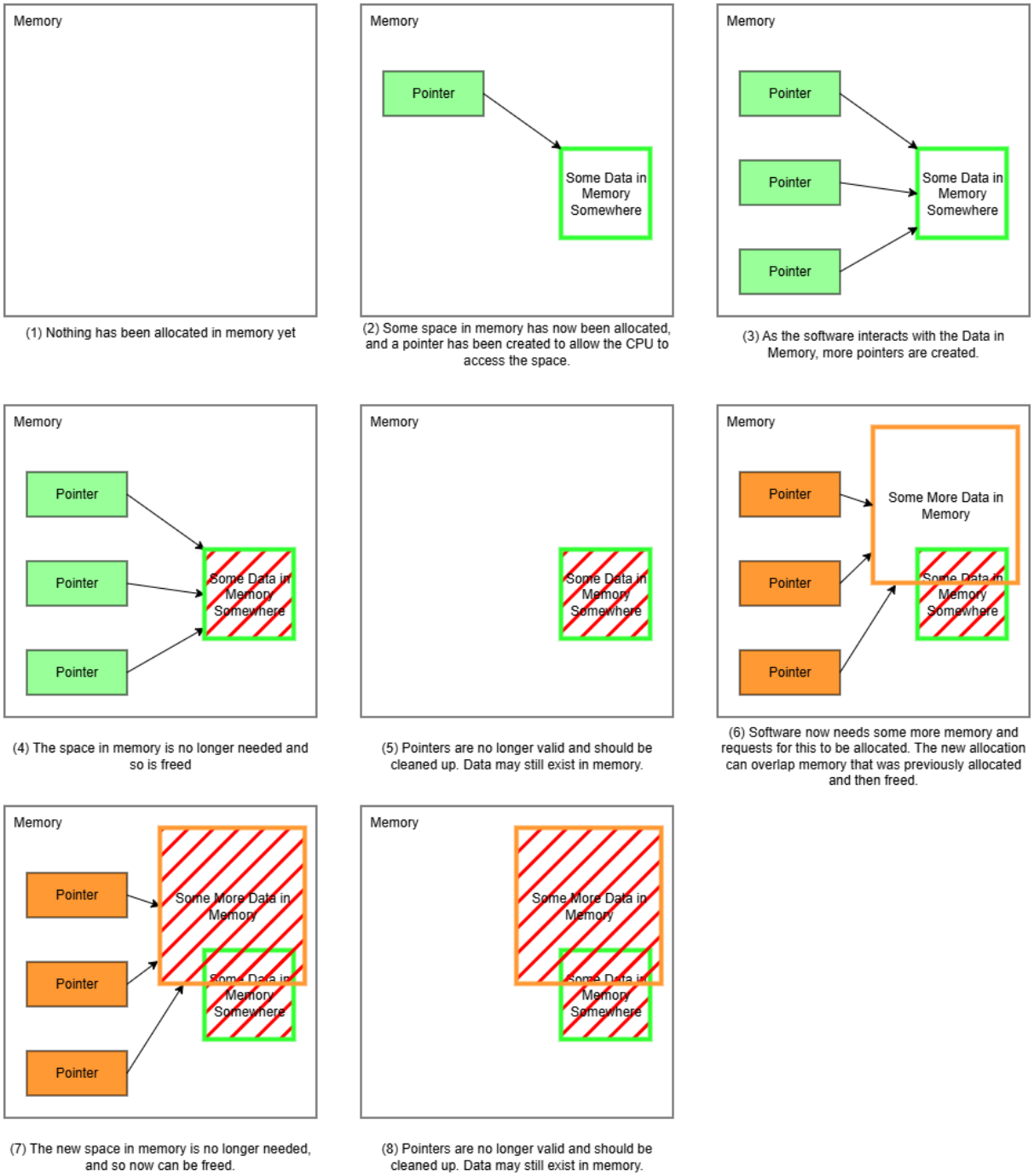

Figure 2 – Memory Life Cycle with Free and Reallocation

The first five phases of the life cycle are the same as described previously.

Phase 6: The software requests some more memory to be allocated. Assuming enough memory is available, the allocation happens, and may include the allocation of memory that has previously been allocated and freed. There may well be data in the memory from

that previous use. In this case the software has been well written, and there are no dangling pointers into the allocated memory, so the thread that requested the allocation is the only one that is reading or writing from that space.

