Mem Errs Def Mechanisms

Non-executable Memory. Code injection attacks involves writing and executing data. Strategy is to try and prevent injected code by making some portions of memory non-executable. Code injected as data is not executable **does not prevent vulnerability but prevents exploitation.** Non-executable stack and data requires OS and hardware support, usually with a flag in page tables.**Doesn't prevent attacks reusing existing code**.
Stack protection/detection. Memory error violates **integrity** of stack. This is **not a prevention**.
Reordering variables on stack. **reduces not prevent** data corruption from buffer overflow.
Shadow stack. Don't use normal stack for return addresses/check integerity. Maintain shadow stack in a hard to buffer overflow part of memory. **Prevents** control flow hijack from return address. Cna lead to some synchronising issues with non trivial costs.
Costs. Instrument calls forward phases of control flow allows modifying function call instruction sequence. **Push return addr on shadow stack** and normal stack. Instrument return modifies the return instruction sequence. **compare normal stack return address with shadow stack**, throw exception if different.
Stackguards. Add canary (or cookie) value between value to be protected and stack variables. Assumes buffer overflow overwrites canary/be stopped by canary value (**incomeplete protection**). This **does not remove vulnerability**. Canary values are set to prevent certain attacks. NULL can prevent a strcpy attack. Random values make it difficult for attacker to predict what value to overwrite.
Compile flags. Checking code needed in function epilogue to verify canary value integrity before return instruction. Usually needs recompilation of source code. Attacker guesses canary or try to discover it through information leak.
Addr Space Layout Randomisation. Randomise address of some stack, heap, globals code. In practise, only some things are randomised such as starting position. Usually constrained to randomise at page level. Code is required to be compiled as position independent in Linux.
Protection. **Dont prevent vulnerability**. Just makes it difficult for attack to succeeed, assuming attacker dont know how to exploit code/data address. Assumption is that attack can fail as address may be wrong but ultimately depends on the level of randomness. Information leak can still leak information.
Buffer Overflow Checking. Prevent by checking all accesses are valid. Need to instrument pointer code since only instrument code is protected. Some approaches includes checking of pointer arithmetic and address of pointer access by checking variable with pointer(need to store metadata), object based (recover bounds from object), boundary checks (check redzones around object).
Object bounds. Detect error outside object.
Sub-bounds. Detect error outside subject, difficult.
Allocation bounds. Detect errors outside allocated memory for object.
ASAN. Uses boundary checking with redzone. Requires maintainance of shadow map, redzone checks are incomplete since overflows are only detected in redzone. Increased mem usage with redzone and shadow map. Can call uninstrumented code but mem usage shouldnt conflict with shadow map.
Shadow Memory. Compact map of main memory to indicate whether address is accessible. Every aligned 8 bytes has 9 states. First n are addressable, 8-n are not, encode in 1 shadow byte.
Object redzones. Add no-access redzones around object, taking up actual virtual address space, affect cache, redzones may need to deal alignment. Usage of redzones is a buffer overflow. Add redzones for everything which is to be protected like global/stack vars and heap stuff.
Detecting Temporal Errors. Memory allocation clears poison (remove red zones) if poisoned of object memory. Freeing memory poisons object memory and delay re-using of memory for new alloaction. Poison red zone detects UAF errors.
Fat Pointers. replace machine pointers with these instead. Store object meta information in pointer. This changes the object layout, increases mem usage and interfere with binary compatibility.
LowFat Pointers. Encode meta-info within machine pointer. Machine pointer size unchanged (achieving binary compatibility). Requires large address space. Heap provides freedom in allocation, check heap alloc bounds (whether access within malloc size + padding). Protects **heap, stack**.
