TD 7: dynamic/static analysis

Get the TD tarball from
`https://dept-info.labri.fr/~thibault/SecuLog/td7.tgz`
unpack it, run `make`

1 Dangerous functions

Look at `getpw.c` which calls `getpw`, compile it. The linker warns that this function is dangerous, why ? Try to reduce the size and see that it segfaults on return, explain how that happens.

The *gecos* field is an information that users can usually change at will for their own account with the `chfn` command (but that's disabled at CREMI). Explain how that could be exploited.

`man 3 mktemp` and the linker say that this function is dangerous and `mkstemp` should be used instead, why ? Look at `mktemp.c` to confirm your suspicion.

2 Memory leak reporting

2.1 Definite loss

Look at `test-leak.c`. Try to run
`cppcheck test-leak.c`
See that the cppcheck static analyzer could spot one leak location, the most obvious one.
Now try
`valgrind ./test-leak`
Valgrind also spotted both leaks. It doesn't enable precise tracking by default, enable it :
`valgrind --leak-check=full ./test-leak`
There it tells you which allocations were never freed. Notice that this is a *backtrace* : at the top of the trace it shows you that it's a `malloc` call which was never matched with a `free`, and just below it we see the actual function that did the `malloc` call. This does not necessarily mean that it's the culprit ! Check further in the trace and in the source code to see what function was supposed to free the data. If the pointer was transferred via global variables etc. it'd be even less clear. But at least valgrind tells in which context it was allocated.

Also try to run
`./test-leak.asan`
It also spots the leak locations.
Why can such a memory leak be a security concern ?

2.2 Still reachable loss

Look at `test-reachable-leak.c`
This is strictly speaking also an improper memory cleanup, but the pointer is still reachable, which makes a difference. Check whether `cppcheck` and `asan` report it. Check `valgrind`, see that for the full report an additional option is needed :

```
valgrind --leak-check=full --show-leak-kinds=all ./test-reachable-leak
```

3 Uninitialized values

Look at `test-uninitialized.c`. Since `c[0]` is inside a `malloc`-ed area, it has an uninitialized value. Try to run it, it will probably print `foo is 0`, but there is really no guarantee that it does so. Indeed, try to first set

```
export MALLOC_PERTURB_=123
```

and run it again. See `man mallopt` for the details (use `/` to look for something in the manual).

Check whether `cppcheck` and `asan` report it. Check `valgrind`, see how the report looks like : it points at where the value is *actually* used, not where it was allocated. As seen in the course, when `f` was called, valgrind simply passed over the information that the parameter was uninitialized. Fortunately, see that valgrind tells that `--track-origins=yes` can be used to track the origin (note however that this is quite expensive).

Now look at `test-uninitialized-printf.c` and check what `valgrind` says about it. It spits out a lot of warnings about printfs, but the real culprit is our program !

Note that beyond the problem of erratic behavior, this can potentially leak information that was previously stored in the area that was just allocated !

4 Undefined behavior

Look at `test-undefined.c`, it is trying to compute `INT_MAX+1`, which is an integer overflow and thus undefined behavior. Try to run the program, see that it indeed produces a negative value ! Try to run through valgrind, it does not report anything : it just sees an `incl` instruction, it does not have the source code, so it does not know whether it is actually an `unsigned` or a `int` variable.

Try the `usan`-built `test-undefined.usan`, see that since it's the compiler that injects usan checks, it does know the type and could thus introduce the appropriate check.

5 Buffer overflow

5.1 Heap buffer overflow

Look at `test-overrun.c`. Try to run it, see that it just "works" fine ! But that's only because `malloc` rounds up the allocation size, and thus the overflow happens to be non-fatal by pure luck !

See how `cppcheck`, `asan`, `usan`, and `valgrind` report the error (or not). Explain closely what their formulation mean ("`0 bytes after a block of size 10 alloc'd`"), and for each tool see how you get to know the origin of the allocation.

5.2 Static buffer overflow

Look at `test-overrun-data.c`, it's basically the same, but in the data segment. Why only `usan` and `cppcheck` manage to spot the overflow ?

5.3 Stack buffer overflow

Look at `test-overrun-stack.c`, again it's the same, but on the stack. Why `asan` is now able to spot it, but still not valgrind ?