Phase 7: The memory is no longer required by the software, and so it is freed.

Phase 8: Any pointers that were being used to read or write the memory space are cleaned up. No dangling pointers are left.

2.3 Memory Life Cycle with Reallocation (Extending Available Space)

In this variation of the life cycle, the software requests some memory to be allocated, which it is. Later the software realizes that it needs more memory than was originally requested. It therefore requests a reallocation. In "C" this would be done by calling the "realloc" function. This process is shown in Figure 3.

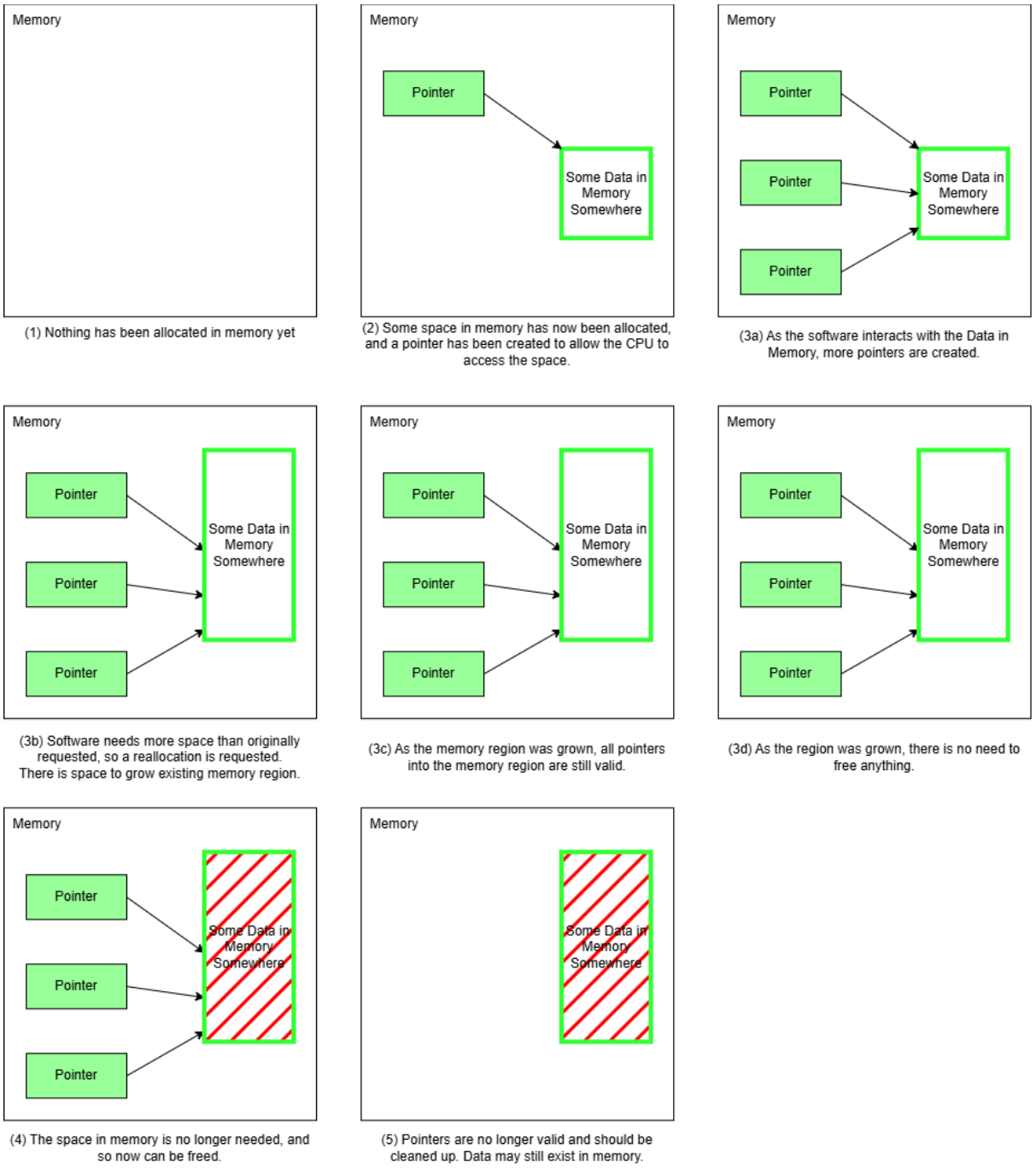

Figure 3 – Memory Life Cycle with Basic Reallocation

Phase 1 and 2 are the same as in the previous description.