LowFat Allocator. Creates lowfat heap pointers with a special memory layout. Creates different regions, where each region is for objects within a specific size range base on a config table. Regions 0, S maps to be unrestricted. Non-lowfat pointers **always succeed in bounds check**. Passing non-lowfat pointers will not cause error.
Finding meta-info. Based on pointer, we can get the index and size. Then we can compute the base address from these. From the size of the table, we can divide the pointer by the size and multiply it with the size.
Tradeoff summary. **Non-executable data** has almost 0 overhead, low effectiveness and high compatibility except JIT compilation. **Stackguard** Very low overhead with low effectiveness and high compatibility for source code. **ASLR** Low overhead, low-medium effectiveness depending on entropy with high compatibility. **CFI** Medium overhead with medium-high effectivness for control flow and medium compatibility. **Buffer Overflow Checking** has medium-high overhead with high effectiveness and compatibility.
Binary Hardening. Detect memory error given an attakcer by making it harder for attacker to bypass.
Unspecified behavior. Provide multiple possibilities and impose no further requirements on which is chosen in any instance.
Implementation Defined Behavior. Unspecified behavior where each implementation documents how the choice is made.
Undefined behavior. Anything is allowed. Accessing things that are out of bounds. Compiler may choose the meaning.
Spatial Error. Access outside the bounds of an object.
Buffer Overflow. Spatial memory error. Behavior is undefined - anything is allowed. This can modify behavior of program.
Boundary Condition. Can pass the boundary just cannot access it.
C strings. char in C is an integer type. Strings are array of char and must be null character terminated. Size includes the null at the end.
Stack Overflow. Local buffer overflow; corruption on data and control. Violates integrity.
Control Flow Hijack. Attacker control return address with chosen input. Can corrupt return address for caller, multiple frames, data and %ebp.
Return-into-libc. Uses existing code in program. Return address overwrite is existing code to call.
Return Oriented Programming. Generalised version of the above. Return to **any** code fragment ending with a return type instruction. Chain multiple fragments to execute code controlled from stack or stack control PC.
Heap Buffer Overflow. Buffer overflow on the heap. Usually with the program code. For a structure, we can overflow a field to update another field. Corrupting multiple objects depends on how malloc works where objects are located.
gets(). inherently unsafe. Ready bytes from stdin until new line or EOR. Amount of data written to buffer is controlled by input, no way to restrict to the size of the buffer.
fgets(). Safer alternative from gets(). Reads bytes until n-1 bytes or new line then transfer to buffer. Difference is that there is no overflow but adds newline to buffer.
strcpy(). Potentially unsafe. Copies s2 into s1 including null terminator. Possible overflow if the size of s2 $\geq$ s1.
strnlen(). strlen() can go past non-null terminated data. This is the safer version.
Safer string copy. strncpy() not terminated if s2 $\geq$ n, can lead to null terminations errors. strlcpy() always null terminates.
Format string vulnerabilities. Variable arguments and input can be controlled by attacker. Possible to buffer overflow (for sprintf) and creates DOS if there is mismatch number of args.
Format string conversion. %n conversion is dangerous as it writes some chars written so far to value pointed by argument.
Temporal Error. Access memory of object which has been deallocated.
Malloc. Returns pointer to aligned mem region of **at least** n bytes. Memory is not initialised. If size is 0, may return a unique pointer which can be freed.
free(). if pointer is null do nth. Undefined for some random value. Logically deallocates memory (!= physical unallocated).
Use-after-free. Any value in block of memory already free is accessed. Leads to undefined behavior. Reading after free usually succeeds but is a memory bug. Write after free can corrupt memory.
Dangling Pointers. A form of UAF. There may be pointers still pointing to an object that was freed. Vulnerability if those pointers are accessed.
Double Free. Leads to undefined behavior.
realloc(). deallocated old object pointed by pointer and allocate a new size. If mem cant be allocated, doesnt perform dealloaction and nothing happens. Successful calls frees the previous memory and performs allocation again.

