Terms

Before actually talking about how to break software, let’s first define the terminology which is being used in this article to make it clear what this article is about. A **Security Risk** is a (potential) way a software can break, if it can break and it has some potential to cause a damage (data loss, data leak, memory leak, …), then it is called **Weakness**. A Weakness can lead to a **Vulnerability**, if there is an actual possibility to be exploited (a weakness might be hidden by a firewall and thus not being able to be accessed by the public for example).

When the Attacker then uses an **Exploit** to exploit the Weakness, he is breaking into the software and showing up a demo. In order to break in, he uses an **Attack Vector**, the Data with that he feeds the application being attacked, and attacks at a for him visible part of the application, the **Attack Surface**.

A **Bug** is a single problem in a software, that causes a Vulnerability, a bad design decision, which might lead to a potential security risk is called **Flaw**.

Binary Exploitation

Binary exploitation means, that given a binary executable which accepts certain input by a user, this input or a part of it is being executed on the machine. If such a vulnerability exists, attackers are able to run their own machine code on using this binary allowing them to manipulate data, change data, etc.

How does it work? Well, a process usually allocates memory for using. They need a stack, a heap, some free memory in between, a data section for storing constant values, and a txt section where the real executable code is being stored. The instruction pointer in the processor always points to the code segment which is currently being executed.

The Stack pointer of the processor points to the current recent value in the process’s stack. If the current function being executed contains a *ret* command, it’s assumed that the stack frame contains the address the process should jump back to. If it now is possible - and it sometimes is - to get a different address to jump back to, the processor will jump to this location, and execute this code.

One possibility is, when a function being called does not check for the boundaries of the data, for example the C function `strcpy()`, this might cause a buffer overflow. This might override some data in the memory following the targeted memory. If then some intended behavior of the program copies this data to a stack frame of the stack, the attacker might gain the possibility to override the instruction’s return
address, and thus, whenever this function is being executed, some other instruction might also be executed.

The next step, after having discovered such a vulnerability, the attacker is able to place his own code, and point to this code - it is then directly being executed. Therefore, he has to get the memory address of his injected code segment which might be randomized by certain security functions of the operating system, then the attacker has to jump to the stack pointer (SP).

However, to make it harder for attackers to attack the application, when using today’s operating systems, every data that the attacker stores, has the non-executable (NX) flag set, causing the processor to terminate execution of the process when such a segment is reached. To avoid this, the attacker, when he is in possession of the binary, has to build his exploit by using existing code of the binary in a non-intended order (because jumping to this code is allowed), by putting those pieces together, there are tools to automatically morph code into this jump chain leading to the problems.

**Fuzzing**

Fuzzing, sometimes also called “fuzz testing”, describes a method to - usually automated, but partly also mostly automated - testing a software for problems in a software. Input is modified more or less randomly into dozens or even hundreds or thousands of test cases, and then being executed with this input data. The main idea is by using so many cases, that if there is a potential vulnerability in the software, that causes a stack / heap / buffer overflow, an infinite loop, or some other crash / hang of the problem, one of those testcases fails, and a software engineer is able to identify the problem why the software has crashed under given circumstances.

There are two different approaches when generating test cases. First, security testers can generate those testcases by hand by modifying the (mostly binary) input data, the other approach is to use automated fuzzing by randomly leaving out bites, bitflipping bytes, or adding some random bytes of a given input sample.

Both approaches might lead to success, however, when generating the test cases by a (machine learning based) algorithmic approach, it is easier to generate nearly an unlimited number of testcases.