Phase 3a: The software is running, more pointers have been created, and at some point the software realizes it needs more memory than was originally requested.

Phase 3b: The software requests the reallocation (in "C" this is done by calling the "realloc" function). In this case there is space available in the memory adjacent to the currently allocated region. This means the reallocator can just allocate more memory and grow the existing region.

Phase 3c: As the memory region has not moved, it has just grown, all pointers into the memory region are still valid.

Phase 3d: As the memory region has not moved, it has just grown, there is no need to free anything.

Phase 4 and 5 are the same as in the previous description, just operating on a larger memory space.

2.4 Memory Life Cycle with Reallocation (Relocating and Extending Available Space)

In this variation, the software again realizes it needs more memory than originally requested. It again requests a reallocation, but this time there is not enough free memory adjacent to the existing memory region to satisfy the request. Therefore the memory reallocator has to allocate a new, larger region somewhere else, and then copy the existing data to that new location. This process is shown in Figure 4.

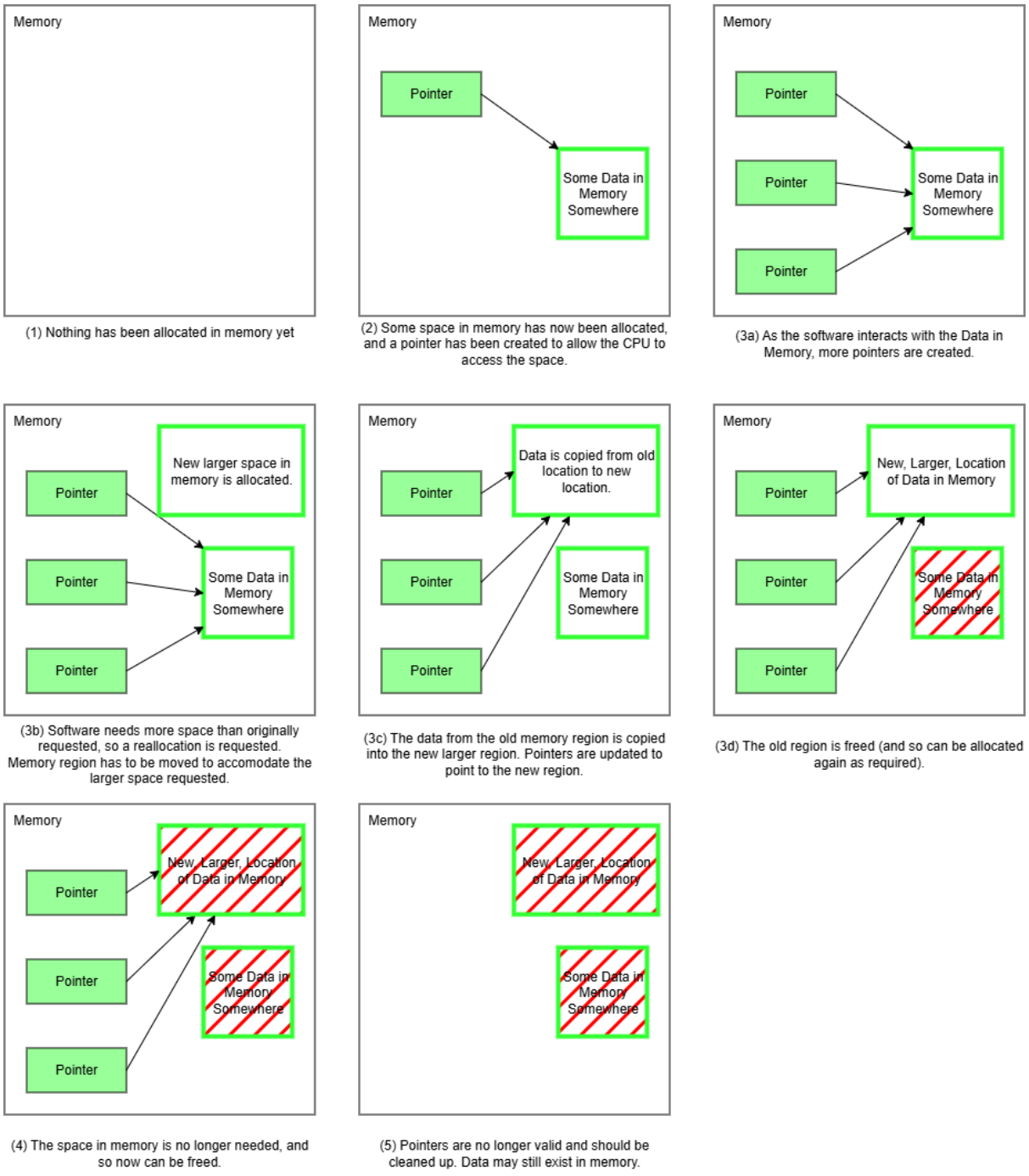

Figure 4 – Memory Life Cycle with Basic Reallocation

Phase 1 and 2 are the same as in the previous descriptions.

Phase 3a: This is the same as in the previous description. The software runs as normal, more pointers are created along the way, and at some point the software realizes that more memory is needed.

Phase 3b: The software requests the reallocation. In this case there is no space available in the memory adjacent to the currently allocated region. This means the reallocator must find a new location in which to allocate the entirety of the new and larger space.

Phase 3c: The reallocator must copy the data from the old space to the new space. Once this is done, software will need to update pointers into the old memory space to point to the equivalent location in the new memory space.

Phase 3d: The old (smaller) region in memory can now be freed and returned to available memory for future allocations.

Phase 4 and 5 follow the same principles as in the previous descriptions.

2.5 Spatial Safety

Spatial safety is about making sure software running on a CPU can only access the memory that it is authorized to access. ETSI TS 104 198 classifies weaknesses such as buffer overflows, out-of-bounds reads, and out-of-bounds writes as examples of spatial memory-safety failures.

If a system has strong compartmentalization, then it will offer some spatial safety guarantees. The granularity of the compartmentalization will be proportional to the level of safety offered.

If a system does not offer spatial safety then it is possible for a CPU to create a pointer to some memory that it should not have access to, and then use that pointer to access the contents. This can have many consequences (from data corruption and causing the system to behave in an unexpected way, to revealing secrets that should not be shared).

Figure 5 shows an example of a spatial safety issue.

