Resumos Teste 1

Basic Concepts

1.1 The Cybersecurity Problem

- **Vulnerabilities in Software**: Software commonly has bugs and security flaws, which can be exploited for unauthorized access or control.
- **Social Engineering**: A highly effective method for attackers, social engineering exploits human psychology (e.g., phishing) to bypass security.
- **Markets for Exploits and Malware**: There is a large, lucrative market for both malware and exploits, which allow attackers to take control of systems.

1.2 Cybersecurity Markets

- **Malware Market**: Involves selling malware, including spyware, ransomware, and botnets.
- **Cybersecurity Solutions Market**: Offers products and services for defending against these threats, including firewalls, antivirus software, and intrusion detection systems.

1.3 Real-World Cybersecurity Threats

- **Supply Chain Attacks**: Attackers compromise critical systems through indirect means, such as infecting software distributed by a trusted third party (e.g., the 2017 SolarWinds hack).
- **Critical Infrastructure Attacks**: These can target power grids, government institutions, and transportation systems, often with significant real-world impacts.
- **Device and Network Compromise**:
  - **Spyware and Surveillance**: Tools like Pegasus enable governments or malicious actors to monitor and control devices.
  - **Ransomware**: This self-replicating malware encrypts data and demands ransom (e.g., the Wannacry attack of 2017).
  - **Network Attacks**: Attackers may hijack IP addresses for activities like spam or launching Distributed Denial of Service (DDoS) attacks.

2.1 Compromising User Devices

- **Stealing Credentials**: Attackers target login details for banking, business, and social accounts.
- **Financial Spyware**: Often spread through phishing or malicious downloads, financial spyware monitors financial transactions and can manipulate account data.
- **Crypto-Mining Malware**: Attackers use compromised machines to mine cryptocurrency, using up the victim's processing power.
- **Botnets**: Networks of infected computers controlled by a central attacker. Botnets are often used for spamming, DDoS attacks, or other malicious activities.

2.2 Compromising Servers

- **Data Breaches**: Stealing sensitive user data, like credit card details or login credentials. High-profile breaches are common and have far-reaching impacts.
- **Geopolitical Hacking**: Attacks motivated by political goals, targeting national or critical infrastructure (e.g., Ukraine's power grid, the Stuxnet worm).
- **Compromising Users via Server**: Attackers can infect servers to target users indirectly. Examples include:
  - **Supply-Chain Attacks**: Compromising software distributed through trusted channels.
  - **Web Server Exploits**: Injecting malicious code on web servers that can infect users' browsers or systems.

2.3 Attacking Software Development Supply Chains

- **Typo-Squatting**: Attackers create malicious packages with similar names to popular software libraries, hoping developers will accidentally install the wrong one.
- **Supply Chain Security**: Even small errors, such as an incorrect package name, can introduce malicious code into software.

3.1 Definitions of Security

- **Security Principles**:
  - **Dependability in Adversarial Contexts**: Security is often defined as the system's ability to operate reliably even under malicious conditions.
  - **Protecting Confidentiality, Integrity, and Availability (CIA)**: Security protects data from unauthorized access (confidentiality), ensures data accuracy (integrity), and maintains system functionality (availability).
- **Security as a Defensive Practice**:
  - Unlike functionality, security is defined by preventing bad outcomes rather than ensuring specific actions occur. Security requirements vary based on context, application, and the types of threats involved.

3.2 Key Security Actors

- **Actors and Trust**: Security involves multiple actors (e.g., users, developers, administrators, third-party services) whose roles and access levels affect security requirements.

- **Trusted Third Parties (TTPs)**: Some security models assume certain actors or systems are inherently trusted. However, maintaining security may require assuming that actors can become compromised, motivating a layered defense approach (defense in depth).

3.3 Adversaries and Attacker Types

- **Adversary Profiles**:

  - **Script Kiddies**: Low-skilled attackers using readily available tools to exploit basic vulnerabilities.
  - **Occasional Attackers**: Attackers with moderate skill who actively investigate systems for potential weaknesses.
  - **Malicious and Organized Groups**: Sophisticated attackers, often in organized crime, who have clear motivations (e.g., profit, espionage).
  - **Nation-State Actors**: Government-sponsored hackers, often well-resourced, with political or economic goals.

- **Adversarial Motivation**: Attackers target the weakest link in security and are motivated by opportunities in the system that developers or security professionals may overlook.

