

Software security fiche

Pierre Colson

Contents

General	1
Basics	2
Basic security principles	2
Secure software lifecycle	3
Policies and Attacks	4
Security policies	4
Software bugs	5
Attack vectors	5
Stopping Exploitation	6
Mitigations	6
Advanced mitigations	8
Finding bugs	8
Testing	8
Sanitizer	8
Case study ?	8
Browser security	8
Web security	8
Mobile Security	8
Summaries	8

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 enforce 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 system that causes it to produce an incorrect or unexpected result, or to behave unintended ways. Bugs arise from mistakes made by people in either a program's source code or its design, in frameworks 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 benefit
 - 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/hinder 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 attacker. Threat models enable structured reasoning about the attack surface
- Security is
 - expensive to develop
 - expensive to maintain
 - may have performance overhead
 - may be inconvenient to users
- Fundamental security mechanisms
 - **Isolation**: Isolate two components from each other. One component cannot access 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 constrains an attacker in the obtainable privilege
 - **Fault compartments**: Separate individual components into smallest functional entity possible. These units 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 explanations. Abstraction allows an encapsulation of ideas without having to go into implementation details
- **OS abstraction**
 - *Single domain OS*
 - * A single layer, no isolation or compartmentalization
 - * All code runs in the same domain: the application can directly call into operating system drivers
 - * High performance, often used in embedded systems
 - *Monolithic OS*
 - * Two layers: the operating system 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 separate process
 - * Only essential parts are privileged

- * Applications request access from different OS process
- *Library OS*
 - * Few thin layers, flat structure
 - * 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 privilege 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 policy
 - * 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 *confidentiality*
 - * Biba : *read up and write down* => enforces *integrity*
 - **Discretionary Access Control** (DAC)
 - * Object owner specifies policy
 - * MAC requires central control, DAC empowers the user
 - * User has authority over her resources
 - * User sets permissions for her data if other users want access
 - * Examples : Unix Permissions
 - **Role-Based Access Control** (RBAC)
 - * Policies defined in terms of 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 boundaries. Define security specification, identify assets, assess 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
 - * **Security 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 implementation, the design may 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 whenever 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 prototype, 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 compliance
 - * **Review all licensing agreements**
- **Maintenance** : After shipping software, continuously maintain security properties
 - * **Track third party software** and update accordingly
 - * **Provide vulnerability disclosure contacts** through, e.g. a bug bounty program or at least a public contact
 - * **Regression testing** : whenever an update is deployed recheck security and functionality requirements
 - * **Deploy updates securely**

Policies and Attacks

Security policies

- A **policy** is a deliberate system of principle to guide decisions and achieve rational outcomes. A policy is a statement of intent, and is implemented as a procedure or protocol.
- **Isolation** is the process or fact of isolating or being isolated. Two components are isolated if their interactions are restricted.
- **Least privilege** ensures that each component 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 among 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 goes out of bound or become dangling
 - The pointer is dereferenced (used for read or write)
- **Spacial memory safety** is a property that ensures that all memory dereferences are within bounds of their pointer's valid object.
- **Temporal memory safety** is a property that ensures that all memory dereferences are valid at the time of the dereference
- Policies differences
 - **Object based** policies store 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** policies store 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 propagation
 - 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
 - * Area was reallocated to new object and the version 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 vulnerabilities > 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 processor 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 sequence of 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 values and a pointer to the next gadget. The gadget uses the data values and transfers control to the next gadget
- **Address Space Randomization (ASR)**
 - Successful control flow hijack attacks depend on the attacker overwriting a code pointer with a known alternate 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 address layout
 - The security improvement of ASR depends on
 - * the entropy available for each randomized location
 - * the completeness of randomization (i.e. are all objects randomized)
 - * 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)
 - Heap, 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 sections are randomized)
 - Attacker follows path of least resistance, i.e. targets the object with the lowest entropy
 - Early ASLR implementation had low entropy on the stack and no entropy on 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 continuous 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 exception 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 no exception is thrown)
- **Exception tables**
 - * Compiler emits per function proper object tables that link code addresses 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); this 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
 - * Check if all arguments are used
- **Arbitrary computation:** implement a Turing machine
- **Arbitrary code execution:** execute any instructions

Advanced mitigations

Finding bugs

Testing

Sanitizer

Case study ?

Browser security

Web security

Mobile Security

Summaries

- **Basis Security principles**
 - Software security goal: allow intended use of software, prevent unintended use that may cause harm
 - Security triad : Confidentiality, Integrity, Availability
 - Security of a system depends on its threat 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 policies**
 - Security policies against specific attack vectors
 - Generic policies: isolation, least privileges, compartmentalization
 - 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 safe 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 of 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