Session 1: How software breaks (1/2)

Part 1: Terminology

- Understanding how software typically breaks is a requirement for successful secure design.
- A way software can break is known as a WEAKNESS. A single security issue is known as a VULNERABILITY. The weakness is EXPLOITED via one or more vulnerabilities. (Often the necessary software and activities required is called an EXPLOIT.) When you fix a security issue, you fix a vulnerability or you redesign the system not to exhibit the weakness.
- The act of exploitation is one type of an ATTACK. Note that “Attack” does not imply a malicious intent: If I am testing a system legally, I am still attacking it.

DISCUSS: Given the attacker, who is the defender?

- A security RISK can exist whether or not there is a weakness. A risk is a probability (likelihood) of a system having a weakness, and the impact of someone exploiting a vulnerability.
- A THREAT is an actor that may exploit a vulnerability (or an event that someone does it). The terminology of a threat varies a bit according to source. If you want to be specific and talk about the person, talk about “THREAT ACTORS” or something.
- Weaknesses are catalogued by CWE. Vulnerabilities are catalogued by CVE.
- Vulnerabilities are security BUGS or FLAWS. These can be implementation-level, design-level, or even business-case level, or span several levels.
- The way to deliver the exploit is an ATTACK VECTOR. The more exposure the system has, the larger ATTACK SURFACE it has. Usually, if a system is complex, it has more attack vector possibilities and a larger attack surface. We will get into these concepts more closely in sessions 4 and 5.

DISCUSS: How could adding a security feature make a system less secure?
(Hint: It has to do with increasing the attack surface.)

- The difference between an architecture-level and implementation-level issue is fluid.

DISCUSS: If your password is stolen by a virus, was that a bug in Windows that allowed the virus in, or was it an architectural flaw that made you to use Windows in the first place? If your WordPress blog is hacked, is it WordPress' bug, or was it your fault because you made the
Part 2: A very quick dive into binary exploitation

- One of the weakness categories is where an attacker gets his (*) own code to run in your system as you. (*) I am using the male term throughout for simplicity; attackers can be female or not natural persons.
- This is a particularly interesting category of weaknesses, because it has a possibility of delivering very devastating attacks.
- We will quickly remind ourselves how operating systems work: Program code is loaded into memory for execution. It is executed as a process. Modern operating systems ensure that processes cannot access each others’ memories - except shared memory.
- A process has several sections of memory. The executable binary code resides in the 'text section'. The other interesting areas are the data storage areas of heap and stack. If the process has linked libraries (DLLs, SOs), the code in those libraries is also regarded as a part of the process. (This has meaning for risk analysis; plugins that are DLLs/SOs get executed with your process privileges.)

An example process layout widely shown in literature.
To see a real one on Windows, use VMMap (sysinternals), on Linux, /proc/PID/maps

- The processor has a register called the Instruction Pointer (which we denote by IP; do not confuse with Interner Protocol). This points to the memory address of the binary opcode to be executed by the processor.
- When the binary performs a jump to a function, the operating system places the current IP on the stack, at a location pointed to by a Stack Pointer (SP). When the function ends in a return opcode, the previous IP value (return address, below) is popped from the stack, and execution continues from there.
● It may be possible for an attacker to overwrite the value of the IP stored on the stack, and thus point the processor to code selected by the attacker. We will use this 'old-school' technique to explain what is happening.

● The canonical example is a buffer overflow resulting from careless memory copy operation. One such can be strcpy() in C. The issue is that one has to be really careful when using these functions, because it is very easy to copy too many bytes. If this happens, the overflowing bytes will overwrite something in the memory located after the target memory area.

● An attacker could feed an input to the system where a programming error ends up copying the attacker-controlled input over something else. If this 'something else' happens to be a stack frame allocated from the stack, the attacker may be able to overwrite the IP value stored on the stack frame. Now, when the processor hits a return opcode, the processor continues execution from the IP value that the attacker overwrote the real one with.

● The next step for the attacker is to provide some attacker-controlled code. In the easiest case, the data that overwrote the IP also contains this code, and it gets written to the stack. The attacker will then choose an IP value to overwrite the return value on the stack that will cause a jump to the attacker's code.

● The remaining challenge with this old-school method is that we don’t really know the memory address our exploit ends up in. Hence, we need to do slight trickery, and use a method called a ‘trampoline’ where we jump to the stack pointer (SP). The stack frame that we inject looks like the one below in (2), and the meaning will become clearer in the demo.
Stack layouts. Note that in these pictures, stack grows down and memory addresses grow up; hence, memory copying and execution move upwards in these pictures.