CFG and CFI

Control Flow Graph. Directed graph where nodes are basic blocks (sequence of straight line code with a single entry, possibly branching at end). Edges are control flow transfer between blocks. Nodes can have multiple incoming and outgoing edges. Model ofhow control flow works/meant to work in a program at source code/binary level.
Interprocedural CFG. Nodes in call graph are functions. Directed edge show function calls. f-to-g edge for function g called inside function f.
Control Flow Integrity. Limit control flow transfers to what program allows. Only allow control flow within static CFG of program. A CFI violation is a flow that doesn't follow CFG.
Challenges. How to determine CFG accurately if there is dynamic behavior (pointers). Computing accurate CFG is undecidable; over approximation to allow FN.
Foward edge. Jump or call instruction. Jump/call target is indirect address. Absolute target cant be changed by attacker. Branch target not possible as it will be a CFI error. **possible != correct**.
Backward edge. Return instruction. Is return target continuation of original call. Can be checked with shadow stack defences.
Instrumentation scheme. indirect call instruction, call L(special tag), R. Success if address in register R has matching lablel, else CFI error. Return instructions are ret L. Success if return address has matching label L else CFI error.
Assumptions. Read-only code. Data execution prevention is on.
Limitations. Only deals with control flow. Needs to be an approximation of true execution. Branching to address is impossible at runtime without attack. Under-approximation can cause errors. Over-approximation may allow some attacks.
Evaluation. Low overhead with more FN. Forward egde only CFi more common but overhead of shadow stack may not be small.

Blackbox Fuzzing

Blackbox Fuzzing. Generate test case based on domain. Can use specs but not source code. Can mutate/generate input based on specs. Fast but limited coverage.
Limitations. No guarentee bug is found, but some may be. Cannot prove there are no bugs as it does not show correctness, no misbehavior and usually no guarentees.
Random Fuzzing. Generate random input and run program with input. If crash, log that input.
Assumptions. test case encodes the bug, buggy input causes crash. Avoid/minimal analysis of program, **no source needed**.
Test Oracle. Check the result, assumes error means test failure.
Weak Test Oracle. Doesn't check for correctness of behavior/output. Weaker property may or may not indicate correctness in the form of assertion failure, memory error, timeout and crash.
Functional Testing. Deriving test cases from program specifications. Functional specification is the description of intended behavior. Timely (refining specs before writing code), effective (find faults that can elude other methods), widely applicable (can work for any spec and any type of app), economical(less expensive to design and execute).

Doesn't require program code but need description of intended behavior. Side benefits that reveals ambiguities of spec.
Random testing. Pick possible inputs uniformly. Avoids designer bias but may not be efficient. Closer to fuzzing.
Systematic testing. Try to select inputs that are more valuable, usually by choosing representatives of classes that are apt to fail often or not supposed to fail. Does not mean complete. This is functional testing.
Random + Systematic Testing. Generates some boundary values first to capture corner cases/important relationships. Cover the rest of the input domain by random testing.
Cluster Crashing Tests. Ideally its a semantic analysis of test to find a reason by storing a bucket of tests against each logical formula. In reality, its based on point of failure location/function/CFG path.
Black-box Fuzzing (Random). Random permute input and try to crash. Highly effective but may have issue with codecov and checksum.
Mutational Fuzzing. Takes in seed and mutation ratio. Obtain an input by performing mutation op. Run the input through program to see if it crashes. Document outcome **regardless** and generate next input. When some limit is reached (iteration/time), verify mutation dont run same input twice. Independent of program and input structure.
Principle. Take well-formed input that dont crash and minimally modify/mutate it to generate slightly abnormal input and see if it crashes.
Generational Fuzzing. Generate well-formed inputs systematically using a grammar or a specification. Provides understanding of behavior and syntax and guides fuzzing based on this understading.
Grammar approach. Selectively mutate input base on grammar producing almost correct input. Mutate subset of inputs. Higher codecov by selectively fuzzing different portions of input.
Blocks approach. Each block have rules written in API form and simpler than grammar-based fuzzing. Identifies a rough structure of input. Aimed at re-using previously known correct structures to generate new inputs.
SPIKE Fuzzer. FIFO Queue through SPIKE API where string variables are fuzzed by SPIKE. Still requires user to manually create the malformed input but provies an API to do so.
Structural testing. Create functional test suite then measure structural coverage to see whats missing. This is automated but does not ensure **effective** test suites.
Statement testing. Each statement (node in CFG) must be executed at least once. Coverage is calculated by dividing number executed with number of statements. Fault in a statement only revealed by executing it. Assumption is that statements are independent.
Coverage. 100% node coverage = 100% statement coverage though improvement in one may not improve the other by the same amount. Does not depend on the number of test case. **Node coverage != Edge coverage**
Branch testing. Each branch must be executed at least once. Coverage calculated same way as statements. Traversing all edges of graph cause all nodes to be visited but the other way may not be true.
Path adequacy. Path testing consider combinations of decisions along paths. Each path must be executed at least once. Number of paths when loop is involved is possibly unbounded. Partition infinite set of paths into finite number of classes.
Coverage-based greybox fuzzing. Use codecov to give execution feedback. Focuses on inputs which can give more coverage. Not blackbox since looks partially into program.
American Fuzzy Lop. Coverage-based greybox fuzzing. Counts CFG edges executed. Coverage array is 64kb shared mem region by caller (overhead).
IsInteresting strategy. If new tuple is created in coverage(from the same trace) or transitions between bucket coverage counts.
Fuzzing strategy. Deterministic when its sequential modifications and non-deterministic for stack operations and splicing test cases.
Architecture. Load tests, forks new process running code. Driver wait for target process to finish. Record branch coverage bitmap logs coverage bitmap on exit. Use shared mem as coverage bitmap. Collect bitmap to decide if input is interesting. Terminate when no error found or error found causes some crash or different output.