3.4 Adversarial Thinking and Security Mindset

- **Adversarial Thinking**:

  - Effective security requires understanding how an attacker would view and exploit a system's weaknesses.
  - Security professionals must question assumptions, scrutinize potential security flaws, and think outside typical use cases to predict attack strategies.

- **Defensive Techniques**:

  - Adopt an "always skeptical" mindset to continuously evaluate the security of systems against potential threats.
  - Implementing defense in depth ensures that if one security layer is breached, additional measures are in place to protect the system.

Security Model

1. Security Models:

- **Correctness**: the system produces correct output for valid input.
- **Security**: the system should respond appropriately to invalid or harmful input.
- **Integrity**: data remains reliable and untampered.
- **Confidentiality**: sensitive data is protected from unauthorized disclosure.

2. Binary vs. Risk Management Models:

- **Binary Model**: common in cryptography, clearly defines the limits of an attacker's capabilities and whether a system is secure or insecure, does not scale for complex systems.

- **Risk Management**: focuses on minimizing risk based on likely threats and balancing cost-benefit in security measures; The risk analysis may be wrong and wea re never really secure if wrongly catalogued threats exist.

3. Security Lifecycle:

- Security is a continuous process, involving responses to new attacks and developing defenses.
- It requires a balance between **paranoia** (attack anticipation) and **rationality**.

4. Trade-offs Between Security, Usability, and Functionality:

- Known as the **CIA triad** (Confidentiality, Integrity, Availability), security involves balancing these core areas.
- Improving one causes issues in the others.

5. Risk Management and Asset Analysis:

- **Risk analysis** helps protect assets like information, reputation, and infrastructure.

- We manage the risk of relevant assets being improperly used or unavailable.

- An **asset** is a resource that holds value for an actor of the system, such as information, reputation, etc.

- The **risk matrix** evaluates threat probability and impact on each asset.

6. Key Terms:

- **Motive / Threat (CWE)**: a potential event that could cause harm defined according to their kind and origin.

- **Opportunity / Vulnerability (CVE)**: a flaw that an adversary can exploit originated by flaws such as bad quality softawre and misconfiguration.

- **Method / Attack (Exploit)**: an attempt to exploit a vulnerability (can be passive (eavesdropping) or active (guessing passwords, DOS)).

7. Threat Model:

- Defines **security objectives** by identifying assets to protect and adversaries (motivations, capabilities).
- Includes concepts like **security perimeter** (boundary of secure context) and **attack surface** (vulnerable points of contact).

8. Security Mechanisms and Policies:

- **Security Mechanism**: methods or tools (e.g., authentication, encryption, cryptography) used to implement **security policies**.
- **Security Policy**: a set of processes to protect the system according to the threat model (e.g., access control policy, password policy).

9. Security Engineering:

- Involves analyzing **security requirements**, defining **threat and trust** models, designing policies and mechanisms, and validating the solution.
- Security auditing and penetration testing (empirical or ethical) assess and strengthen system security.
- The **security mechanisms** are sufficient, together with the **security assumptions**, to implement the **security policies**.

10. Penetration Testing:

- A method to empirically validate security, simulating a real attack (can be “black-box” or “white-box”).

Software Security

1.1 The Creator vs. The Creature:

- **Software**, a creation of **developers**, can be exploited to work against its intended purpose and developers.
- **Vulnerabilities** arise from weaknesses in development, leading to errors and exploits that attackers use to manipulate the software.

1.2 Learning from History:

- Past vulnerabilities and exploits provide valuable lessons, highlighting areas to strengthen.
- Recognizing common flaws helps developers **anticipate and mitigate risks**.

1.3 Interaction of Users and Attackers:

- **Honest users (Alice)** interact with software as intended.
- **Malicious attackers (Mallory)* aim to exploit vulnerabilities in software, operating systems, or hardware through programming errors or security flaws.

1.4 Memory and Control Flow Vulnerabilities:

- Violations of “control flow integrity” allow attackers to alter program flow and execute arbitrary code.
- Software is especially vulnerable when it **accepts inputs from outside secure boundaries** (e.g., user input fields, network data).

1.5 Buffer Overflow and Memory Management:

- **Buffer overflows**, especially in low-level languages like C, are one of the simplest and most frequent vulnerabilities.
- Attacks occur when data “overflows” allocated memory, allowing attackers to **overwrite neighboring memory** and **control program flow**.