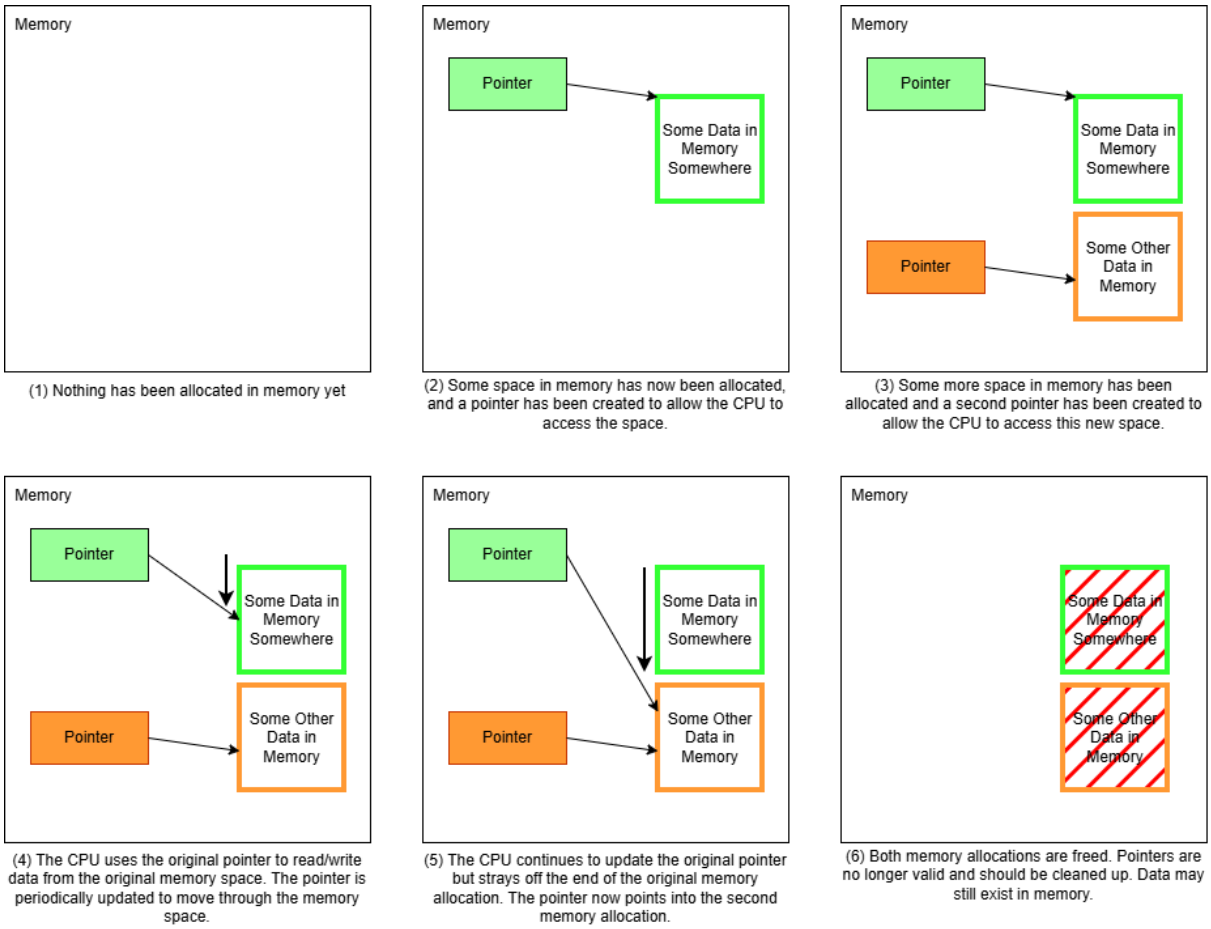

Figure 5 – Example of Spatial Safety Issue

Phase 1 and 2 are the same as in the previous descriptions.

Phase 3: Another region of memory is allocated (possibly to another thread running on the machine).

Phase 4: The original thread is (perfectly legally) accessing memory from the buffer that has been allocated for it (shown in green on the diagram). It is working its way through the buffer, updating the pointer as it goes.

Phase 5: The original thread has continued to increment the pointer, even past the end of the allocated memory space. The pointer is now pointing inside the other thread's allocated

memory (shown in orange on the diagram). This could allow data leakage between threads, or corruption of data if both threads are writing to the memory region.

Phase 6: At some point both regions of memory will have been freed and pointers should have been cleaned up.

2.6 Temporal Safety

Temporal safety is about time: software should only be able to access a space in memory while it is authorized to do so.

There are two main hazards related to temporal safety: use-after-free and use-after-reallocation.

In a use-after-free scenario, software continues to use a pointer after the memory it refers to has been deallocated. The data that this dangling pointer points at will be stale, with no guarantees of correctness; in fact, the data will very likely be incorrect. Using the dangling pointer could also corrupt data that another thread is legitimately using, and can lead to unpredictable and erroneous behavior.

In a use-after-reallocation case, the memory has been reallocated. This could mean either that an existing buffer has been moved and resized (the move resulting in the existence of a dangling pointer) or that a pre-existing dangling pointer is providing access into some memory that has been recently allocated. This is a temporal safety problem because the old pointer survives across a change in the lifetime or identity of the object it used to refer to.

In either of these scenarios, if the memory that the dangling pointer references is subsequently allocated again, the dangling pointer will be able to provide access to any new live objects now living within that memory space. The objects will have potentially incompatible semantics, and this is another mechanism that can lead to unpredictable behavior and data corruption. Additionally, this situation could lead to secrets leaking from the new object to any thread still using the dangling pointer.

3 Example of the Hazards

3.1 Use-After-Free Hazard

For a system to suffer from a UAF hazard it must have ended up in a state with a dangling pointer. Such a state of affairs is shown in Figure 6.

