LINGI2144: Secured System Engineering
Tutorial 2: Memory Safety

Academic year : 2020 - 2021

Teacher : Legay Axel
Course : LINGI2144
Collaborators :
Crochet Christophe
Duchene Fabien
Given-Wilson Thomas
Strebelle Sebastien

1 Prerequisite

Working directory: `~/SecurityClass/Tutorial-02`

Connection:

username	password
admin	nimda

2 Exercise

2.1 Spacial Bounds

Let's look at the `get-n.c` file. This is a simple utility to return the nth character of a string given to the utility. Let's build and test it under normal usage:

```
gcc -o get-n get-n.c
./get-n 0123456789 4
./get-n abcdefghij 8
```

What do you think will happen if we give some different numbers as the second argument?

- Hint: Try a few different numbers of very different values, e.g. 100, 1000, 10000, -4.

How would you fix this code?

Now let's look at `my-cut.c` that acts like the cut utility by return the characters in a range. We can build and test with

```
gcc -o my-cut my-cut.c
./my-cut 0123456789 2 7
./my-cut abcdefghij 3 6
```

What do you expect to see now if we give some very different numbers for the second and third arguments?

- Hint: See how much you can see before you cause a segmentation fault.

How would you fix this code?

From this we can see that memory safety is not only a serious concern, but we can even see things well outside the memory space we should be looking at. The examples above only read this space, but we could of course write into this space in most cases, leading to potentially serious vulnerabilities.

2.2 Use After Free (adapted from lectures)

Let's revisit the use after free example from the lectures. Let's look at `simple.c`. What do we expect to happen if we compile and run it?

```
gcc -o simple simple.c
       ./simple
```

When we use the memory after the free we have a segmentation fault (we cannot use this memory segment, it's been freed)! But if we add in a new allocation as shown in `after-free.c` that can reuse the same memory segment this changes

```
gcc -o after-free after-free.c
./after-free
```

Now we are able to inject the bad function behaviour into the "legitimate" function memory space of `"malloc1"`. As described in the lecture, the (re)allocation of memory has allowed us to replace the `"malloc1"` function with the "bad" function via `malloc2`.

This works because the memory is immediately reused. But we can break this by allocating another segment of memory in between. Have a look at the code for `again.c`. **How do you think this will behave? Why?** We can check the behaviour as usual with:

```
gcc -o again again.c
./again
```

A similar program is `yet-again.c` that has a slight change in the allocations of memory. **What do you expect to happen now? Why?** This can be checked with:

```
gcc -o yet-again yet-again.c
./yet-again
```

2.3 Double Free

Let's look at some other ways that freeing memory can be exploited. Observe that before the same memory was reallocated. (The memory management doesn't want to keep creating new memory segments, instead giving one that was "freed" back to the same program again when the program asks for another small segment of about the same size.)

Look at the code double-free.c that allocates and frees some memory. Observe that the memory allocated to "a" is freed twice. **What do you think will happen?** We can observe some output with

```
./double-free
```

See that despite never assigning anything to "f" initially, we are still able to see a well formatted output. Similarly, editing "f" has effects on "d", despite them (supposedly) being different memory chunks.

WARNING: The binary here is provided since the version of the memory management libraries on the system fixes this problem. (The binary is statically linked which makes it much bigger than a normal binary.)

We can see that the problem is now fixed on this system by compiling and checking:

```
gcc -o fixed-double-free double-free.c
./fixed-double-free
```

This is also true if we use another compiler:

```
clang -o clang-double-free double-free.c
./clang-double-free
```

Here this fix is handled by the memory allocation and management libraries on the system.

2.4 Uninitialised Pointers and compiler support

Notice that for 2.2 and 2.3 you were not asked to try and find a solution to this problem. This is because the "solutions" are very compelled or rely on tools and heavy instrumentation (that we will look at later).

Let's look at `uninit.c` that defines a pointer and then prints the string at that pointer.

```
gcc -o uninit uninit.c
./uninit
```

This prints out some part of memory that we never initialised. Clearly this is not the "correct" way to use a pointer, but the compiler doesn't care.

This code is very simple, so we can ask the compiler to give us all warnings and detect this:

```
gcc -Wall -o uninit uninit.c
```

Now we see that the uninitialisd variable "a" is detected by the compiler. However, we also have a warning about not returning an int from main which we don't care about.

Let's see if `gcc` can find any problems with `double-free.c`

```
gcc -Wall -o double-free double-free.c
```

Here `gcc` can detect the uninitialised variables, but not the double free of "a".

However, this is a limitation of `gcc`, other compilers (or tools) can detect this at compile time. Let's look at `clang`:

```
clang -Wextra -Wall --analyze -o double-free.xml double-free.c
```

The above starts to show the limitations of current (widely used) tools for detecting some of the vulnerabilities that we know and understand well. Of course there are other more advanced (and possibly more expensive) tools that do better, but most of them will not catch complex vulnerabilities that combine control ow and unexpected conditions with basic vulnerabilities like we looked at here.

2.5 `NULL` Pointers

Another approach used in many languages is to not allow any pointer to be "uninitialized" to random values or addresses. Instead, the language (or compiler) sets all pointers to the `NULL` value (typically 0) when they are defined.