6 Mutex ordering (hors-programme)

Look at `test-mutex-order.c`, we have two threads which take the same pair of locks, but in different order, so this can lead to a deadlock. See how `tsan` and

`valgrind --tool=helgrind ./test-mutex-order`

report it. Explain how to understand their reports. Notice that we did not have to actually get a deadlock for them to spot the issue, the tools just happened to observe the differing order.

Notice that `cppcheck` is not able to spot it.

7 Race conditions (hors-programme)

Look at `test-race.c`, we have two threads writing to the same variable without any protection! See how `tsan` and `helgrind` report it as well, and how to understand their reports.

Again, `cppcheck` cannot spot it.

8 Odd expressions

— Look at `test-expr.c`, do you see the bug?
Run `make clean` and `make test-expr`, see that `test-expr.c` gets a build warning. Try the program, see that it indeed misbehaves. Understand the bug, fix it. This looks quite trivial, but we do see such bugs staying for years in e.g. Linux drivers :/

Note : in the following coverity reports, you do not need to understand what the software is about, the snippet is enough to understand what is getting wrong. Also, the blue-background horizontal stripes are not part of the code, they just show a scenario that the coverity tool found that raises a concern. Quite often the very first stripes are not interesting, they for instance just mean that we don't exit the function early. Start looking at the salmon stripes, possibly the yellow stripes, and only if needed look back in the time at the blue stripes.

— Look at `https://scan.coverity.com/o/oss_success_stories/57`
Why line 774 cannot be reached? (and thus no error ever reported!)
What are the advantages and drawbacks of using an assignment as a truth value, for instance in this particular case?
— Look at `https://scan.coverity.com/o/oss_success_stories/73` and check `man assert`
Why can `assert` be actually a dangerous thing according to this issue?
Why in this case it happens not to be a concern, but just by luck?
— Look at `https://scan.coverity.com/o/oss_success_stories/86`
How could the static analysis find out that there was an issue here?
— Look at `https://scan.coverity.com/o/oss_success_stories/87`
What is happening during execution? (no need to look back, focus on the loop itself). What is the fix? Why did this only rarely happen to pose problem in practice? (and thus never fixed before)

9 Control flow

— Look at `https://scan.coverity.com/o/oss_success_stories/59`
What is the actual issue? Why is the issue reported as "dead code"?
— Look at `https://scan.coverity.com/o/oss_success_stories/62`
Why is the analyzer saying that the value is unused?
— Look at `https://scan.coverity.com/o/oss_success_stories/97`
Imagine what kinds of horrible things could happen with such a "case fallthrough".

When you have time, also read the story of a whole telephone network collapse due to a misplaced `break`: `http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/att_collapse.html`

— Look at `https://scan.coverity.com/o/oss_success_stories/99`
What is the terrible consequence of this error?

— Look at `https://scan.coverity.com/o/oss_success_stories/93`
How is it that it cannot be reached?

10 Damn pointers...

— Look at `https://scan.coverity.com/o/oss_success_stories/64`
How can the analyzer be sure that there is a memory leak?

— Look at `https://scan.coverity.com/o/oss_success_stories/100`
Why is this code wrong?

— Look at `https://scan.coverity.com/o/oss_success_stories/66`
What is the contradiction?

— Look at `https://scan.coverity.com/o/oss_success_stories/68`
Why can the analyzer spot that `pos` cannot be 0?
What is the difference between the current code and the proposed fix?

— Look at `https://scan.coverity.com/o/oss_success_stories/51`
What is the difference between the current code and the proposed fix?

— Look at `https://scan.coverity.com/o/oss_success_stories/88`
What was supposed to happen, and what is actually happening? What fix should be done? Why can't we simplify this down to

```
for (cur_module = wmodules; cur_module; cur_module = cur_module->next) {
   cur_module->context->destroy(cur_module->data);
   free(cur_module);
}
```

?

11 Error checking

— Look at `https://scan.coverity.com/o/oss_success_stories/69`
See that the tool apparently learnt from the rest of the code that it is a good idea to check the return value.

— Look at `https://scan.coverity.com/o/oss_success_stories/82`
What do they mean by "tainted string"? (the `dbfcmd()` function sends the string as a command to the SQL database server, and was thus explicitly marked not to be fed with tainted data)
What should be done here?