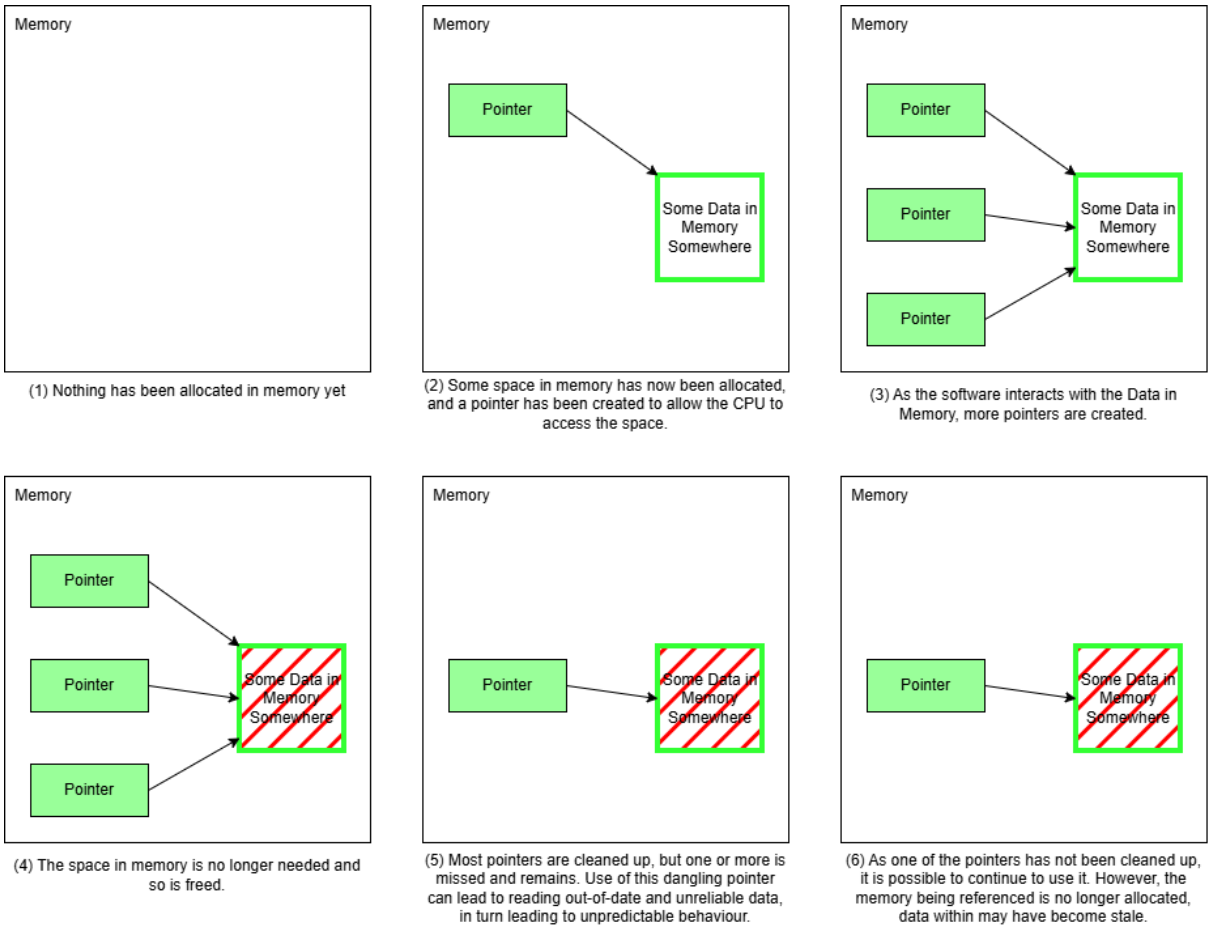

Figure 6 – Example of Use-After-Free (UAF)

Phases 1 to 4 are the same as in the previous descriptions.

Phase 5: It is good practice to clean up any pointers that still exist and point to the deallocated region in memory. However, in practice, it is all too easy to leave a dangling pointer in existence after the associated region of memory has been deallocated. In this example, this is what has happened.

Phase 6: This dangling pointer can then be used to access the deallocated region. The data being accessed will not be valid, and use of this data will lead to unexpected behavior. In any system, unexpected behavior is not useful, and could lead to many problems (including security vulnerabilities).

3.2 Use-After-Reallocation Hazard (Example 1)

If the region of memory has been fully or partially reallocated this becomes a UAR hazard, which is described in Figure 7.