Patching

Patching. Manually or automatic through dynamic instrumentation vs source code. Add error checks, learn invariants and enforce, code repair from tests and symbolic execution.
Patch delivery. Signing/verifying patches. Is downtime needed vs hotpatch.
Challenges. Undecidability with FP and FN. Intractability (cant scale). Pratical issues like managing FP, tradeoff accuracy v precision.

White-box fuzzing

Whitebox fuzzing. Complete knowledge of program.
Greybox fuzzing. Offline analysis with partial knowledge of program. Use knowledge to guide blackbox fuzzing. **Symbolic path exploration.** Explore all program paths starting with (un)known inputs.
Grey-box fuzzing. Combine blackbox fuzzing with cheap code analysis. **Symbolic execution.** Execute program with un-known input values. Evaluate RHS symbolically, assign LHS as part of state. Execute condition by trying both branches. Update PC (path condition) with the condition. PC become more specific. Check if PC constraint is satisfiable. On termination, solve constraint. **Inputs.** Inputs are not specific concrete values but instead could be a set of values instead. Do not know the test case. Creates observable differences through mathematical output from tests. **Path conditions are constraints**. Concrete executions only execute some path and does not guarentee a specific path.
Solving constraints. Given some constraints, say YES if there is a solution, NO otherwise. Integer constriants: $\sum a_i x_i \leq c$ is NP complete for integers and poly time for real numbers for x_i. Conjunction (and) of string match constraints are NP-complete. Transcendental functions $sin^2(x) + cos^2(x) = 1$ is undecidable in general. Multivariate polynomials $x^3 + y^3 = z^3$ over integers are undecidable.
On-the-fly path exe. Instead of analysing the whole program, shift from one path to another. Continue searching for failing inputs and try those which dont go through the same path. Directed Automated Random Testing (DART).
Assignment store. Set of variable-value tuples that contains mappings **after** executing a particular line.
Path condition. Boolean expression in order for a particular line to be executed.
Random Test Driver. Generate test driver automatically to simulate random env of interface. Compile program with test driver to create a closed executable. Run executable to see if assertions are violated.
Concolic. Concrete + Symbolic execution. Execute condition and resolve branch condition using concrete values. Then update PC depending on condition.
DART approach. Maintain concrete and symbolic states after executing each line. Start with a random input. Then trace through the program while maintaing the states and constraints. On termination, negate one constraint to find the next possible input and repeat.
Challenges. Incomplete constraint solver. Negation could be problematic and solving of path constraints may be undecidable.
Execution Tree. For each satisfiable leaf, there exists a concrete input for which real program will reach the same leaf so we can generate tests. Path conditions associated with any 2 satisifable leaves are distinct leads to codecov.