1.6 Stack Smashing:

- **Stack smashing** is a specific type of buffer overflow attack that targets the stack to overwrite the return address with a malicious value.
- This allows attackers to **control program execution** and **run their own instructions**, commonly using shellcode (a sequence of machine instructions to execute commands).

1.7 Memory Model in Attacks:

- Operating systems like **Linux** use a memory model where each process has virtual memory managed by the OS.
- **Attackers** manipulate memory boundaries to overwrite the **stack**, **heap**, or other memory segments, bypassing OS-level restrictions.

1.8 Control Hijacking and Shellcode Injection:

- **Shellcode** is a compact sequence of instructions, often written in assembly, that allows attackers to open a command shell or perform other unauthorized actions by using functions such as `strcpy` , `strcat` , `gets` and `scanf` .
- **Attackers** inject shellcode into memory and modify the return address to point to the shellcode, ensuring it executes when the function exits.

```
|--------------------------|
|        ...               |
|   Higher Memory Address  |
|--------------------------|
|  Return Address (f2)     | <--- Saved address to return to after `f1` completes
|--------------------------|
|  Old Frame Pointer (ebp) | <--- Previous frame pointer for `f1`
|--------------------------|
|  Local Variable 1 (f1)   | <--- Local variables of `f1`
|--------------------------|
|  Local Variable 2 (f1)   |
|--------------------------|
|  Parameter 1 (f1)        | <--- Parameters passed to `f1`
|--------------------------|
|        ...               |
|--------------------------|
|  Return Address (main)   | <--- Saved address to return to after `main` completes
|--------------------------|
|  Old Frame Pointer (main) | <--- Previous frame pointer for `main`
|--------------------------|
|  Local Variable 1 (main)  | <--- Local variables of `main`
|--------------------------|
|  Local Variable 2 (main)  |
|--------------------------|
|  Parameter 1 (main)      | <--- Parameters passed to `main`
|--------------------------|
|        ...               |
|--------------------------|
|   Lower Memory Address   |
```

2.1 Heap Buffer Overflow:

- **Unlike stack-based buffer overflows**, **heap overflows target dynamically allocated memory**, which makes them more challenging but potentially more impactful.
- By overwriting memory blocks and manipulating pointers, **attackers can disrupt program control**, enabling malicious code execution.

2.2 Exploit Targets in Memory:

- **Attackers** target vulnerable structures in memory, including:
  - **Function pointers** and **vTables** (in C++) for redirecting function calls.
  - **Exception handlers** for manipulating how exceptions are processed.
- By overwriting these memory locations, attackers control program flow.

2.3 Heap Manipulation Techniques:

- **Heap smashing** involves overwriting memory allocation structures (e.g., malloc/free pointers) to **redirect code execution** or **corrupt memory**.
- This attack requires carefully structured malicious code to align with the specific heap memory allocation patterns of the program.

2.4 JavaScript Exploits and Heap Spraying:

- **In browsers**, attackers use JavaScript to perform heap spraying, filling memory with "NOP sleds" and shellcode to increase the chance of executing malicious code.
- This technique is commonly used to **exploit vulnerabilities in web applications**, where attackers exploit memory models by flooding them with their code.

2.5 Other Memory Exploits:

- **Use-After-Free**: Accessing memory after it has been freed can lead to unpredictable behavior, allowing arbitrary code execution.
- **Integer Overflow**: Errors in arithmetic operations can cause incorrect memory allocations or buffer overflows, enabling attackers to control program flow.
- **Out-of-Bounds Read**: Attackers read beyond allocated memory to expose sensitive information.
- **Format String Vulnerabilities**: If format strings are improperly handled (e.g., in printf), attackers can read or write arbitrary memory.

3.1 Challenges in Control Hijacking with Modern Defenses:

- **Attackers face increasing difficulty in injecting code** and taking control of programs due to modern OS countermeasures, such as **Data Execution Prevention (DEP)** and **Address Space Layout Randomization (ASLR)**:
  - **Data Execution Prevention (DEP)**: Restricts memory regions, ensuring that writable memory cannot be executed and executable memory cannot be written to. This prevents direct code injection in many areas but has exceptions, such as for Just-In-Time (JIT) compilation.
  - **Address Space Layout Randomization (ASLR)**: Randomizes memory addresses of key program segments with each execution, making it difficult for attackers to predict the location of critical functions or code blocks.
- These countermeasures mean attackers can no longer rely on predictable memory layouts or easily inject and execute malicious code.

