Correction indications for Software Security Mid-Term 2025-26

Exercise 1 (5 pts)

Q1. (4 pts)

Calling function "foo" with base=1 and offset=INT_MAX will produce an arithmetic overflow at line 11 when computing base+offset. In case of wrap-around (the default behavior at the processor level) the result will be -1 when interpreted as a signed value (int).

- When the code is compiled without optimization nor canaries:
The check performed at line 12 returns true, following the default processor behavior, with index equals to -1. The execution then stops at line 13.

- When the code is compiled without optimization but with canaries:
The program behaves as previously, the canaries not changing the way arithmetic operations are performed.

- When the code is compiled with optimization but without canaries:
The optimizer will try to take advantage of the potential undefined behavior. Here, the underflow can be predicted statically since:
- base is assumed to be strictly positive between lines 7 and 20
- offset is assumed to be strictly positive between lines 11 and 22

At line 11, index *should be* strictly positive, unless an overflow occurs. Therefore lines 12 to 15 are dead code and can be omitted for code generation. Hence, at line 20, index will be equal to -1 (default processor behavior) and a memory will occur, trapped by the OS.

- When the code is compiled with optimization and without canaries:
The use of canaries only prevents for memory errors allowing to rewrite the stack cell above the return address. With index being equal to -1 (as above), the address accessed will be outside the valid address range allocated to the program and the execution will be stopped by the OS itself (leading to a seg. fault).

Q2. (1 pt)

Even though the user can control base and offset, the only unexpected behavior he may produce is to make index strictly negative at line 11. Then the check performed at line 12-15 can be bypassed (in case of optimizations), but not the one at line 16. Hence, the index used at line 20 will be either in [0,10[or negative (leading to an error trapped by the OS).

The only attacker benefit will therefore be a "denial of service".

To correct this problem, we can for instance (other solutions being also probably suitable):
- use the CERT secure coding rule to properly check for arithmetic overflows;
- or declare index, base and offsets are "unsigned int"

Exercise 2 (8 pts)

Q1. (2 pts)

Function main() reads a string from the user/attacker and stores it a stack-allocated buffer without checking its size, an overflow may then occur.

1. This overflow may be exploited to overwrite the return address with the address of a ROP-chain of gadgets

2. This overflow may also be exploited to change the local variable uid, which could give privileged accesses to the user in some contexts (although it is not the case with the code example provided in the subject)

Q2. (2 pts)

Using SafeStack protection the buffer is not stored in the same stack than the return address. Therefore:

- the ROP attack cannot occur anymore ...
- the uid variable cannot be overwritten any more

Q3. (2 pts)

Using StackProtector option (aka canaries), the buffer is located at the bottom of the stack frame, just above the canary. Therefore:

- the return address cannot be overwritten without altering the canary (assuming its value has not been disclosed), preventing any ROP attacks
- variable uid cannot be overwritten, since it lies above the buffer

Q4. (1 pt)

Function bar() is now executed. During its execution previous answers to Q2 and Q3 are unchanged. However, the call to function gets() in bar() may overwrite the buffer declared in function main().

Q5. (1 pt)

Comparing SafeStack (SS) and StackProtector (SP):

- *protection level:*

SP: not secure in case of canary disclosure or if the canary an be "skipped" when the buffer is assigned, and do not protect against buffer over-read (but only against buffer over-write)
SS: more secure than SP assuming a clear separation between the 2 stacks (no risk to overwrite a return address).
Note that both protections do not prevent against:

- overwriting from one buffer to another
- crashes (i.e., DoS) due to overwriting of invalid memory addresses (segfault raised by the OS
- re-using data left in memory by a preceding function call

- *memory overhead:*

SP: very low, only needs to store the canary
SS: a bit higher, needs to allocate and manage a second stack

- *time overhead:*

SP: very low, set (resp. check) the canary value at function call (resp. return)
SS: a bit higher, needs to allocate and manage a second stack
(more code to execute at each function call/return)

Exercise 3 (7 pts)

Q1. (2 pts) function non_ct_lookup

1. when only index is secret, an exception raised to an index error can leak information about the index (being negative or greater than size). If the size can be assigned by the attacker he may also get the exact value of the index using such exceptions.
Moreover, execution time may leak information about the index accessed through the data cache.

2. when only the size is secret, an exception raised due to an index error can also leak information about the table size. Of course, if the user may also control (i.e. assign) the index he may find the index using brute-force.

Assuming all values in the table are different, another option to get the index is look at the table content to know which index corresponds to the returned result. Without this assumption we only get a set of possible index ...

Q2. (2 pts) function ct_lookup

1. Attacks through the cache are no longer valid. Alternatively, increasing size will give directly the index by consulting the returned value when it becomes different from 0.

2. The only very partial information which may leak about the size came from the overall execution time (the larger it is, the larger is size ...)

Q3. (2 pts)

1. If the time used to execute line 10 may change, timing attacks may now give information about the index accessed by iterating over successive values of size (when the average time per iteration changes we know that size==index ...).

2. No change from Q2.

Q4. (1 pt)

Using static type-checking we would like to detect unwanted flows from confidential to public data. In statement "return table[index]" there is such *potential* flows between:
- the content of the table and the index
- the size of the table and the index

This code will be therefore flagged as "potentially leaking" by a static type-checker ...