Arithmetic Errors

Fixed sized ints. Various sizes allows for signed and unsigned. Signed represented as 2s complement.
Arbitrary precision ints. infinite precision, allowing no loss of precision but more storage and time.
Floating point. Approximation of real numbers. Not associative and distributive in general. Double more precision and exponent than single in bits, but **does not mean more bits guarantee more correct answers**.
Numeric exceptions. Division by 0, under and overflows. Most exceptions by default are not raised.
Representation. n-bit unsigned ints represent $0, ... 2^n - 1$. n-bit signed ints represents $-2^{n-1}, ..., 0, ..., 2^{n-1} - 1$.
Unsigned Wraparound. Computation is modulo 2^n. No overflow. May wraparound for extreme values.
Signed int overflow. On overflow, up to implementor. No signal may be raised. Number of negative values 1 more than positives excluding 0, meaning positive and negative cases may need different handling.
Loop counters. Avoid wraparound of loop index due to overflow and wraparound.
Up Conversion. Converting from smaller width to bigger width. Preserves value including sign.
Down conversion. Converting from bigger width to smaller. Loss of precision as less bits for representation.
Conversion from unsigned. Up promotion usually safe. Down promotion truncates and preserve low order bits. Result mod new type.
Conversion from signed. Up promotion is safe. Down promotion is implementation defined.
Mitigation. Input validation, consistency check, right size vars, usually unsigned index, arbitrary precision arithmetic, check ranges, static analysis, fuzzing, compiler warning and runtime checks.

Operating Systems

Environment Variables. Reset env vars if attacker can control the env.
Compiler as Trojan. Build trojan compiler to the same name as the original compiler. Use trojan compiler to build main file binary. Since trojan compiler has backdoor in code when compiling main file, there will be backdoor once main file is compiled. Assumptions of main file is that there is no vulnerabilities in the file. Compiler and binary cannot be trusted.
Character Encoding. Unicode has variable byte length encoding for unicode. Complex schemes lead to bugs especially for UTF-8.
Right to Left Override. Using `[U+202E]` reverses the characters after that, creating a false name that looks legit.
Homograph Attack. Spoof similar looking characters by replacing them with lookalikes.
Pathname Sanitisation. Consider exact semantics of pathname. Replacing all ../ will not work since attacker can just add an additional instance of it to target path. Using strcmp may not work because there are many equivalent pathnames.
Symbolic links. File where its contents is recursively a pathname. Opening a symlink is the equivalent of opening a file at the base of the recursion.
Pathname Canonicalisation. Takes a path and returns absolute path without any dots, symlinks and extra slashes.
Hard links. Same file refers to the same inode. Hard links are restricted to 1 filesystem.
TOCTOU. Time of check, time of use. Check that invariant is true, error if cant meet condition. Use assumes invariant from Check and use the object.
TOCTOU vulnerability. Sequence may not be atomic. Attacker can context switch to modify things in between function calls.
Temp Files. Use hard to guess random filename
Process Identity. ruid is the real owner of the process. euid is the effective uid, used to allow/deny access. ssuid is saved uid. root is privileged user, perform most ops without being denied. euid + egid for access control.
File Permissions. read, write and execute bits for files. same set for dirs and a sticky bit, whether can rename/delete. other bits applies if user/grp of process dont match file user/grp.
Changes privileges. if setuid/gid bit is on, changes process identity when executable is executed. this changes identity of process and effective permissions. Possible to switch to a nobody which has no privileges. Changing and reducing privilege may be permanent. Only done at process level.
System calls. setuid drops root privileges to user if euid of process is root. seteuid only changes euid even for root. temporarily drops privileges with seteuid. setresuid sets ruid, euid and ssuid atomically, can be used to drop privilege permanently if all parameters are the same.
Unprivileged slave process. Privilege monitor creates unprivileged child processes. fork child process. change their identity to nobody. restrict file access with filesystem isolation/sandbox.