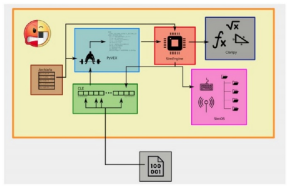

- Static and Dynamic Analysis tools
  - Offensive Security (try to find vulnerabilities and bugs)
    - Fuzzy testing: invalid input can trigger a bug. Automatically generate test cases with many different inputs (memory and cpu intensive). Together with traditional testing improves quality of software.
      - Pros
        - Can help finding bugs and improve security testing (complements usual testing procedures)
        - Can be used to mount attacks
      - Cons
        - May not be applicable to big, complex applications like OSes or videogames
        - May not be able to give enough information to describe the bug
        - Expensive (cpu and memory)
        - Does not detect unwanted behavior that does not cause crashes, errors (viruses, worms, Trojans)
        - Only finds simple faults
        - Difficult to find boundaries in GB, BB
      - Fuzzing Attacks: leave the fuzzer go and wait. Able to find zero-days, crash the application (DoS), eventually RCE.
      - Prevention: security controls: limit bandwidth using firewall, least privilege: chroot jails
      - Workflow
        - 1. Study program input format (generate valid sequences)
        - 2. Fuzz some data according to criteria/decisions/algorithms/models, tools and programs
        - 3. send data to application
        - 4. look for errors (especially memory), crashes, exceptions, invalid answers, hangs, race conditions
        - 5. If error occurred locate it, investigate its cause, explain it, report it
      - What can I do with fuzzing
        - Detect bugs
          - Crasher
          - Memory related errors
          - Hangs
          - Race Conditions
        - Regression testing
        - Make fuzzers interact with other analysis tools (debuggers, memory profilers, sanitizers) to improve analysis results
        - Effectiveness: can't check all application states. Approximations are needed, for example try to obtain full code coverage
      - What to fuzz
        - Generic inputs (strings, integers)
        - Files (text: JSON, HTML, configurations, or binary like images, videos)
        - Network traffic (play client or server role to check configurations (IP, TCP, UDP, HTTP, QUIC))
        - To fuzz more, fuzz a file and convert it into an input / create your fuzzer / find fuzzer
      - Approaches
        - Depending on knowledge of the program (WB, BB, GB: no data structures but some static analysis data, like variables types)
        - Depending on input generated
          - Generation-based: input from scratch
            - PROS
              - Easy to configure
              - Does not need a corpus (set) of seed inputs to exist or to be of high quality
              - Good to stress/find bugs in badly written programs
            - CONS
              - Low coverage
          - Mutational: starts from valid inputs seed and mutate it to generate new.
            - PROS
              - Easy to configure
              - May reach good coverage
              - Most used approach
              - Works very well for image recognition
            - CONS
              - Requires seeds of good quality
          - Model/grammar/protocol-based
            - PROS
              - Excellent coverage
            - CONS
              - High effort needed to configure
              - If software is proprietary, model may not be available
              - Model could be unknown or very complex
        - Depending on complexity of transformation (dumb: generic input transformation, smart: abstraction and other analysis tool outputs to generate new valid inputs)
      - American Fuzzy Loop for fuzzy testing
        - Properties
          - Free
          - Efficient (inserts appropriate instructions into the application during compilation)
          - Effective (used to find bugs in many applications)
          - Considered dumb because it doesn't have the input structure, so the mutations are not perfect, but it uses sophisticated generation techniques (genetic algorithms (mutational) to improve coverage after reaching plateau
          - Grey-box (partial knowledge of abstractions)
          - Connects to other tools to perform more extensive bruteforce fuzzing
          - Generates fuzzed files and provides them as input to the application
        - Workflow
          - 1. Build application with afl-gcc to be fuzzed (extracts information used by afl to perform the grey-box fuzzing)
          - 2. Fuzz a previously instrumented application: afl-fuzz -i dir (containing seeds) -o logs (directory to write logs into) cmd (command to run application)
            - Generates input files and reports (logs)
              - logs/fuzzer_stats: full mutation cycles, executions, paths, executed paths, crashes with unique paths, timeouts with unique paths
              - logs/crashes: fuzzed input file which resulted in crashes
              - logs/hangs: fuzzed input files which resulted in timeouts
              - logs/queue: fuzzed input files which lead to distinctive paths
          - 2b. Dictionaries
            - ALF optimized for compact binary files
            - Slow learner for textual/verbose text files
            - afl-fuzz -x dictionary (key="value" pairs of string, key ignored, value used as keywords useful for the fuzzer)
          - 2c. Parallel fuzzing: It is not enough to launch separate AFL instances because many test cases will be identical. Parallel mode is used. Each copy could use a different generation mode (-M id for deterministic mutations, -S id for random mutations). Check results with afl-whatsup logs.
