Software security fiche

Pierre Colson

Contents

Markdown version on *github*
Compiled using *pandoc* and *`gpdf` script*

General

- **Security** is the application and enforcement of *policies* through *mechanisms* over data and resources
  - *Policies* specify what we want to enforce
  - *Mechanisms* specify how we want to enfore the policies (i.e. an implementation/instance of a *policies*)
- **Software Security** is the area of Computer Science that focuses on testing, evaluating, improving, enforcing, and proving the security of software.
- A **Software bug** is an error, flaw, failure, or fault in a computer program or sytem that causes it to produce an incorrect or unexpected result, or to behave unintented ways. Bugs arise from mistakes made by people in either a program's source code or its design, in framworks and operating systems, and by compilers
- A vulnerability is a software weakness that allows an attacker to exploit a software bug
- Ethics
  - Black hat: Attack other systems for profit (illegal)

  - Grey hat: look out for your own benfit
  - White hat: Honest security professional

Basics

Basic security principles

- CIA Security triad
  - **Confidentiality** : an attacker cannot recover protected data
  - **Integrity**: an attacker cannot modify protected data
  - **Availability**: an attacker cannot stop/hider computation
  - Accountability/non-repudiation may be used as fourth fundamental concept: it prevents denial of message transmission or receipt
- The **Threat model** defines the abilities and resources of the attcker. Threat models enable structured reasoning about the attack surface
- Security is
  - expensive to develop
  - expensive to maintain
  - may have performance overhead
  - may be inconvinient to users
- Fundamental security mechanisms
  - **Isolation**: Isolate two components from each other. One component cannot acces data/code of the other component except through a well-defined API.
  - **Least privilege**: The principle of least privilege ensures that a component has the least privilege needed to function.
    - Any further removed privilege reduces functionality
    - Any added privilege will not increase functionality
    - This property constraints an attacker in the obtainable privilege
  - **Fault compartments**: Seperate individual components into smallest functional entity possible. These unit contain faults to individual components. Allow abstraction and permission checks at boundaries
  - **Trust and correctness**: Specific components are assumed to be trusted and correct according to a specification
- **Abstraction** is the act of representing essential features without including the background details or explenations. Abstraction allow an encapsulation of ideas without having to go into implementation details
- **OS abastraction**
  - *Single domain OS*
    - A single layer, no isolation or compartimentalization
    - All code runs in the same domain: the applcation can directly call into operating system drivers
    - High performance, ofter used in embedded systems
  - *Monolithic OS*
    - Two layers: the operating sytem and applications
    - The OS manages resources and orchestrates access
    - Applications are unprivileged, must request access from the OS
    - Linux fully and Windows mostly follows this approach for performance (isolating individual components is expensive)
  - *Micro Kernel*
    - Many layers: each component is a seperate process
    - Only essential parts are privileged
    - Applications request access from different OS process
  - *Library OS*
    - Few thin layers, flat struture

        * Micro-kernel exposes bare OS services
        * Each application brings all necessary OS components
- **Hardware abstraction**
  - Virtual memory through MMU/OS
  - Only OS has access to raw physical memory
  - DMA for trusted devices
  - ISA enforces provilege abstraction
  - Hardware abstractions are fundamental for performance
- **Access Control**
  - **Authentication** : Who are you ?
  - **Authorization** : Who has access to object
  - **Audit/Provenance** : I'll check what you did
- There are three fundamental type of **identification**
  - What you know : username, password
  - What you are : biometrics
  - What you have : smartcard, phone
- **Information Flow control** : Who can see what information ?
  - Access policies are called access control models
- Type of **access control**
  - **Mandatory Access Control** (MAC)
    * Rule and lattice-based policiy
    * Centrally controlled
    * One entity controls what permissions are given
    * Users cannot change policy themselves
    * Examples : The admin sets permissions for each file
    * Bell and LaPadula : *read down and write up* => enforces *condifentiality*
    * Biba : *read up and write down* => enforces *integrity*
  - **Discretionary Access Control** (DAC)
    * Object owner specificies policy
    * MAC requires central control, DAC empowers the user
    * User has authority over her resources
    * User sets permissions for ther data if other users wanc access
    * Examples : Unix Permissions
  - **Role-Based Access Control** (RBAC)
    * Policies defined in terms or roles (sets of permissions), individuals are assigned roles, roles are authorized for tasks.

