How software breaks (native)

There is a distinction between security software and secure software. The first term deals with software that enhances computer security, for example antivirus software and firewall fall into this category. The latter is (or should be) a goal for any software no matter the function. There is probably no such thing as a completely secure software (at least not on the larger scale) but steps can be taken to at least decrease the number of security issues and that is what this course is about.

Terminology

Terminology in the field of secure software is a bit debated issue. McGraw in his book Software Security - Building Security In dedicates a full page and a half rant for Microsoft in his opinion misusing the term threat modeling when they should be talking about risk analysis [1]. This course doesn’t take a stance on the term war but uses its own terms throughout the course, after all some kind of unified consensus is needed for a comprehensible discussion.

All software has its own weaknesses and a list of common weakness types can be found in Common Weakness Enumeration web page [2]. If you can find a weakness that can be used against the software, that is a called a vulnerability. A list of common vulnerabilities and exposures can be found in [3]. Exploit is something that leverages a vulnerability.

Attacker is someone who tries to misuse the system regardless of intentions. Attacker might as well be a good guy testing software for security issues as a bad guy trying to actually exploit software. Attack surface is everything the attacker can interact with in his/her mission to find the weak spots.

A threat is a sentient being threatening the well-being of your software (and related) for example a bad guy getting access to your bank account by exploiting a vulnerability in your bank’s software is a threat. Having a feature in program that makes aforementioned possible leads to risk (McGraw defines the term risk = probability*impact[4]).

A bug is an implementation issue as opposed to flaw that is a logic / design issue. A bug might for example be a misuse of some c function allowing for buffer overflow. A flaw might be for example accepting a letter “a” as a password. However, the difference between these two isn’t always clear and may depend on the point of view.

Buffer Overflow

A buffer overflow is a vulnerability commonly caused by careless use of dangerous c or c++ functions. It is low-level and quite effective: in the lecture it was demonstrated with Windows XP how an attacker might manipulate a vulnerable server software and remotely open a calculator.

The program memory is divided (from low memory addresses to high memory addresses) to program itself, data, heap and stack. The instruction pointer points somewhere in
the program to the next machine code instruction to be executed. On the purpose of buffer overflow we are interested in the stack in which reside call parameters, return address, automatic variables and space for the user input.

When the length of the user input isn’t checked in any way and instead the input is blindly stored to the space that has been allocated for it the troubles begin. If the user gives too long an input the allocated space for the value overflows to the other areas in stack. With a little testing and adjusting of the input the attacker can overwrite the return address with the value of his/her choosing.

Nowadays many defense mechanisms have been put to place to prevent the exploitation of buffer overflow. For example randomness has been introduced to memory addresses to prevent attackers from guessing and abusing the right ones.

Special canary bird values are also used to indicate the overwriting of wrong data in stack. When the input value overflows it also destroys the canary bird value and when it is later checked if the value is still in place the overflow is detected. There exists three types of canaries: Terminator canaries which are build of NULL terminators, random canaries which are random to prevent the attacker from guessing them and random XOR canaries which are random canaries with XOR scrambling. If the attacker is able to guess or read the canary value the defense mechanism can be mitigated by placing the canary value to its right place in the value that causes the buffer overflow. This way the buffer overflow overwrites the legit canary value with itself and the canary value check naturally goes through.

The third possible way is to make stack nonexecutable. This thwarts the usual way of exploiting buffer overflow but it is still possible to store the malicious code for example to the heap or other unprotected memory areas. Attacker can also use so-called return-oriented programming where he/she chooses from existing program code or system libraries snippets that suit to his/her purposes and end with return. Then the attacker chains these snippets and executes them one after another to accomplish his/her original intentions.

As the defense mechanisms evolve the attack mechanisms also grow more sophisticated. The basic version of the attack demonstrated in the lecture on Windows XP doesn’t work in the newer operating systems and more sophisticated use of attack knowledge is needed (but it is still possible, only not as easy).

**Fuzzing**

Fuzzing is a testing technique where different kinds of inputs are provided to the target program. The target program is then monitored to find if the inputs cause crashing, memory leaks or other unwanted behaviour. Fuzzers come in different types and functions, they can for example do bit flipping, see if the program breaks around magic numbers or use model-based approach (where the fuzzer knows the syntax).
One example of a fuzzer is Radamsa (which is actually a set of fuzzers) made by people in University of Oulu. Radamsa is used by giving it a sample of valid data, for example a valid picture or pictures if you are trying to test an image viewer. Radamsa then generates a specified amount of random fuzz cases out of the valid data set. After this the fuzz cases are injected to the test program and the programs behaviour is monitored for anomalies. For example in the case of image viewer the pictures are opened with image viewer to see if they cause crashing, freezing or other kinds of misbehaviour. This injecting of the fuzz cases to the target program is often automated.

Sources