          - 2d. Integration with Driller (by Shellphish): AFL version that help automating parallelization, uses concolic analysis to increase coverage and make smarter mutations
          - 3. afl-plot logs dir (directory to write plots into): monitor AFL work to find how is the coverage increasing (see if we reached a plateau)
          - 4. Advanced crash analysis: afl-fuzz -C. Generates a file related to a single crash events, with many mutations that lead to that event, so that it will be easier to analyse the cause of the crash and to fix the bug
          - 5. Corpus minimization: remove unneeded files that do not increase coverage so they only slow down fuzzing. Minimize: afl-cmin -i dir (directory with original corpus: set of initial files seed) -o min (directory with minimized corpus) cmd
          - 5. Repeat from 2.
          - 6. When to stop: 1 mutation cycle, no more paths/bugs found, code coverage high enough
    - Symbolic and concolic analysis
      - ANGR modular python framework for symbolic,concolic analysis, symbolically-assisted fuzzing, automatic exploit generation, binary loading, rewriting, type inference
        - Architecture and workflow
          - CLE Loads Everything: Extracts executable code and data from any format, guesses architeture, creates representation of the program's memory map as if the real loader had been used! Generic Loader!!!!
          - Archinfo: collection of classes with architecture-specific information useful for cross-architecture tools!!!
          - PyVEX vector extension: higher level abstraction of the opcodes: intermediate representation of the instructions!!!
          - Simulation Engine: Interprets the code (in intermediate representation) and simulates its execution moving the program's states (registers+memory+other archinfo attributes). Generates the set of successors' states. For each branch collects the constraints (conditions) that allow entering that branch
          - Claripy: models the result of the simulation engine managing concrete values, symbolic expressions, building symbolic trees of expressions over variables, adding constrains. Uses SMT solver Z3 to solve every branch expressions and obtain inputs needed to reach that state, compose and simplify expressions. Allows to stop when custom conditions (also advanced) are reached
          - Z3 open source SMT solver with excellent performance. Converts constrains on other data type into Satisfiability problems (this system of boolean formulas has a solution?). Find solutions that maximise custom objective function
          - SymOS: Models the Operating system: files, pipes, network object, syscalls, libraries. SimLinux uses symbolic version of library functions and syscalls instead of simulating the their behavior every time.
        - ROP generators
        - Driller fuzzer
        - Code
          - import angr; p=angr.Project("executable_name", load_options={"auto_load_libs": False})
          - Archinfo data: p.arch
          - CLE: p.loader
          - States objects: blocks to be executed, value of all registers, memory, I/O
          - Simulation managers: objects that perform concolic execution: transform state into the next one
            - sm = p.factory.simulation_manager(p.factory.entry_state())
            - sm.explore(find=0x0804870, avoid=0x0804871e)
          - Basic blocks: pieces of binaries with no jumps
  - Reverse Engineering: Extract human understandable knowledge from anything man-made
    - Levels of abstraction and reconstruction
      - Application-level: Understand high-level system description, source architecture and business model
      - Function-level: logical and functional specification, non-functional requirements
      - Structure: data and control flow, dependency graphs, structure and subsystem charts, architectures
      - Implementation-level: symbol tables, source text
    - Goals:
      - Understand high-level system description, source architecture and business model
      - Reconstruct the source code
      - Correct/adapt binaries according to your needs: build attacks/cracks/patches
      - Understanding the behavior of proprietary information PI (check software protection used/steal PI)
    - Executable binaries: complex data structures (data+code+management informations: memory allocation info, dynamic linking info, symbols..). Portable Executable (Windows), Executable and Linkable Format ELF Linux
      - Format
        - Program header table: segments+how to create memory image of the program
        - Symbol table: all the debugging symbols, dynamic linking and relocation info
        - Dynamic linking: info for runtime loading of shared library
        - Different types of sections: RO, W, data, code
        - Procedure Linkage Table PLT section
        - Global Offset Table GOT section
      - Tools
        - Objdump
        - Readelf
        - Elfutils
      - State of the art of the binary code composition
        - Variable-sized instruction set (Intel x86, CISC)
        - Instruction sets densely used
        - Instructions may overlap: same bytes executed multiple times (JMPs), interpreted as belonging to different intructions. Recursive path traversal is needed, linear path traversal is not enough!)
        - Data may be embedded in the code (jump table/obfuscator placed data there!): separating data and code is an undecidable problem!