Secure software lifecycle

- **Secure software engineering**
  - Prevent loss/corruption of data
  - Prevent unauthorized access to data
  - Prevent unauthorized computation
  - Prevent escalation of privilege
  - Prevent downtime of resources
- **Secure development Cycle** (SDC)
  - **Requirement Analysis** : Define scope of a project and security/privacy boundatries. Define security specification, identify assets, asses environment, and specify use/abuse cases
    * **Threat Modeling** : threats, attack vector, and emergency plans (i.e. how to react when things go wrong)
    * **Security requirements** : privacy policy, data management plan
    * **Third party dependencies** : define third party dependencies along with their update policies, risk analysis on dependencies
  - **Design** : the classic design phase focuses on functionality requirements. here we make security

concerns an integral part of the analysis
    * Continuously **update threat model** as requirements change
    * **Secuity design review** : a major milestones, review security design and its interaction with functionality/requirements
    * **Design documentation** : up-to-date document of requirements and functionality with security assessments
  - **Implementation** : During implementaion, the design mau be slightly refined and the security documents must be updated accordingly along with continuous reviews and analysis
    * Continuous **code review** ensure software is built according to specification and checked for bugs
    * **Static analysis** ensures high code quality and highlights flaws
    * **Vulnerability scanning** of external dependencies for exploits
    * **Unit tests** ensure functionality/security across components
    * **Accountability** : use a source code/version control system
    * **Coding Standards** : assertions and documentation
    * **Continuous integration** : run unit tests, static analysis, and linter whenver code is checked in
  - **Testing** : Completed components are rigorously tested before they are finally integrated into the prototype
    * **Fuzzing** is a form of probabilistic test integration
    * **Dynamic analysis** complements fuzzing with heavy-weight tests based on symbolic execution and models
    * **Third party penetration testing** provides external validation and clean slate testing
  - **Release** : Before release of the final prorotype, verify the base assumptions from the initial requirements analysis and design
    * **Security review** : check for compliance of security properties
    * **Privacy review** : check for privacy policy complicance
    * **Review all licensing agreements**
  - **Maintenance** : Afte shipping software, continuously maintain security properties
    * **Track third party software** and update accordingly
    * **Provide vulnerability disclosure contacts** through, e.g. a bu bounty program or a least a public contact
    * **Regression testing** : whenever an update is deployed recheck security and functionality requirements
    * **Deploy updates securly**

Policies and Attacks

Security prolicies

- A **policiy** is a deliberate system of principle to guide decisions and achieve rational outcomes. A policy is a statment of intent, and is implemented as a procedure of protocol.
- **Isolation** is the process or fact of isolating or being isolated. Two components are isolated if their intercations are resitrited.
- **Least privilege** ensures that each components has the minimal amount of privileges to function. If any privilege is removed, the component stops working
- **Compartmentalization** General technique of separating two more parts of a system to prevent malfunctions from spreading between or amoung them
  - Requires combination of isolation and least privilege
  - String policy to contain faults to single components
- **Memory safety** is a property that ensures that all memory accesses adhere to the semantics defined by the source programming language
  - Memory unsafe languages like C/C++ cannot enforce memory safety. Data accesses can occur through stale/illegal pointers

  - Memory safety prohibits buffer overflows, NULL pointer dereferences, use after free, use of uninitialized memory, or double free
  - A program is memory safe, if all possible executions of the program are memory safe
  - Runtime environment is memory safe, if all runnable programs are memory safe
  - A programming language is memory safe, if all expressible programs are memory safe
- Every memory safety is a bug
- A bug who's input can be attacker-controlled is a vulnerability
- The goals of the attack depends on your threat model
- **Memory safety violation** rely in two steps
  - Pointer foes out of bound or become dangling
  - The pointer is deferenced (used for read or write)
- **Spacial memory safety** is a property that ensures that all memory dereferences are within boudns of their pointer's valid object.
- **Temporal memory safety** is a property that ensures that all memory dereferences are vamid at the time of the dereference
- Policies defferences
  - **Object based** policies strore metadata (size, location) for each allocated object (none for pointer)
    - Allow you to check if an access targets a valid object
    - Cannot distinguish different pointers
    - Trade lower security for lower overhead
  - **Pointer based** polices strore metadata for each pointers
    - Allow you to verify if each access is correct for each pointer
- C/C++ **softbound** : compiler based instrumentation to enforce memory safety for C/C++
  - Initialize(disjoint) metadata for pointer when it is assigned
  - Assignment covers both creation of pointers and propgation
  - Check bounds whenever pointer is dereferenced
- Temporal memory safety is orthogonal to spacial memory safety
  - The same memory area can be allocated to new object
- **CETS** leverages memory object versioning. Each allocated memory object and pointer is assigned a unique version. Upon dereference, check if the pointer version is equal to the version of the memory object
  - Two failure versions
    - Area was deallocated and version is smaller
    - Are was reallocated to new object and the vesion is larger
- **Type safe** code accessed only the memory locations it is authorized to access.