3.2 Return-Oriented Programming (ROP):

- In response to DEP and ASLR, attackers adapted by leveraging **Return-Oriented Programming (ROP)**, a technique that **reuses** existing code **instead of injecting** new code.
- **ROP Chaining**: Attackers link together small segments of existing, executable code, called **gadgets**. These gadgets are usually single instructions that end in a return `ret` instruction, which allows the attacker to chain them.
- **Advantages of ROP**: Bypasses **DEP**, since it doesn't require executable code injection, and can partially bypass **ASLR** if it reuses libraries at predictable addresses.
- **Challenges of ROP**: Attackers must carefully structure the stack to call these gadgets in sequence, which requires an intricate understanding of the program's stack and libraries.

3.3 Protection Mechanisms: Stack and Heap Defenses:

- **Stack Canaries**: A "canary" value, generated at runtime, is placed before each function's return address on the stack. **If a stack overflow alters the canary**, it triggers a **program shutdown** to prevent code execution.
- **Shadow Stacks**: Separate stacks, called shadow stacks, are maintained for control information. These shadow stacks store return addresses and function pointers, which the program verifies against the main stack before returning from a function.
- **Limitations**: While effective against simple buffer overflows, these defenses can be bypassed with more sophisticated techniques, such as ROP or exploiting other vulnerabilities (e.g., heap overflows or use-after-free).

3.4 Control Flow Integrity (CFI):

- **Purpose of CFI**: Protects the control flow of a program by limiting indirect jumps to valid targets, ensuring the program's flow matches a predefined control flow graph.
- **Implementation**:
  - **Basic CFI**: Direct function calls and jumps that use hard-coded addresses don't need protection, but indirect calls and returns, which attackers can manipulate, are validated against valid targets.
  - **Advanced CFI**: Uses cryptographic methods to store Message Authentication Codes (MACs) with each address. These MACs ensure that control-flow targets are genuine and unaltered.
- **Limitations**: CFI may introduce performance overhead and is challenging to implement with complete accuracy, as it requires precise computation of the program's control flow graph.

3.5 Defense Against Format String Exploits:

- **Format string vulnerabilities** occur when user-controlled input is passed into format specifiers (e.g., `printf`) without validation. This can expose or alter memory.
- **Mitigation**: Enforcing strict format string parameters prevents unauthorized memory access and modification, blocking attacks that rely on arbitrary format specifiers.

3.6 Leveraging Existing Code: Library Function Reuse:

- **Attackers may hijack functions in libraries** (e.g., libc) to perform actions without injecting new code. This includes using functions like `system` or `mprotect` by setting up **fake stack frames** that appear as legitimate **function calls**.
- This technique, **often paired with ROP**, allows attackers to execute sequences of malicious actions through legitimate system libraries, further complicating detection.

4.1 Common Security Vulnerabilities:

- Attack techniques like **stack/heap buffer overflows**, **use-after-free** errors, **integer overflows**, and **format string vulnerabilities** can all be exploited by untrusted input from outside the security perimeter.
- A recurring theme in these vulnerabilities is that **improper input validation** or **memory management** allows attackers to alter control flow or access sensitive data.

4.2 Defense in Depth:

- Security is most effective when applied at multiple levels, from language protections to hardware-based defenses. Layers include:

    - **Programming Language**: Safe languages (e.g., Rust, Java) inherently prevent many memory issues, like buffer overflows, by enforcing bounds checks and memory safety.
    - **Application-Level**: Techniques such as input validation, memory tagging, and runtime monitoring help detect hijacking attempts.
    - **Operating System-Level**: Countermeasures like **DEP** and **ASLR** prevent code injection and randomize memory locations, limiting predictable memory access for attackers.
    - **Hardware-Level**: Trusted execution environments (e.g., Intel SGX) and secure boot processes ensure secure initialization and runtime, particularly for critical processes and data.

4.3 Data Execution Prevention (DEP):

- **DEP/W^X (Write XOR Execute)**: Memory can either be writable or executable but not both, blocking typical code injection attacks.
- DEP is often implemented in hardware (e.g., NX bit) and enforced by the OS, which prevents executing code on data pages. However, it doesn't prevent all exploits (e.g., ROP attacks that reuse existing code).