        - Indirect jumps/calls -> destination address is only know at runtime: execution flow must be reconstructed
        - Important data not available at the disassembler (names, data types, macros, comments) (hard for humans to understand)
        - Difficult to correctly reconstruct functions and prototypes (distinguish functions and parameters from other information on the stack)
        - Pointer aliasing: 2 pointers that refer to the same memory area-> only known at runtime, difficult to reconstruct execution flow
        - Self-modifying code: code changing at runtime (malware, super-optimized code)
      - Dynamic Linking
        - Lazy binding: binding of shared function only happens when you call a new shared function, not at loading time
        - 1. Call shared_function() for the first time
        - 2. Look into the Position Linkage Table PLT for the shared_function() symbol and jump to the GOT entry corresponding to shared_function()
        - 3. GOT entry contains the address (in the PLT) that calls the dynamic linker
        - 4. Dynamic linker resolves the shared_function() absolute address and places it in the GOT entry so that for the next calls step 3 can be avoided
        - Attack: write in the GOT the address of functions that you want to call instead of the original ones, maybe exploiting format string attack
        - Format Strings Attack
          - Must be able to control the first argument of the printf
          - Add more %. indicators than the number of parameters to be able to access portions of the stack (or content of registers)
          - AAAA%x%x.. to read stack until you find the marker that you inserted
          - %offset_of_the_markersx to skip the other parameters: will only print the marker
          - Substitute marker with an address and %x with %s so that the printf will follow the address that we passed and read the string pointed by it -> let us read every readable address!
          - With "..%n", &var_addr we can assign to the variable the number of characters written until the %n
          - %numx prints num hex characters
          - To write into a variable the value that we want, we make the printf load with that number of characters, then we point to the address passed and write that number there -> let us write any writable address (some limitations): useful if we manage to write the value of the address of a function that we want, where we want
        - Mitigations
          - Position Independent Execution PIE: program and libraries can be placed in any memory location -> disassembler can't find real addresses which will be used in execution
          - RELocation Read Only RELRO protection
            - Partial RELRO: GOT section read-only (GOT.PLT not readonly) and sections rearranged to minimize the risk of overwriting the GOT
            - Full RELRO: all dynamic code resolved at link time before the start of execution (slow start time) GOT.PLT integrated into GOT and made all read-only
    - Static Analysis Tools
      - Disassembler: generates a listing written in assembly such that a given assembler will encode it to an executable syntactically equivalent to the original one. It is not deterministic because compilers use very advanced and complex techniques (state of the art binary composition)
        - Linear Sweep: sequentially decodes bytes into instructions from the beginning of the executable. Easily desynched (objdump -d)
        - Recursive Path Traversal: Built by parsing code listing and following targets of branches and procedure calls ->expected/reconstructed execution flow. With Heuristic-driven recursive traversal disassembling, the disassembler tries to build a cfg also for pieces of code that appears to be disconnected from the cfg (radare2, ida pro)
        - Interactive disassembly: disassembler asks to manually resolve misinterpretations of data as code
        - Control Flow Reconstruction: problem of determining an overapproximation of all possible sequences of addresses that will be executed by a given program -> overapproximation Control Flow Graph (Call Graph: only calls and returns).
        - Tools
          - Objdump -d
          - Radare2: static (disassembler) and dynamic (debugger integration with gdb) analysis, patching, advanced search, scripting support, extensible framework (plugins). Includes Rabin2 tool to get binary (ELF) information
      - Decompiler: tries to create high-level source file starting from an executable file. Very hard problem, but result is easier to understand than disassembled code
        - Ghidra: disassembler, decompiler, grapher, scriptable, extensible
    - Dynamic Analysis Tools
      - Debugger: allow controlled execution of an application. Mediate interaction with execution environment (file system, network, syscall), interrupt execution, examing cpu status, examine memory locations, examine stack, write values in memory, conditional execution to detect conditions
        - Software breakpoints overwrite the instruction where debugger needs to stop with a SIGTRAP -> change process address space: could be invasive for certain applications.
        - Hardware breakpoints use dedicated registers (DR0-DR7 for intel, or JTAGs (Joint Test Action Group) 32 more hw breakpoints) to keep the breakpoints -> don't overwrite instructions
        - Gdb+pwndbg
      - Tracer: understand system behavior, gather statistical data, monitor application
        - Ptrace() system call that allows one tracer process to observe and control the execution of another process (also threads). Writes in target memory or code segments (breakpoints, patching). Used by gdb
          - strace reports about systems calls and their parameters, signals, child processes. Good reporting but very intrusive.
          - ltrace traces calls to library functions using PLT trampoline
          - Malicious users can use ptrace to dynamically inject arbitrary code in the running process
          - To disable or constrain ptrace in production environment
            - Prctl() syscall to make the process not dumpable
            - Security modules to restrict users able to access it (YAMA)
            - Use dockers to confine traceable processes: ptrace yourself
            - Monitor: detect ptraced processes, monitor ptrace events with ebf kernel data structure or netlink when switching between user and kernel space
        - Trace-cmd: interface to configure Ftrace kernel built-in tracer, designed to monitor what happens inside the kernel
        - SystemTap framework for automatic tracing with scripting support
        - VM introspection: execute application in emulated environment, then collect information from the outside: indispensable when analyzing the malware. Cuckoo sandbox
    - Application monitoring
      - Profiler
        - gcov: line, branch, basic block coverage compiling using -coverage. lcov exports gcov coverage report as html. Less powerful than a fuzzer, useful for profiling information useful for developers