Software bugs

- **Attack primitive** are building blocks for exploits
- Software bugs map to attack primitives
- A chain of attack primitive results in an **exploit**, the underlying bugs of the attack primitives become vunlnerabilites > nothing more in the slides ...

Attack vectors

- **Denial of service** (DoS): Prohibit legit use of a service by either causing abnormal service termination or overwhelming the service with a large number of duplicate/unnecessary requests so that legit requests can no longer be served
- **Leak information**: An abnormal transfer of sensitive information to the attacker. An information leak abuses an illegal, implicit, or unintended transfer of information to pass sensitive data to the attacker who should not have access to that data
  - Outout sensitive data
- **Code execution** allows the attacker to break out the restricted computation available through the application and execute arbitrary code instead. This can be achieved by (i) injecting new code, or (ii)

repurposing existing code through different means
  - **Control flow Hijacking** is attack primitive that allows the adversary to redirect control flow to locations that would not be reached in a benign execution. It requires :
    * Knowledge of the location of the code pointer
    * Knowledge of the code target
    * Existing code and control flow must be use the compromised pointer
  - **Code injection**: Instead of modifyign/overwriting code, inject new code into the address space of the process. It requites :
    * Knowledge of the location of a writable memory area
    * Memory area must be executable
    * Control flow must be hijacked/redirected to injected code
    * Contruction of shellcode
    * Code can be injected either on the *heap* or on the *stack*
  - **Code reuse**: Instead of injecting code, reuse code of the program. The main idea is to stitch together existing code snippets to execute new arbitrary behavior. It requires :
    * Knowledge of a writable memory area that contains invocation frames
    * Knowledge of executable code snippets
    * Control flow must be hijacked/redirected to prepared invocation frames
    * Contruction of ROP payload
  - **Data corruption**: This attack vector locates existing code and modifies it to execute the attacker's computation. It requires :
    * Knowledge of the code location
    * Area must be writable
    * Program must execute that code on benign code path
  - Code execution requires control over control flow
    * Attacker must overwrite a code pointer
    * Force program to dereference corrupted code pointer
- **Privilege excalation**: An unintended escalation (increase) of privileges. An attacker gains higher privileges in an unintended way. An example of privilege escalation is gaining administrator privileges through a kernel bug or a bug in a privileged program. A common example is setting the `is_admin` flag.
- Low level attacks start with a memory or type safety violation
- Accessing out-of-bounds violates **spacial memory safety**
- Accessing reclaimed object violates **temporal memory safety**
- Casting object into incompatible type violates **type safety**
- "*Writing shellcode is an art*"
- **Format string** are highly versatile, resulting in a flexible exploitation
  - Code injection: place shell code in string itself
  - Code reuse: encode fixed gadget offsets and invocation frames
  - Advanced code reuse: recover gadget offsets, then encode them on-the-fly
- **Type confusion attacks**
  - Control two pointers of different types ot single memory area
  - Different interpretation of fields leads to "opportunities"

Stopping Exploitation

Mitigations

- **Mitigations** make it harder to exploit a bug
- The bug remain in the program
- Factor for mitigation adoption
  - Mitigation of the most imminent problem
  - Very low performance overhead
  - Low developer cost
- **Data Execution prevention** (DEP)