**DEMO: Simple stack-based old-school buffer overflow exploitation on Win XP**

- This fairly simple (by modern standards) attack is becoming harder. There are various architectural aspects that contribute:
  - Stack and heap areas should not contain executable code. On modern processors, it is possible to define these memory areas as "non-executable" (NX, XN ("eXecute Never"), XD ("eXecute Disable"). The feature is enabled by DEP ("Data Execution Prevention") on Windows. If the processor is told to jump to stack, and start executing attacker code there, the process will be terminated.
  - So called STACK CANARIES or STACK COOKIES are values that are written on the stack with (i.e., before) the return address, and their integrity is checked when returning. If an attacker mistakenly overwrites a stack canary, the process is terminated.
  - NX can be countered by the attacker through Return Oriented Programming (ROP). In this model, there is no executable code in a non-executable memory section. Instead, the attacker overwrites the stack with a series of return addresses, each jumping back to the main binary or one of the loaded dynamic libraries (also known as "Return-to-libc").
  - The attacker will examine the target binary to find short snippets of code, called "gadgets", and by chaining them, is able to perform something - for example, to set up parameters for a system() call and then call it. These gadgets can be found, and even exploits built, by automated tools (see https://github.com/JonathanSalwan/ROPgadget).
  - ROP can be countered by randomising the memory address where binaries are loaded. This way, the addresses to jump to, including those of gadgets, change from invocation to another, and it is difficult to write an exploit that reliably chains them, especially across systems. This technique is known as Address Space Layout Randomization (ASLR).
ASLR can be implemented in various ways. Windows executables that are linked with a specific linker option are patched at runtime so that they can run from different memory addresses. On Linux, ASLR-enabled binaries are compiled into PIE (Position Independent Executable) binaries that can be run from an arbitrary memory location. Dynamic libraries (i.e., libraries that are linked dynamically at runtime) are often prelinked (library relocation is precalculated) for performance reasons, and the relocation of libraries can be randomised in prelinking as well.

- ASLR may have a performance penalty (e.g., with PIE on 32-bit x86 systems) or in embedded systems where libraries never change and prelinking is only done once at firmware creation stage, these strategies may not be good enough. ASLR should not be seen as a silver bullet solution (like nothing else really).
- NX/XN, stack canaries, and ASLR are not necessarily universally in use in all targets. For example, on a certain platform, shared libraries that were not built to allow NX would also force their parent process not to have NX. Stack canaries are implemented by code in function prologue/epilogue and if not compiled in, they may be missing. ASLR might suffer from bad entropy or only partial randomisation (leading to predictability) or it might not be used for some binaries, or perhaps the underlying kernel doesn’t use ASLR. The attacker could, therefore, find modules or other executables reachable from your attack surface that just lack these protection methods.

- There is a lot to be learned in binary exploitation. I am not the right person to learn binary exploitation from, and this is not an exploitation course but instead a software security course, so this is the extent to which we go.

Part 3: Discovering input processing problems with a fuzzer

NOTE: Fuzz testing, discussed below, is the topic of this week’s exercise. There is no demo during the lecture. There may be a demonstration during the optional “lab hours” where you can get face-to-face support for the exercise.

- Another way to look at your system is that it has (a) intended functionality and (b) unintended functionality. The intended functionality is used with specified, valid inputs. The unintended functionality is triggered through unexpected, often invalid inputs. Because the unintended functionality is unspecified, it is usually not clear as to what triggers are required. (See the reading notes on weird machines.)
- Ok, as a defender, you’re going to build your binaries with ASLR, NX, and stack canaries. However, those really are the last line of defense; your program has already crashed and it might not be running on your nice and patched box. So, you need to ensure that you don’t even crash. So you need to test for input processing problems.
- There are various input processing problems, to name a few:
  - Overflow: A copy operation overflows the intended buffer and overwrites bytes beyond the buffer. These can be stack or heap based.
  - Format string attack: Many functions use format strings that dictate where variables are to be placed (like %s for a string). Putting format string characters in input may cause the program to write out contents of variables into the string.
- Out-of-bounds read: A read operation reads memory beyond the intended memory area.
- Use-after-free: A memory location is referenced (read or written) after being freed.
- State machine issues: The system accepts wrong inputs in a given state, or a wrong input causes a state change that should not happen.
- Privilege elevation for inputs: The system acts on the inputs that it shouldn’t be acting on, because the input comes from a party who doesn’t have the authority to do that. SQL injection is one example: The system executes SQL statements supplied by the user.
- Denial of Service: The system stops processing (or timely processing) due to an input. This may be a direct result of the input or a subsequent state due to a state machine problem.

These problems can be detected in many ways. The two main categories are:

- **STATIC ANALYSIS:** Program code is inspected "statically", that is, when it is not executing. This can be manual, basically reading code, or helped by a tool. Static analysis ranges from simple pattern matching to syntax checking ("linters"), and the most advanced ones trace execution flows and branches. (Lecture 3 will have more on static analysis tools.)
- **FLAW INJECTION:** Incorrect inputs are fed to the program, and its execution is observed. Contrasting with static analysis, flaw injection is DYNAMIC testing. Observing the target can be done, for example, by running the target in a debugger or a dynamic test harness, or just by monitoring the process and its CPU usage.