Let's look at the code in stupid-null.c that re-uses out functions from 2.2 and some function pointers. Clearly the second "bad" function pointer will be `NULL` when we try to call the function. Let's see what this does

```
gcc -o stupid-null stupid-null.c
./stupid-null
```

As we can see trying to use `NULL` as a function doesn't work very well.

This example is stupid and trivial, but notice that even with all warnings `gcc` doesn't detect this!

```
gcc -Wall -o stupid-null stupid-null.c
```

We can see that `clang` does a little better than `gcc` at detecting this:

```
clang -Wall -Wextra --analyze -o stupid-null.xml stupid-null.c
```

Now let's look at some code that is a little more complex in null-use.c. **Can you work out what this code does and how it breaks?** Again we can build and run as usual:

```
gcc -Wall -o null-use null-use.c
./null-use
```

Observe that gcc does not report any warnings for this code, but when we run it we can see evidence of a memory capture or use after free. Further, even when an improved "free exploitable" function is used to (better) clean up on free, this does not prevent a different exploit (**null** dereference.

Note that in the code for null-use.c pointers are generally initialised to NULL, and set to NULL upon free inside objects. Also the printing and freeing functions check their argument is not NULL before proceeding. This is the beginnings of standard coding practices in many languages to try and avoid the problems illustrated here.

You can check this code with clang and see that the only warning clang can find is in the "free exploitable" function. Note that clang detects that the argument to "free exploitable" could be uninitialized, but not that the function pointer in "bad1" could be uninitialized.

```
clang -Wall -Wextra --analyze -o null-use.xml null-use.c
```

2.6 Open Exercises

Observe that for 2.2 to 2.5 there is no "solve this" type exercises to do. The goal of these exercises is for you to understand the simple concepts underneath many kinds of security vulnerabilities and how they work. In practice such errors in software are excellent ways to gain access to areas of memory that should not be available, or to execute code that should not be executed.

The rest of this tutorial (after this section) has three parts. 2.7 Used a fuzzing tool and should be able to be completed in the tutorial. 2.8 Lists some other tools that operate in this space, either offensively or defensively. Note that if you have **not** installed them prior to the tutorial, some may take some time to download and install. Also many of the tools here are not very stable or reliable, they will only work on specific systems, architectures, environments, languages, etc. 2.9 Is to explore other languages and try to cause faults/crashes that could be exploitable. This is open to your choice of language and expertise.

Note that the rest of this section suggests some potential programs to create yourself. However, we advise that you start on 2.7 and come back here when it is running.

Most of the examples shown in 2.1 to 2.5 rely on obvious and straight forward mistakes. **Can you combine of obfuscate them so they are hard to reach or only operate under special conditions?** Some examples to consider:

1. Adapt the programs with gcc or clang warnings (2.4 2.5) to not show any warnings for either compiler.
2. Cause the bad behaviour from one of the exercises today via an overflow or other unusual condition.
3. Adapt a program from Tutorial 1 or 2 to only display bad behaviour when a specific input is detected.
4. After 2.7 today is done, detect a crash from Tutorial 1 with afl

2.7 Fuzzing with American Fuzzy Lop (afl)

Let's look at some code that reads a string from input and then acts according to this string. We can compile and test with

```
gcc -o fuzz-me fuzz-me.c
echo hello | ./fuzz-me
echo deadbeef | ./fuzz-me
```

Observe that when we encounter "deadbeef" our code aborts.

Now let's rebuild our code using the American fuzzy lop (afl) fuzzer. This approach is when we have access to the source code:[1]

```
afl-gcc -o afl-fuzz-me fuzz-me.c
```

This builds an instrumented version of the code for use with a afl.

Now we can ask afl to execute the instrumented binary with the input we provide and produce some output. Let's try a simple example:

```
afl-showmap -o /dev/null -- ./afl-fuzz-me < <(echo hello)
```

We should see some basic output from the program and the information on tuples captured (but we directed this to /dev/null so we won't see it).

Aside: tuples are the blocks of code that a went through to produce the output. We won't be looking at them today, but they can be of interest later.

Let's repeat this for some other inputs:

```
afl-showmap -o /dev/null -- ./afl-fuzz-me < <(echo dead)
afl-showmap -o /dev/null -- ./afl-fuzz-me < <(echo deadbeef)
```

Observe that the program output is as we expected and the number of tuples changes (since we go deeper into the nested conditionals).

Now that we have seen the basics of afl interacting with the instrumented program, let's try fuzzing our program. Let's create a test case to start our fuzzing:

```
mkdir fuzz-tests
echo hello >./fuzz-tests/foo
```

and we will create a directory for our results

```
mkdir fuzz-results
```

We may need to set the following:

```
sudo echo core >/proc/sys/kernel/core_pattern
```

WARNING: the next command may be very slow and very CPU intensive, if you don't have good battery life or 5-15 minutes to wait, don't do this now! Now we can start the fuzzer with

```
afl-fuzz -i fuzz-tests -o fuzz-results -- ./afl-fuzz-me
```

[1] https://github.com/google/AFL/blob/master/afl-gcc.c

This command starts the fuzzer and specifies the inputs, outputs, and the program to fuzz. Now we wait (and this may take some time - this is a great moment to take a break, get coffee, talk to a classmate, etc.)...

Eventually you should see a crash in the a output screen. Once this appears, you can kill a with `"Ctrl+C"`. Now if we look in

```
cd /fuzz-results/crashes
```

we should find a file containing a string that crashes our program, in this case "deadbeef"