  - Policy : data is never executable, only code is executable
  - CPU checks if page is executable on instruction fetch
  - Stop all code injection
  - Page table extension, introduce NX-bit (No eXecute bit)
  - This is an additional bit for every mapped virtual page. If the bit is set, then data on that page cannot be interpreted as code and the procesor will trap if control flow reaches that page
  - Attackers can still redirect control flow to existing code
- **Code reuse** : the Attacker can overwrite a code pointer and prepare the right parameters on the stack, reuse a full function (or part of a function)
  - Instead of targeting a simple function, we can target a gadget
    * Gadgets are a sequance if instructions ending in an indirect control flow transfer
    * Prepare data and environment so that pop instructions load data into registers
    * A gadget instruction frame consists of a sequence of 0 to n data valuers and a pointer to the next gadget. The gadget uses the data values and transfers control to the next gadget
- **Address Space Randomization** (ASR)
  - Sucessful contorl flow hijack attacks depend on the attacker overwriting a code pointer with a known aleternate target
  - ASR changes (randomize) the process memory layout
  - If the attacker does not know where a piece of code (or data) is, then it cannot be reused in an attack
  - Attacker must first learn or recover the adress layout
  - The security improvment of ASR depends on
    * the entropy available for each randomized location
    * the completeness of randomization (i.e. are all object randomize)
    * the absence of any information leak
- **Address Space Layout Randomization** (ASLR) is a practical form of ASR
  - ASLR focuses on blocks of memory (it is coarse grained)
  - Head, stack, code, executable, mmap regions
  - ASLR is inherently page-based, limiting cost of shuffling
- **ASLR entropy**
  - Entropy for each region is key to security (if all section are randomized)
  - Attacker follows path of least resistance, i.e. targets the obkect with the lowest entropy
  - Early ASLR implementation had low entropy on the stack and no entropy en x86 for the main executable
  - Linux (through Exec Shield) uses 19 bits of entropy for the stack and 8 bits of mmap entropy
- **Stack canaries**
  - Memory safety would mitigate this problem but adding full safety checks is infeasible due to high performance overhead
  - Instead of checking each pointer, check its integrity
  - Assumption: we only prevent RIP control flow hijack attacks
  - We therefore only need to protect the integrity of the return instruction pointer
  - Place a canary after a potentially vulnerable buffer
  - Check the integrity of the canary before the function returns
  - The compiler may place all buffers at the end of the stack frame and the canary just before the first buffer. This way, all non buffer local variables are protected as well
  - Limitation: the stack canary only protects against continious overwrites iff the attacker does not know the stack canary
  - An alternative is to encrypt the return instruction pointer by xoring it with a secret
  - Protects against most stack based hijack attacks
  - Simple, low overhead compiler based defense
- **Safe Exception Handling** (SEH)
  - Exceptions are a way of indirect control flow transfer in C++
  - A safe alternative to setjmp/longjmp or goto
  - Make control flow semantics explicit in the programming language

  - Exceptions allow handling of special conditions
  - Exceptions safe code safely recovers from thrown conditions
  - **Inline exception handling**
    * The compiler generates code that registers exceptions whenever a function is entered
    * Individual exception frames are linked across stack frames
    * When an excpetion is thrown, the runtime system traces the chain of exceptions frames to find the corresponding handler
    * This approach is compact but results in overhead for each function call (as metadata about exceptions has to be allocated)
    * Whenever you call a function, in the function prologue store all information on the stack, this makes it much easier to process but has cost for each function call (even if o exception is thrown)
  - **Exception tables**
    * Compiler emits per function orper objet tables that link code addresse to program state with respect to exception handling
    * Throwing an exception becomes a range query in the corresponding table and locating the correct handler
    * These tables are encoded very efficiently. This encoding may lead to security problems
    * For each code location, you mark in the table what kind of exceptions could be thrown. When an exception is thrown, use the return instruction pointers in the stack to walk backwards and infer where you are at and recover the information by walking the translation tables
- **Format string mitigations**
  - Deprecate use of `%n` (Windows); the is the sane option
  - Add extra checks for format strings (Linux)
    * Check for buffer overflows if possible
    * Check if the first argument is in read only area
    * Chek if all arguments are used
- **Arbitrary computation**: implement a Turing machine
- **Arbitrary code execution**: excecute any instructions
- **Forward-edge control-flow** transfers direct code forward to a new location and are used in indirect jump and indirect call instructions, which are mapped at the source code level to, e.g., switch statements, indirect calls, or virtual calls.
- The **backward-edge** is used to return to a location that was used in a forward-edge earlier, e.g., when returning from a function call through a return instruction. For simplicity, we leave interrupts, interrupt returns, and exceptions out of the discussion.