4.4 Address Space Layout Randomization (ASLR):

- **Memory Randomization**: Each time a program runs, ASLR changes the memory locations of key segments, such as the stack, heap, and libraries, making memory addresses unpredictable.

- **Advanced ASLR**: Includes Kernel ASLR (KASLR) and Runtime ASLR, which add further randomization to kernel and runtime environments, reducing predictability.

- **Limitations**: While effective, ASLR can sometimes be bypassed by exploiting memory leaks or using techniques like heap spraying, which attempt to locate predictable memory structures.

4.5 Compiler Countermeasures:

- **Position-Independent Executables (PIE)**: Compilers generate code that can execute from variable addresses, complicating address prediction.

- **Stack Canaries**: Small, random values are inserted before the return address on the stack. If the stack canary changes, it indicates a stack overflow, triggering program termination.

- **Runtime Monitoring**: Compilers may add code to observe critical sections of code execution, halting execution if an anomaly is detected.

4.6 Memory Tagging and Hardware Extensions:

- **ARM's Memory Tagging Extension**: Tags are added to pointers and memory regions, with mismatched tags triggering exceptions. Memory tagging ensures that only valid pointers can access designated memory.

- **Trusted Execution Environments and Secure Boot**: Environments like Intel SGX and ARM TrustZone establish secure runtime zones, isolating critical code and data from potentially malicious code.

4.7 Control Flow Integrity (CFI) and Secure Compilation:

- **Control Flow Validation**: CFI prevents control hijacking by validating jumps and returns against known, permitted destinations, securing control flow paths.

- **Taint Analysis**: User inputs are marked as "tainted" and carefully checked to prevent unvalidated data from influencing sensitive operations, reducing vulnerability to injection attacks.

- **Secure Compilation**: Ensures no security issues are introduced during code compilation, protecting against unintentional vulnerabilities.

4.8 Challenges and Limitations:

- Even with multiple defense layers, determined attackers can sometimes **bypass protections by chaining vulnerabilities** (e.g., Trident/Pegasus exploit).
- **Defensive programming, memory safety**, and **secure design** are essential but often add performance overhead. These techniques help but do not guarantee absolute security, as some threats remain (e.g., Denial of Service).

Systems Security

1.1 Principles of Security

- **Economy of Mechanism**: Simplify systems by limiting features to only essential functions. Simplicity helps reduce vulnerabilities, eases validation, and improves usability.

- **Fail-Safe Defaults**: Systems should deny access by default, allowing only necessary permissions. For example, a new user account should have minimal permissions until explicitly granted.

- **Complete Mediation**: Every access to a resource must be checked against a security policy. This ensures that unauthorized access is consistently prevented.

- **Open Design**: Security should not rely on secrecy. Systems should be open to scrutiny, with only critical data (e.g., cryptographic keys) kept secret.

- **Separation of Privilege**: Functionalities and resource usage shall be compartmentalised and each compartment must be isolated from others, in a separate trust domain.

- **Least Privilege**: Users, programs, and processes should operate with only the permissions needed to complete tasks, reducing the impact of potential security breaches.

- **Least Common Mechanism**: sharing of resources of access mechanisms shall be minimised, especially when users have different levels of privilege, such as not reusing passwords across accounts in different services.

- **Defense in Depth**: Also called the "Castle Approach," this principle advocates multiple security layers to protect against failures in any single layer. For example, firewalls, intrusion detection systems, and strong access control work together to protect a network.

- **Psychological Acceptability**: Security features should not make the system difficult to use, as overly complex security requirements may be ignored or bypassed.

1.2 Access Control Models

- **Access Control Matrix**: Defines possible actions (**permissions**) an **actor** (user or process) can perform on **resources**, such as read or write. While effective, it can be complex to manage on a large scale.
- **Access Control Lists (ACLs)**: ACLs are specific permissions (resource, operation) attached to resources. They store access rights on each resource, such as read/write access to files.
- **Capabilities**: Opposite to ACLs, capabilities focus on actors by attaching permissions to users or processes, making it easier to manage user rights across multiple resources.
- **Role-Based Access Control (RBAC)**: Assigns permissions to roles rather than individuals. This model is common in enterprises, as it simplifies management by grouping similar permission sets.
- **Attribute-Based Access Control (ABAC)**: Permissions are based on attributes (e.g., user roles, resource sensitivity, context), allowing for flexible, policy-based access decisions.

