Short Question 1

Scenario: You are responsible for maintaining a server and notice unusual traffic spikes and network slowdowns. Upon investigation, you find that the SQL server was compromised due to a known vulnerability.

Question: What type of malware could exploit a buffer overflow vulnerability in a SQL server, and what are some of the immediate actions you should take to contain it?

Answer:

- **Type of Malware:** Malware like worms or SQL injection-based payloads could exploit buffer overflow vulnerabilities in SQL servers. These types of malware can inject malicious code into the server, potentially granting attackers control over server processes and enabling unauthorized access to sensitive data.
- **Immediate Actions:**
    1. **Isolate the Server:** Disconnect the compromised server from the network to prevent the malware from spreading or exfiltrating data.
    2. **Check Logs and Trace Activity:** Review server logs to identify any suspicious activity or unauthorized access. This helps understand the scope of the compromise.
    3. **Apply Patches and Updates:** If the vulnerability is due to outdated software, update the SQL server to the latest secure version to mitigate the specific vulnerability.
    4. **Scan for Malware:** Run a comprehensive antivirus and malware scan to identify and remove any malicious code.
    5. **Review Access Controls:** Ensure that only authorized users have access to the SQL server to prevent future attacks.

Short Question 2

Scenario: You are developing a new software application in C and want to ensure it's resistant to buffer overflow attacks. You plan to use stack protection mechanisms as part of your compile-time defenses.

Question: Explain how stack protection can help defend against buffer overflow attacks and describe any limitations associated with this approach.

Answer:

- **Stack Protection Mechanism:** Stack protection works by placing a "canary" value between local variables and the return address in the stack. If a buffer overflow overwrites the return address, it would also overwrite the canary value. Before executing a function return, the canary value is checked. If it has changed, the program will detect

the corruption and typically terminate, preventing code execution from a malicious buffer overflow.

- **Limitations:**
  - **Performance Impact:** Stack protection adds overhead to function calls, which may slow down program execution, especially in resource-constrained environments.
  - **Partial Protection:** Stack protection only guards against buffer overflows affecting the stack. It does not protect against overflows on the heap or other areas of memory.
  - **Bypass Techniques:** Some advanced attacks can bypass stack protection, such as by using return-oriented programming (ROP) to avoid altering the canary.

Long Question 1

Question: Discuss the various compile-time and run-time defenses against buffer overflow attacks. Compare these defenses in terms of effectiveness, performance impact, and practicality for different types of software applications.

Answer:

- **Compile-time Defenses:**
  1. **Safe Libraries**: Libraries with bounds-checking functions (e.g., `strncpy` instead of `strcpy`) help prevent overflows. These libraries are effective and have minimal impact on performance, making them practical for most applications.
  2. **Language Extensions**: Extensions in languages like C/C++ (e.g., `__attribute__((buffer_size))`) enforce buffer boundaries. While effective, these may introduce complexity in development and are less practical for high-performance applications due to performance trade-offs.
  3. **Stack Canaries**: Insert a canary value in the stack to detect overflows. Effective and widely used but can be bypassed in advanced attacks. Minimal performance impact makes it suitable for most software.
- **Run-time Defenses:**
  1. **Non-executable Memory Regions**: This prevents execution of code in certain memory areas (e.g., stack and heap). Highly effective for stopping execution-based attacks but may impact performance slightly. Practical for most modern applications.
  2. **Address Space Layout Randomization (ASLR)**: Randomizes the memory address space, making it harder for attackers to predict target addresses. Effective for high-security applications, with minor performance impact. Common in modern OSs.
  3. **Guard Pages**: Add non-accessible pages around critical memory areas to catch overflow attempts. Effective but can increase memory usage, making it less suitable for memory-constrained applications.

Comparison: Compile-time defenses are usually lightweight with minimal performance impact but offer partial protection, while run-time defenses are more comprehensive but may have a higher performance cost. Applications requiring high security should combine both for optimal protection.

Long Question 2

Question: Explain the differences between a worm and a virus in terms of propagation and impact. How do these types of malware typically exploit vulnerabilities in systems, and list specific countermeasures to prevent them.

Answer:

- **Propagation**:
  - **Worms**: Self-replicating and spread over networks without user intervention. They scan for vulnerable systems and exploit them to propagate.
  - **Viruses**: Require user action to spread (e.g., opening an infected file). They attach themselves to executable files or code.
- **Impact**:
  - **Worms**: Rapidly spread and cause network congestion, slowing down services.
  - **Viruses**: May corrupt or delete files and impact system integrity.
- **Common Exploited Vulnerabilities**: Both exploit software vulnerabilities like buffer overflows, unpatched systems, and weak access controls.
- **Countermeasures**:
  - **Network Level**: Use firewalls, intrusion detection systems (IDS), and restrict open ports to limit worm propagation.
  - **System Level**: Keep software up to date, enforce strong access controls, and run regular security scans.
  - **User Education**: Educate users on phishing and safe browsing to reduce virus infections.
- **Examples**:
  - **Worm**: The “ILOVEYOU” worm caused widespread damage in 2000 by exploiting email systems.
  - **Virus**: The “Michelangelo” virus targeted systems in the early 1990s, causing file and data corruption.

Short Question 3 (Quiz 3)

Scenario: You are an IT administrator and notice that several workstations are running slower than usual. Further investigation reveals a trojan has infected the network and is propagating by exploiting shared network resources.

Question: What is the Trojan's method of propagation, and what measures can you implement to prevent this type of malware from compromising your network in the future?

Answer:

- **Propagation Method**: The trojan spreads by exploiting shared network resources, potentially leveraging weak access controls or outdated software on shared drives.
- **Preventive Measures**:
    1. **Restrict Shared Resource Access**: Limit access to network shares and implement strong authentication mechanisms.
    2. **Apply Security Patches**: Regularly update all networked systems and shared resources to patch known vulnerabilities.
    3. **Monitor Network Activity**: Use IDS/IPS systems to detect unusual activity that may indicate malware spread.
    4. **Limit Privileges**: Apply the principle of least privilege, ensuring users have access only to resources necessary for their role.

Short Question 4 (Quiz 3)

Scenario: During a code review, you notice that a developer has used the `strcpy` function in a C program without checking the length of the input string. You are concerned about potential security risks.

Question: What vulnerability does the use of `strcpy` without length checking introduce, and how could it be mitigated?

Answer:

- **Vulnerability**: The use of `strcpy` without checking the length of the input string can lead to **buffer overflow** vulnerabilities. If an input string is longer than the buffer, it will overflow, potentially overwriting adjacent memory and allowing attackers to execute arbitrary code or crash the program.
- **Mitigation**: Replace `strcpy` with safer functions like `strncpy` or `strlcpy`, which allow specifying a maximum length, thus preventing overflow. Additionally, validate input lengths before copying to ensure they don’t exceed buffer limits.