Advanced mitigations

- **Stack Integrity** ensures that the return instruction pointer and the stack pointer cannot be modified
  - **Return instruction pointers** are code pointers, stack integrity guarantees return instruction pointer integrity, i.e. only valid return instruction pointers are dereferenced
  - Pointer to other stack frames are stored on the stack, stack integrity ensures the integrity of this metadata. Note that modifying the base pointer indireclty modifies the return instruction pointer
  - **Stack canaries** are a weak form of stack integrity
  - **Shadow stacks** are a strong form of stack integrity
    * It is a second stack for each thread that keeps track of control data
    * Data on the shadow stack is integrity protected
    * Limitation: data corruption in uncaught
  - **Safe stacks** are a strong form with partial data protection
    * A shadow satck always keeps two allocated stack frames for each function invocation
    * Core idea: for each variable in a stack frame decide if it is safe
    * Variables are safe if they are only used in a safe context, i.e., they don't escape the current function and are only used with bounded pointer arithmetic
    * Push any unsafe variables to the unsafe stack

        * Performance benfit: an unsafe stack frame is only allocated if there are unsafe variables, i.e., if it is needed
        * Limitation: unsafe data corruption is uncaught
    - **Memory safety** gives full stack integrity
- **Control flow Integrity** (CFI) is a defence mechanism that protects applications against control flow hijacks. A sucessfull CFI machanism ensures that the control flow of the application never leaves the predetermined, valid control flow that is defined at the source code/application level. This means that an attacker cannot redirect control flow to alternate or enw locations
    - Limitations:
        * CFI allows the underlying bug to fore and the memory corruption can be controlled by the attacker. The defense only detects the deviation after the fact, i.e., when a corrupted pointer is used in the program
        * Over-approximation in the static analysis reduces security
        * CFI itseft is stateless, no dynamic control flow or data flow
        * Attacker may bend control flow along valid CFG
        * Attacker may run advanced data only attacks to jump between useful blocks
        * Assume fully precise CFI, no stack integrity
        * Divert control flow along this path and control argument
    - Attacks:
        * An attacker is free to modify the outcome of any JCC
        * An attacker can chosse any allowed target at each ICF location
        * For return instructions: in set of return targets is too broad ane even localized returns sets are too braod for most cases
- **Target set construction**
    - Core idea: restric the dynamic control flow of the application to the control flow graph of the application
    - A static analysis can recover an appriximation of the control flow graph (Precision of the analysis is crutial)
    - Trade off between precision and compatibility
    - On set of **valid functions** is highly compatible with other software but may result in imprecision given the large amount of functions
    - **Class hierarchy analysis** results in small sets but may be incompatible with other source code and some programmer patterns
- **Code Pointer Integrity** (CPI)
    - String memrory based defenses have not been adopted
    - Weaker defenses like strong memory allocators also ignored
    - Only defenses that have a negligible overhead are adapted
    - Code Pointer Integrity ensures that all code pointers are protected at all times
    - Instead of protecting everything a little protect a little completely
    - Strong protection for a select subset of data
    - CPI deterministically protects against CF hijacking
    - Sensitive pointers are code pointers and pointer used to access sensitve data
    - We can over approximate and identify sensitive pointer through their types: all types of sensitive pointer are sensitive
    - Over approximation only affects performance
    - Memory view is split into two views: control and data plane
        * The control plane is a view that only contains code pointers (and transitively all related pointers)
        * The data plane contains only data, code pointers are left empty (void/unused data)
    - The two planes must be seperated and data in the control plance must be protected from pointer dereferences in the data plane
    - CPI protects pointers and sensitive pointers
        * CPI enforces memory safety for select data
- **Sandboxing**

  - Kernel isolates process memory
    * The kernel provides the most well known form of sandboxinng
    * Process are sandboxed and connot access privileged intructions directly
    * To access resources, they must fo through a system call that elevated privileges and asks the kernel to handle the access
    * The kernel can then enforce security, fairness, access policies
    * Sandboxing in enable through HW, namely different privileges
  - `chroot` / containers isolate process from each other
    * Containers are lighweight form of virtualization
    * They isolate a group of processes from all other processes
    * Root us restricted to the container but not the full system
    * Sandboxing powered in SW, through kernel data structures
  - `seccomp` restricts process from interacting with the kernel
    * Seccomp restricts system calls and parameters accessible by a single process
    * Processes are sandboxed based on a policy
    * In the most constrained case, the allowed system calls are only : read, write, close, exit, sigreturn
    * Sandboxing powered in SW, through kernel data structures
  - Software absed Fault Isolation isolated components in a process
    * SFI restricts code execution/data access inside a single process
    * Application and untrusted code run in the same address space
    * The untrusted code may only read/write the untrusted data segment
    * Sandboxing is enabled through SW instrumentation