1.3 Quiz & Example Scenarios

- Security Breach Examples: Real-world breaches illustrate these principles:
  - Email Virus Example: Opening an infected attachment violates Least Privilege if the virus gains user-level permissions.
  - OAuth Example: OAuth’s distributed authorization model aligns with Separation of Privilege and Least Common Mechanism by delegating authorization roles and allowing shared resource access only through authorized actors.

2.1 Kernel Security and User Mode

- **Kernel Basics**: The kernel is the core part of the OS, managing access to system resources and executing with full privileges (kernel mode). Only it can perform sensitive tasks, such as controlling hardware.
- **Privilege Levels**: Processors provide different privilege levels, with the highest privileges reserved for the kernel. User mode is a restricted environment in which applications run without direct access to hardware, minimizing the risk of system compromise.
- **Kernel Protection Mechanisms**:
  - ***Memory Isolation***: Kernel memory is separate and inaccessible to user-mode processes. Processes communicate with the kernel through controlled system calls.
  - ***Separation of Privilege***: Specific privileges are segmented across different parts of the system to ensure minimal risk from individual components. For example, the kernel enforces Least Privilege by restricting access levels for different user roles (administrators, standard users).

2.2 Monitoring and System Calls

- **System Calls** (Syscalls): Syscalls are functions that user-mode applications use to interact with the kernel (e.g., accessing files, executing processes, network communications).
- **Reference Monitor**: This is a critical mechanism that validates every system call against security policies, ensuring Complete Mediation. It is always active, preventing any unauthorized resource access.
- **Fail-Safe Defaults**: If a system call fails or encounters an error, the system defaults to a conservative security setting to avoid unintended access.

2.3 Process Isolation and User Permissions

- **Process Creation and Isolation**: The kernel isolates processes to ensure they operate independently and securely, using mechanisms like unique process IDs (PIDs) and memory segmentation.
- **Inter-Process Communication (IPC)**: Processes communicate via controlled IPC mechanisms such as pipes, shared memory, and sockets, ensuring information flow complies with security policies.
- **User Permissions and UID/GID**: Processes inherit the user ID (UID) and group ID (GID) of the user who created them. In Linux, the root user (UID 0) has maximum permissions, while other users have more limited access.
- **Process Tree**: In Unix-like systems, processes are managed hierarchically. For example, the init process (PID 1) starts all other processes. Each child process inherits permissions from its parent, and certain tasks are reserved for processes created by root or privileged users.
- **Daemons and Services**: These background processes (e.g., logging services, indexing) run with specific privileges, often higher than regular user processes, to perform essential OS tasks.

3.1 File System Permissions

- **Basic Permissions** (r, w, x): Files and directories on Unix-like systems have three primary permissions: read (r), write (w), and execute (x). These permissions can be applied to the owner, group, and other users.
- **User and Group IDs**: Each file or directory is associated with a user and a group. Permissions allow for finer access control, enabling or restricting actions like viewing, modifying, and executing files.
- **Extended ACLs**: Advanced permissions extend the basic model by specifying permissions for named users and groups beyond the primary owner and group.

ACLs can control access more granularly, supporting specific needs.

3.2 Privilege Escalation and Security Models

- **setuid and setgid Bits**: These special permission bits allow executables to run with the privileges of the file owner rather than the user running the file. This is helpful for tasks requiring elevated privileges, like password changes, but also poses risks if misused, leading to privilege escalation vulnerabilities.
- **Discretionary Access Control (DAC)**: The default model in Unix-like systems, DAC allows file owners to set permissions. However, it can be risky if mismanaged since owners can grant excessive permissions.
- **Mandatory Access Control (MAC)**: Used by systems like SELinux, MAC enforces strict access controls based on policies set by the administrator, regardless of the file owner's preferences. This model provides additional security, especially in environments with sensitive data.

3.3 User Authentication, Login Process, and Security Mechanisms

- **Login Process**: The system initially authenticates users by running a login process with `root` permissions. After verifying credentials, the process drops privileges to the user's UID and launches a shell with those permissions. This reduces the risk of unauthorized access.
- **setuid Program Risks**: Programs with the setuid bit (like passwd) are necessary for certain privileged actions, but they can be exploited if not secured properly. For example, bugs in setuid programs can allow unauthorized users to gain root privileges.
- **Privilege Escalation Examples**:
  - CVE-2004-0360: A vulnerability in `passwd` allowed unauthorized access through buffer overflow.
  - CVE-2023-22809: A vulnerability in `sudoedit` exposed systems to privilege escalation attacks by allowing unintended modifications.
- **Effective User ID (EUID), Real User ID (RUID), Saved User ID (SUID)**: These IDs help manage privilege transitions in processes. A process can change its effective user ID temporarily (using seteuid), allowing it to reduce or restore privileges based on the task. Mismanagement of these transitions, however, can result in security risks.