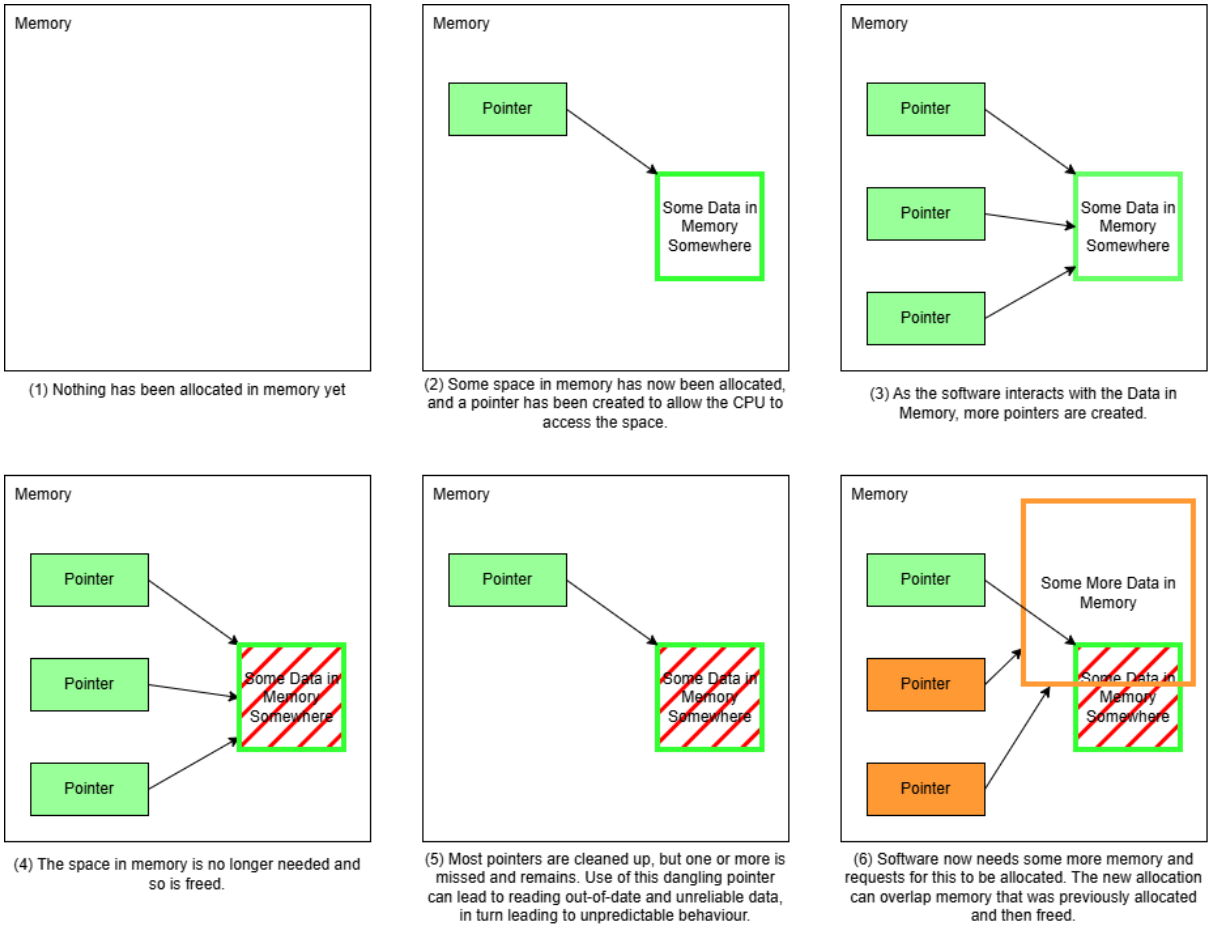

Figure 7 – Example of Use-After-Reallocation (UAR)

Phases 1 to 4 are the same as in previous descriptions.

Phase 5: One of the pointers is not cleaned up, and remains as a dangling pointer within the system.

Phase 6: A (potentially different) thread requests some memory to be allocated, and this results in it having access to some memory that overlaps the previously freed memory. The dangling pointer now points inside this newly allocated region, allowing unauthorized access to the original thread.

If the original thread reads data from the newly allocated region, it is very likely to lead to problems with its operation, as the data will not be as expected. Secrets could also be revealed, and vulnerabilities could be facilitated.

Writing data to this region will cause wider system issues, as other threads in the system (who do have proper authorization to read or write that same memory) will experience data corruption.

3.3 Use-After-Reallocation Hazard (Example 2)

UAR hazards can also be seen when software requests to increase the size of a memory allocation. This scenario is shown in Figure 8.