Finding bugs

Testing

- Testing is the process of analyzing a program to find errors
- An **error** is a deviation between observed bahaviour and specified behaviour
- "*Testing can only show the precesence of bugs, never their absence*" (Dijkstra)
- Testing checks whether implementation agrees with **specification**
  - Without specification, there is nothing to test
- A **Specification** defines illegal operations, given a policy
- Test cases detect bugs through
  - Assertions
  - Segmentation fault
  - Division by zero
  - Uncaught exceptions
  - Mitigations triggering termination
- Augment code with **checks** that validate your specification
- **Human testing**: have a human test the code
  - Debug by `printf`
  - Unit tests
  - Integration tests
  - Limitations
    * Requires manual effort to create each test
    * Tests must be kept up to date as specification evolves
- **Static analysis** analyze the program without executing it. Static analysis abstracts across all possible executions. The large amount of constraints often results in a state explosion limiting scalability
  - **Compiler warnings**: `-Wall -Wextra -Wpedantic`
  - **Fast checkers**: linters and per-unit checks
  - **Heavy-weight static analysis** clang checker
  - **Advantages**

        * Absolute coverage (no need for complete test cases)
        * Complete (test cases may miss edge cases)
        * Abstract interpretation (no need for a runtime environment)
    - **Disadvantage**
        * Computation depends on datan resulting in undecidability
        * Over-approximation due to imprecision and aliasing
        * May have large amounts of false positives
    - Can't check code you don't see
    - Can't checl code you can't parse
    - Not eveything is implemented in C
    - Not a bug
    - Not all bugs matter
    - False positives matter
    - False negatives matter
    - Annotations are extremely costly
    - **Requirements**
        * **Abstract domain**: what is computed by out static analysis
        * **Transfer function**: how are abstract values computed/updated at each relevant instruction
        * We must consider the instruction semantics for the transfer function
    - **Limitations**
        * Trade off between soundness and completeness
        * **Soundness** find all bugs of type X
        * Findind more bugs is good
        * Soundness is costly: checks are weak or complexity explodes
        * Diminishing returns: initial analysis finds most bugs, spend exponentially more time in few remaining bugs
        * **Formal verification**: 100 loc, find all bugs
        * **Bounded model checking**: 1000 loc, finds most bugs
        * **Symbolic execution**: 10 000 loc finds all bugs
        * **Concolic execution**: 50 000 loc finds bugs close to provided concrete execution
        * **Fuzzing** 1 000 000 loc, finds many bugs
        * **Warnings/simple anaysers** 100 000 000 locn find a lot of bugs interesting locations
- **Dynamic analysis**: analyze the program during a concrete execution. Dynamique analysis focuses on a single concrete run. The limited focus allows detailed analysis of a run but testing is imcomplete
    - **Whitebox testing**: aware of functionality specification
    - **Greybox testing**: partially aware of specification
    - **Blackbox testing**: unaware of specification
- **Continuous integration** systems run your tests at fixed intervals (dayly of even for each commit)
    - Only as good as your test case and setup
    - Ensures that changes comply with your specification
    - An active form of ongoing testing

Fuzzing

- **Fuzzing** finds reachable bugs effectively
- **General fuzzing**
    - Produces inputs according to input specification
    - Requires significant amount of manual work
- **Mutational fuzzing**
    - Generates unputs by mutating seeds
    - Most recent work focuses on mutational fuzzing
- The **taget program** is the binary you're searching bugs in
    - It needs to be en executable binary
    - It need to accept some form of input

  - Ideally, it crashes whenever you hit a bug
- The **execution environment**: the fuzzer executes the program with the generated input
  - Provide an environment for the progam
  - Create a process and feed input to the program
  - Make sure the environment is safe
- **Crash detection**: inputs that trigger a crash are put aside for later (human) analysis
  - Execution environment registers faults
  - May also detect resource exhaustion or hangs
  - Making program more likely to crash is a separate research area
- **Fuzzinq Queue**: Fuzzer maintains a queue of inputs to try next, executot picks first in queue
  - Queue initally filled with seed inputs
  - Seed inputs are reasonable starting inputs
  - Quality influences fuzzing results (as they serve as starting point)
- **Input mutation**: fuzzers must generate new inputs: pick an input and mutate it
  - Fuzzers often split mutation into deterministic and havoc phases
  - Mutation operators include flipping 1-2-4-8-16 bits, randomizing some part, replacing patterns, slicing inputs, merging inputs
- **Coverage collection**: Fuzzer tracks global coverage
  - Compare most significant bit of collected coverage with global coverage
  - If any edge increase, store seed