Here, we will discuss flaw injection that is intended to discover input processing issues. Getting back to the definition of secure software - "software is secure if it does what it is supposed to do, and fails gracefully if it cannot" - this testing is often called ROBUSTNESS TESTING. A specific subclass of this is FUZZ TESTING or FUZZING.

Again, within fuzz testing, there are a variety of strategies:

- **GENERATION-BASED or MODEL-BASED fuzzing**, where the syntax of the input is modelled through its specification, typically BNF (Backus-Naur Form or Backus Normal Form), ASN.1 (Abstract Syntax Notation) or an XML (Extensible Markup Language) doctype. Then one or more of the fields or parts of the syntax are manipulated. Model-based fuzzing has large and guaranteed input space coverage with a smallish number of test cases, but requires heavy preparatory work.
- **MUTATION-BASED or SAMPLE-BASED or MODEL-INFERRED fuzzing** takes a number of valid data types and breaks them in various ways (bit flipping, truncation, repetition, shuffling, replacement, etc). This can be automated to a great degree and does not require a specification (only samples). The input coverage is not guaranteed.
- **INSTRUMENTATION based fuzzing** breaks the inputs based on the execution of the target when it is being fed with data. Strategies could vary. For example, instrumentation could try to aim for maximum code coverage.
BLIND fuzzing applies the transformations to inputs without any samples or knowledge of the input. It is usually the least effective way, but may be the only solution if we have unique or one-off inputs that cannot be replayed to the target system.

- Once a fuzzed input is fed to the system, the system is observed. The expected, "good" way to process the corrupted input is to drop the input.

DISCUSS: Why is silently dropping a bad input often the best way to handle broken inputs? When will this strategy fail?

DISCUSS: Let’s assume we are fuzzing an image viewer. Some of the test cases cause garbled images to be shown, but the program works otherwise just fine. Is this a bug, or indication of a security problem?

- If the system exhibits a bad response to the fuzzed input, it is usually a crash (process terminates), or a hang (process goes to an undefined state). The crashes can be analysed further to determine whether the crash could be exploitable.
- Observation of the system can be done simply by watching the processes and their CPU usage to catch crashes and hangs. Running the target under a debugger, such as gdb, has the benefit that you can get a stack trace from the crash and use that for classification of the issue (see next part). Test harnesses, such as Valgrind and AddressSanitizer, can be used to detect memory errors.
- In defensive systems engineering, all inputs that cause undefined systemic responses are bugs / flaws.

DISCUSS: If the developers claim that the crash is not a bug because the input is not supposed to exist, or will never happen, is this claim acceptable? Are there situations when an undefined response is not a bug?

DISCUSS: Is the famous Windows Blue Screen of Death a controlled failure?

NOTE: Automated exploitability analysis with a triage tool, below, is extra material and not demonstrated during the lecture. If there are interested persons and time, we can go through this during the “lab hours”.

- Crashes can be analysed for exploitability. This is necessary when there are too many crashes to fix them all. This is used for prioritisation of fixing work, known as TRIAGE.
- Exploitability analysis uses a set of heuristics, including
  - Did the program crash after a return instruction?
  - Did the program try to use an address on the heap that was already freed?
  - Did the program crash on a branch instruction?
  - ...and so on.
- Exploitability tools:
  - !exploitable for WinDbg
- CERT Triage Tools for gdb on Linux
- CrashWrangler on Mac

Exploitability tools necessarily only provide a lightweight analysis of the crash. The fact that an exploitability tool determines that a crash is exploitable does not mean it actually is; and a "non-exploitable" crash could easily have systemic effects which make it useful in an attack.

Exploitability tools also calculate an unique hash of the call stack. This can be used to determine which of the crashes are duplicates (i.e., share the same root cause: the process goes through the same function calls just before the crash).

NOTE: Corpus distillation is an optional topic, and likely not be covered during the lecture. If there are enough interested persons and we have time, we may touch this during the optional “lab hours”.

- A way to increase the effectiveness of fuzzing is to select a very good set of positive, valid samples (the corpus).
- Corpus distillation is a process where a large set of valid test cases is reduced into a set that has the potential to maximally exercise the target system.
- A way to do corpus distillation is to first obtain a large number (say, tens of thousands) of valid cases; run them through a target system with coverage analysis instrumentation; and then select those samples that cause the most differing coverage, with the hypothesis that those exercise the most different decision flows, and fuzzing those has most probability to hit interesting things. Ideas on how to do this include:
  - Checking which ‘basic blocks’ (unique blocks of code between branch instructions) are being executed. Different sets of basic blocks indicate different execution flows.
  - Checking function call graphs. This is on a higher level but can still yield differences between the runs.
- Interestingly enough, you may not need to do the corpus distillation analysis with the target system, but perhaps with another system that processes the same kind of data, and still get good results. For example, if you aim to fuzz an image viewer on a mobile phone, but cannot instrument the viewer on the phone, you could do corpus distillation using an image viewer on your desktop computer.