4.1 Confinement Techniques and Containers

- **FreeBSD Jails and Linux Containers**: Jails and containers limit resource access and enforce isolation by creating separate runtime environments for applications. Linux containers, which use namespaces and cgroups, isolate userland processes, while cgroups control resources (CPU, memory, network) for efficient resource management.
- **Docker**: A popular container technology built on LXC (Linux Containers), Docker uses namespaces and cgroups for confinement. Docker isolates each container's environment to prevent processes in one container from accessing resources in another. Docker Compose helps orchestrate multiple containers, providing a structured and scalable way to manage isolated environments.

4.2 System Call Interposition (SCI) and seccomp-bpf

- **System Call Filtering (seccomp-bpf)**: This Linux feature allows fine-grained control over system calls. Once a process is restricted by seccomp-bpf, only approved syscalls are permitted. This filtering provides additional isolation, preventing unwanted access by monitoring and blocking risky operations.
- **Docker and seccomp-bpf**: Docker uses seccomp-bpf to control container syscall access. Containers are sub-processes of the Docker daemon, and seccomp-bpf policies restrict the syscalls available to these sub-processes, reducing the risk of security breaches within containers.
- **Challenges of ptrace**: Early syscall monitoring using ptrace was limited by inefficiencies and vulnerabilities (e.g., TOCTOU—Time of Check, Time of Use—race conditions), as monitoring and actual execution of syscalls were not always atomic. Seccomp-bpf offers a more robust solution.

4.3 Virtual Machines (VMs) and Hypervisor Isolation

- **Hypervisors**: The hypervisor layer, which can be Type 1 (bare-metal) or Type 2 (hosted), creates isolated virtual environments by partitioning hardware resources. Type 1 hypervisors are typically used in cloud environments for improved security, as they directly control the hardware, bypassing the need for an OS layer.
- **Isolation in Virtual Machines**: VMs provide a high level of isolation, enabling multiple operating systems to run securely on the same hardware. Even if one VM is compromised, the others remain secure because they are isolated by the hypervisor.
- **Practical Applications**:
  - **NSA NetTop**: Uses virtualization to separate classified and unclassified data on the same hardware, ensuring that privileged information cannot be accessed by compromised processes.
  - **Cloud Providers**: Use virtualization to host different client OSes on the same physical hardware, with hypervisors managing isolation.
- **Side-Channel Attacks**: Modern attacks like Meltdown and Spectre exploit shared hardware features, such as CPU caches, allowing attackers to infer data from other processes, revealing limitations in the isolation offered by virtualization.

4.4 Detection of Virtualization and Trust Models

- **Virtualization Detection**: Some applications, including malware, attempt to detect if they are running in a virtualized environment (e.g., latency checks, hardware feature detection). This is useful for security research but can be leveraged by attackers to evade detection.
- **Red Pill vs. Blue Pill**: In virtualization security, the Red Pill refers to discovering that a system is virtualized. Techniques to detect VMs involve measuring hardware responses or memory latency, as the resources available in a virtualized environment differ from those in physical hardware.
- **Advanced Threats**: Attackers can use virtual machine detection to avoid executing in analysis environments, as seen in some malware. Some security researchers, such as Joanna Rutkowska (founder of Qubes OS), have developed techniques for "virtualization rootkits," which manipulate the hypervisor layer for stealth and persistence.

Web Security

1.1 HTTP Protocol and Structure

- **HTTP Basics**: HTTP is a stateless protocol used by clients to request resources from a server, commonly through URLs. Basic HTTP methods include:
    - `GET` : Retrieve data.
    - `POST` : Submit data.
    - `PUT` : Update data.
    - `PATCH` : Modify part of a resource.
    - `DELETE` : Remove a resource.
- **Response Structure**: HTTP responses consist of a status code and a message body, defining the success or failure of a request.