Symbolic execution

- SE in an abstract interpretation of the code
  - Symbolic values, not concret
  - Target conditions must be defined
- Agnostic to concrete values
  - Values turn into formulas
  - Constraints concretize formulas
- Finds concrete input
  - Triggers interesting condition
- Defines a set of code locations
  - Symbolic execution determines triggering input
- Testing: finding bugs in application
  - Infers pre/post conditions and add assertions
  - Use symbolic execution to negate conditions
- Generating Poc input through SAT solving
- **State exposition**: either the lenght of the constraints or the amount of state doubles at each loop iteration/conditional
  - Searching valid paths is crutial
  - Path optimization key research area
  - State prunning another key area
- **Binary processing**
  - **IR based Symbolic execution**
    - Easier to implement
    - Architecture agnostic
    - Easier queries to the solver
    - Less robust
    - Poor execution performance
    - Likely underconstrained
  - **IR-less Sysmbolic Execution**
    - Hard to implement
    - Architecture dependent
    - Harder queries to the solver

        * More robust
        * Good execution performance
        * Likely overconstrained
- **State manegment**
  - Issues
    * Too many states to track
    * Wasting time analysing useless path
    * States have too complex path constraints
  - Approach
    * **Hybrid Execution**
    * **Program summarization**
    * **Path Scheduling**
- **Hybrid execution** mix concrete and symbolic inputs to support symbolic analysis and increase code coverage
  - **Concolic execution**: Execute program with symbolic and concret inputs. Collect path constraints and use concrete values to help symb exec to get unstuck
  - **Symetric Symbolic Execution**: Use backward symbolic execution (BSE) and concrete execution to reason about specific target instruction
- **Program summarization**: Reduce the amount of generated states by using constraints of simplifications
  - **Function summarization**: Avoid paying the cost of re-executing the same functions over and over
  - **Loop summarization**: Avoid the cost of re executing the same loop every time state enters if
  - **State merging**: Model state progression using path constraints rather than generating new states
  - **Third party libraries summarization**: Use models to summarize side effect of non tracked function on the symbolic states
- **Path scheduling**: Manage paths exploration to reach more intersting program's state and avoid state explosion
  - Path running
  - Path priorization
- **Environment**
  - **Abstract Models**: Summarize a call to external procedure with a specific function
  - **Concrete Delegation**: Execution of external functions is delegated to the real system outside of the symbolic executor
- **Constraints Management**
  - **Constraints Reduction**: Sumplify the constraints with equivalents ones to speed up solving time
  - **Constraints caching**
  - **Constraints prediction**
- **Store management**
  - Single concretization
  - Forking model
  - Merge Model
  - Falt Model

Sanitizer

- Sanitizer instrument programs to check for violations, crash immediately
- Fuzzer trigger faults and record crashes, but not every fault crashes
- Sanitizers enforce a security policy to crash the program upon violation
- Sanitizers make faults detectable
- Program is analyzed during compilation (e.g. to learn such properties such as type graphs or to enable optimizations)
- The program is insrumented (e.g. to record metadata for checks and to execute policy checks at other places)
- At runtime, the instrumentation constantly verifies that the policy holds

- **AddressSanitizer** Detects memory erro
  - It places red zones aournd objects and checls those obejcts on trigger events. The tool can detect the folowing types of bugs
    - Out of bounds accesses to head, stack and globals
    - Use after free
    - Use after return
    - Use after scope
    - Double free, invalid free
    - Memory leaks (experimental)
  - Policy
    - Intrument each access, check for poison value
    - Advantage: fast checks
    - Disadvantage: Large memory, Still slow, Not a mitigation: does not detect all bugs
- **MemorySanitizer** detects uninitialized read. Memory allocations are tagged and uinitialized read are flagged
- **UndefinedBahaviorSanitizer** detects undefined behavior. It instruments code to trap on typical undefined behavior in C/C++ programs. Detectable errors are:
  - Unsigned/misaligned pointers
  - Signed integer overflow
  - Converion between floating point types leading to oveflow
  - Illegal use of NULL pointers
  - Illegal pointer arithmetic
- **ThreadSanitizer** finds data recas between threads. Focus ion data races involving writes
  - Multiple thread of execution share an address space
  - Accessing the ame variable requirs a protocol
  - Accesses must be ordered, if at least one thread writes
- **FuZZan** introduce alternate light-weight metadata structures
  - Reduce sparse Page Table Entries
  - Minimize memory management overhead
  - Runtime profiling to select optimal metadata structure
  - Dynamic switching mode : switch to selected metadata structure during fuzzing
  - Periodically measures the target program's behavior
  - Use fork server to avoid unnecessary re initialization
  - Modify ASan to disable the logging functionality