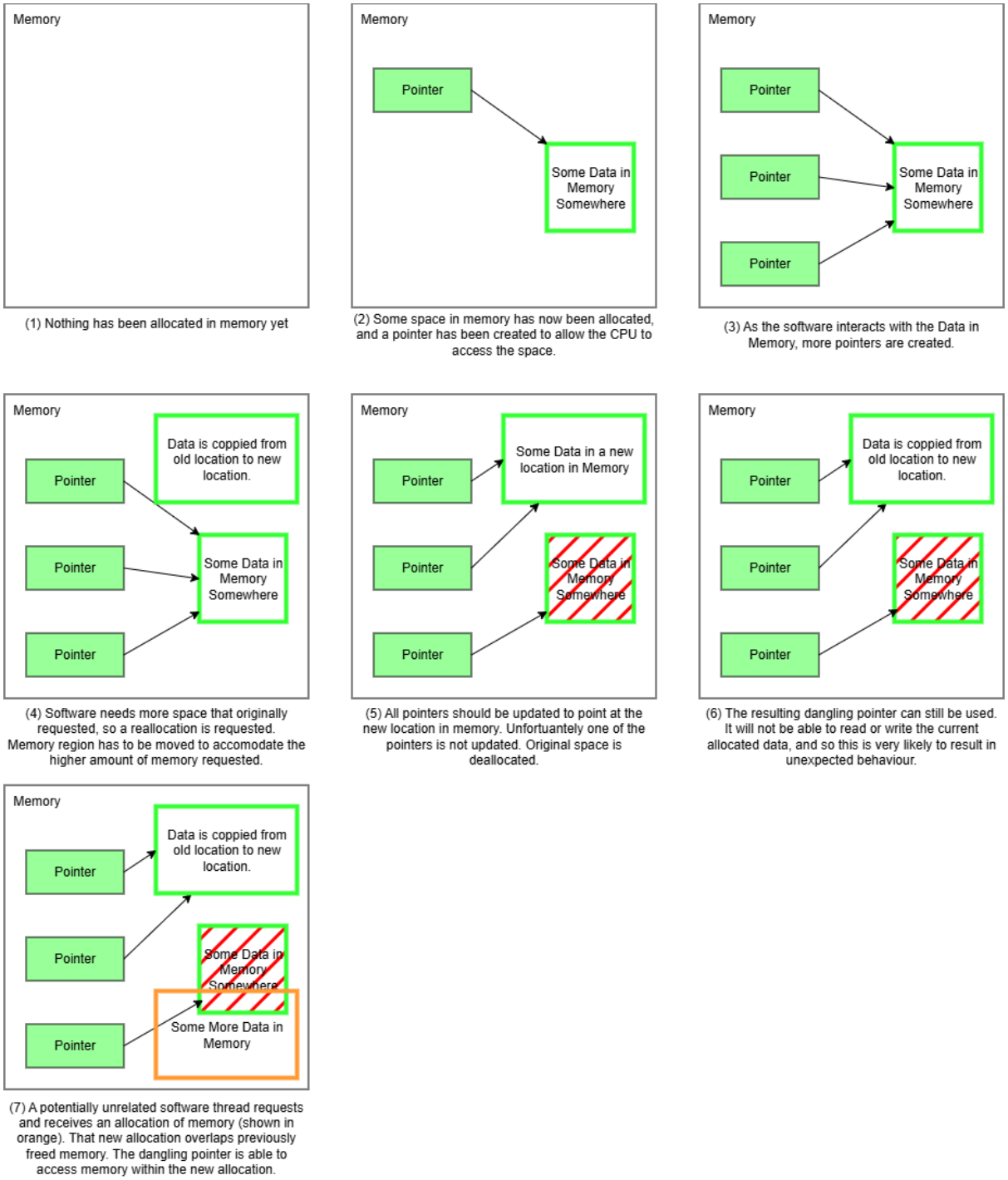

Figure 8 – Another example of Use-After-Reallocation (UAR)

Phases 1 to 4 are the same as in the previous descriptions.

Phase 5: One of the pointers is not updated to point towards the reallocated memory space. It becomes a dangling pointer.

Phase 6: This dangling pointer can then be used to access the deallocated region. The data being accessed will not be valid, and use of this data will lead to unexpected behavior. In any system, unexpected behavior is not useful, and could lead to many problems (including security vulnerabilities).

Phase 7: At some point an unrelated software thread receives an allocation of memory that overlaps with the target of the dangling pointer. When this happens the unrelated software thread could experience memory corruption. This could lead to unexpected behavior not just from the original thread but now a second thread as well. Secrets could again be revealed, and vulnerabilities could again be enabled.

4 References

ETSI TS 104 198, *Cyber Security (CYBER); Memory safety requirements*.