1.2 Cookies and Session Management

- **Cookies**: Used to store session information that identifies the user across different requests. Cookies are tied to a domain and are automatically sent with subsequent requests to that domain.
- **Session Management**:
    - **Same-Origin Policy (SOP)**: Only requests from the same origin as the server can access certain resources or read cookies, providing a layer of access control.
    - **Same-Site Attribute**: Controls which sites can access the cookie:
        - **Strict**: Only same-origin requests can use the cookie.
        - **Lax**: Allows limited cross-site requests, such as following a link.

1.3 Browser Execution Model and Client-Side Technologies

- **Dynamic Content and DOM**: Modern web applications render **HTML**, **CSS**, and **JavaScript** dynamically, where JavaScript manipulates the DOM (Document Object Model) to create interactive web pages.
- **Frames and iFrames**: Enable embedding of content from different origins, often for sandboxing or secure display of external data.
- **Trust Model**:
    - In web interactions, each component (user, browser, server) may be a potential attacker. Browsers enforce the SOP to manage these interactions and limit data access to trusted domains.

2.1 Web Security Models

- **Same-Origin Policy (SOP)**: SOP restricts scripts on a web page from accessing data from another origin, ensuring confidentiality and integrity of user data across domains.
- **Cross-Origin Resource Sharing (CORS)**: Allows servers to relax SOP by explicitly permitting certain other origins to access their resources, controlled via headers like `Access-Control-Allow-Origin` .

2.2 Injection Attacks

- **SQL Injection (SQLi)**: Occurs when malicious SQL code is embedded into database queries, allowing unauthorized access or modification of data. Prevention methods include:
    - Parameterized Queries: Prevent dynamic query generation from user inputs.
    - ORM Libraries: Use Object Relational Mapping (ORM) tools to abstract SQL syntax and sanitize inputs.
- **Command Injectio**: Exploits server-side scripts to execute arbitrary commands, potentially allowing full system access. Prevention includes strict input validation and avoiding direct command execution.

2.3 Cross-Site Scripting (XSS)

- **Reflected XSS**: Executes malicious scripts within a user's session by embedding the script in a URL or request, which the server reflects back.
- **Stored XSS**: Malicious code is stored on the server and affects any user who accesses the infected page or resource.
- **DOM-based XSS**: Occurs purely on the client side when JavaScript manipulates the DOM, without server interaction.
- **XSS Prevention**:
    - **Content Security Policy (CSP)**: Restricts which sources can run scripts on a page, enforcing a whitelist approach.
    - **Input Sanitization**: Validate and escape all user inputs to prevent executable code in the response.

2.4 Cross-Site Request Forgery (CSRF)

- **CSRF Attacks**: Trick users into performing unintended actions on a trusted website by exploiting cookies. SOP ensures that cookies are sent automatically, making these attacks possible.
- **Prevention**:
    - **CSRF Tokens**: Add a hidden, unique token to each form or request.

- **Same-Site Cookies**: Restrict cookies to same-origin requests to prevent unintended use across sites.

3.1 Insecure Direct Object References (IDOR)

- **IDOR Attacks**: Arise when applications expose internal object references, like user IDs, without adequate access controls. Attackers can modify identifiers to access unauthorized data.
- **Prevention**:
  - Implement fine-grained access control.
  - Avoid exposing sensitive object identifiers in URLs or request bodies.

3.2 Session Management and Security Misconfiguration

- **Session Management**:
  - **Server-Side Storage**: Tracks session state on the server, which is more secure than client-managed sessions.
  - **Client-Side Risks**: Storing session data in cookies or local storage can lead to tampering (cookie poisoning).
- **Secure Session Handling**: Use high-entropy session IDs, ensure HTTPS transmission, and set Secure and HttpOnly attributes to protect session cookies.
- **Security Misconfiguration**:
  - Common issues include exposed APIs, default credentials, or unprotected backups.
  - **Countermeasures**: Regular audits, restricting public access to critical endpoints, and disabling directory listings on the server.

3.3 File Upload and Clickjacking Vulnerabilities

- **File Upload Risks**: Allowing user file uploads can expose systems to malicious files, such as executable scripts or large files that consume server resources (zip bombs).
- **Clickjacking**: Tricking users into interacting with hidden elements of a web page by layering a transparent frame over it.
- **Prevention**:
  - **X-Frame-Options**: `DENY` or `SAMEORIGIN` prevents unauthorized sites from embedding content.
  - **Content Security Policy (CSP)**: Specifies trusted `frame-ancestors` and reduces the risk of clickjacking by defining which sites can display the page.