Summaries

- **Basis Security principles**
  - Sofware security goal: allow intended use of software, prevent unintented use that may cause harm
  - Security triad : Condidentiality, Integrity, Availability
  - Security of a system depends on its thread model
  - Concepts: isolation, least privilege, fault compartments, trust
  - Security relies on abstractions to reduce complexity
- **Secure software lifecycle**
  - Secure software development enforces security principles during software development
  - Software lives and evolves
  - Security must be first class citizen
    - Secure Requirements/specification
    - Security-aware Design (Threats?)
    - Secure Implementation (Reviews?)
    - Testing
    - Updates and patching
- **Security prolicies**
  - Security policies against specific attack vectors

  - Generic policies: isolation, least provileges, compartimentalization
  - Runtime policies : memory and type safety
  - Memory and type safety bugs are root cause of vulnerabilities
  - Memory safety: distinguish between spacial and temporal memory safety violations
    * Softbound: spacial memory safety, disjoint pointer metadata
    * CETS: temporal memory safety through versioning
  - Type sfae code accesses only the memory locations it is authorized to access
    * HexType: keep per object type metadata, explicit cast checks
  - Softbound, CETS, and Hextype are sanitizers: they trade performance for correctness during development
- **Software bugs**
  - Memory safety bugs allow state modification
    * Spacial memory safety focuses on bounds
    * Temporal memory safety focuses on validity
  - Type safety ensures that objects have the correct type
  - Large amounts of bug classes lead to *fun* vulnerabilities
- **Attack vectors**
  - Work with constrained resources (buffer size, limited control, limited information, partial leaks)
  - Control environment: write sehllcode or prepare gadget invocation frames
  - Execute outside ot eh defined program semantics
  - Attack vectors
    * Code injection (plus control flow hijacking)
    * Code reuse (plus control flow hijacking)
    * Heap versus stack
- **Mitigations**
  - Few defense mechanism have been adopted in practice. Know their strenghts and weakness
  - Data Execution Prevention stops code injection attacks, but does not stop code reuse attacks
  - Address Space Layout Randomization is probabilistic, shuffles memory space, prone to information leaks
  - Stack canaries are probabilistic, do not protect against direct overwrites, prone to information leaks
  - Safe Exception Handling protects exception handlers. reuse remains possible
  - Fortify source protects static buffer and format strings
- **Advanced mitigations**
  - Deployed defenses are incomplete and do not stop all attacks
    * Deployed: ASLR, DEP, stack canaries, safe execution
  - Control flow hijacking is the most versatile attack vector
  - Stack integrity guards the stack
  - CFI restricts targets on the forward edge
  - CPI prohibits control flow hijacking, key insight: enforme memory safety *only* for code pointers
  - Sandboxing seperates diffetent privilege domains
- **Testing**
  - Software testing finds bugs beforre an attacker can exploit them
  - Teting requires specification or policy
  - Anaysis checks if code follows policy/specification
    * Static analysis is over an abstraction domain
    * Transfer functions to transition between states
    * Static analysis must terminates
    * Differetn strengh of analysis
  - Be aware of laws of static analysis: check all code, find important bugs, trade-offs between false positives/negatives
  - Use the right tool for the job
- **Sanitizer**
  - Sanitizers allow early bug detection, not just on exceptions
  - Different sanitizers for different use case

* AddressSanitizer enforces probabilistic memory safety by recording metadata for every allocated object and checking every memory read/write
* MemorySanitizer targets uninitialized data
* ThreadSanitizer targets data races
* UndefinedBahaviorSanitizer finds different kinds of undefined behavior
* HexType targets type confusion