Department of Computer Science

Computer Security

Exam 2

November 4, 2024

Solutions & Discussion

Prepared with tender loving care by Paul Krzyzanowski

100 POINTS – 25 QUESTIONS – 4 POINTS EACH – For each statement, select the *most* appropriate answer.

1. Hash pointers help in building tamper-evident data structures because a *hash pointer* contains:
   (a) An encrypted hash of the referenced object.
   (b) A hash of the referenced object.
   (c) A value that causes the resulting hash of the current object to have specific properties.
   (d) A hash of the pointer to ensure the pointer has not been modified.

   A hash pointer is a data structure that contains a hash of the referenced object along with a pointer to that object. This allows the data structure to be tamper-evident because any modification to the referenced object will change its hash, which can then be detected by comparing it to the stored hash in the hash pointer. This concept is widely used in blockchains and linked lists to maintain data integrity across linked records.

2. What data structure ***cannot*** be created with hash pointers?
   (a) Linked list.
   (b) Circular list.
   (c) Binary tree.
   (d) All of these can be created.

   In a circular list, the circular dependency means that one node would end up hashing a pointer to another node that ultimately points back to it, which complicates the use of hash pointers. This dependency loop makes it difficult to establish a unique hash chain, as each node's hash depends on the next in a way that creates an unsolvable cycle.
   Consider the list A→B→C→A. The hash pointer in A contains hash(B). The hash pointer in B contains hash(C) but the hash pointer in C would have to contain the hash(A), which would alter the value of hash(A) because hashing A includes the value of hash(B) and the value of hash(B) includes the value of hash(C). The degenerate case is A→A, which means finding a hash value such that hash(hash(A)+other data in A) = hash(A). The case of A→B→A means that A needs to have a hash(B) but then B needs to have hash(A) in its hash pointer, which is dependent on the hash(B).

3. Bitcoin provides *anonymity* by:
   (a) Encrypting your identity with your private key.
   (b) Hashing your identity.
   (c) Using your public key as your identity.
   (d) Storing transactions in an encrypted wallet that only you can access.

   Bitcoin provides pseudonymity (rather than true anonymity) by allowing users to transact using their public key as their identity. In Bitcoin, users are identified by their Bitcoin addresses, which are derived (hashed) from public keys and don't directly reveal personal information. While this provides some level of anonymity, it is not complete, as transactions are stored on a public ledger (the blockchain), and addresses can sometimes be linked to real-world identities through analysis and patterns.

   Note that a bitcoin address is a hash of the public key along with some error detecting codes. This is done to make the address shorter and easier to type. It does not add anonymity since anyone can look through the public ledger for transactions that receive bitcoin from that address. The transaction will have to contain the full public key that corresponds to the address.

4. How does the *difficulty target* in a Bitcoin block header influence mining?
   (a) It determines the maximum size of each block in bytes.
   (b) It limits the number of transactions per block.
   (c) It defines the maximum numeric value for the hash of the block header.
   (d) It sets a limit on the number of miners allowed to mine simultaneously.

   In Bitcoin mining, the difficulty target in a block header sets a threshold that the hash of the block header must be below for the block to be considered valid (currently, a 256-bit number where the 17 high-order bits are zero). Miners repeatedly modify the block header (by changing a value called the nonce) and re-hash it until they produce a hash that meets this difficulty requirement. This mechanism regulates the mining process, ensuring that blocks are mined at a steady rate by adjusting the difficulty based on network conditions.

5. A key feature of Bitcoin's *proof of work* scheme is that:
   (a) A system must prove that it used a given amount of CPU effort.
   (b) Computation is far more difficult than verification.
   (c) Multiple systems work cooperatively to mine cryptocurrency efficiently.
   (d) Blocks can be added more quickly to the ledger as more machines join the Bitcoin network.

A key feature of Bitcoin's proof of work scheme is that the computational effort to find a valid block hash (i.e., mining) is intentionally difficult, requiring substantial processing power. However, once a valid hash is found, the verification process (checking that the hash meets the required difficulty) is simple and quick. This ensures network security by making it computationally impractical for attackers to alter older blocks, as they would need to redo the proof of work for all subsequent blocks faster than the rest of the network can add new ones.

6. A *capability list* is:
   (a) A slice of an access matrix representing all the access permissions of different subjects on an object.
   (b) A complete access matrix implemented as a linked list.
   (c) A slice of an access matrix representing a subject's permissions to access various objects.
   (d) The set of possible access permissions that may be assigned to objects.

A capability list is a data structure that represents the access rights a specific subject (such as a user or process) has for various objects (like files or resources) within a system. In terms of an access matrix, it is essentially a row (or slice) of the matrix, showing what actions the subject is permitted to perform on each object. This approach contrasts with an access control list (ACL), which represents the permissions for all subjects regarding a specific object.

7. What is a *conflict class* in the Chinese Wall security model?
   (a) The grouping of objects based on competitive or sensitive relationships.
   (b) The set of discretionary access permissions that differ from mandatory access permissions for the same object.
   (c) A set of access control rules defining user roles within a company's internal security model.
   (d) The classification of users based on their security clearance levels.

In the Chinese Wall security model, a conflict class groups objects (such as files or resources) that belong to competing or sensitive entities. This model is designed to prevent conflicts of interest by restricting access to information within these classes. For example, if a user accesses an object in one conflict class (e.g., a file from Company A), they are then restricted from accessing objects in competing conflict classes (e.g., files from Company B), thereby preventing the potential for information leakage across competing entities.

8. The *Bell-LaPadula model* primarily focuses on:
   (a) Confidentiality of data.
   (b) Integrity of data.
   (c) Availability of data.
   (d) Authentication of users.

The Bell-LaPadula (BLP) model focuses on ensuring the confidentiality of data. It enforces access controls based on security clearances and classification levels following two key principles:
1. the *"no read up"* (simple security) rule, which prevents users from reading data at a higher classification level, and
2. the *"no write down"*(star property) rule, which prevents users from writing information to a lower classification level. These rules are designed to protect sensitive information from unauthorized access or leakage. Note that these rules do not prevent someone at a lower level from overwriting data at a higher level. The entire concern with BLP is not leaking confidential information.

9. In the Bell-LaPadula model, the *Star (*) Property* enforces the rule that:
   (a) Users cannot write to an object at a lower security level.
   (b) Users can only read and write data at their security level.
   (c) Users cannot read data from an object below their security level.
   (d) Users cannot write to an object above their security level.

In the Bell-LaPadula (BLP) model, the Star (*) Property (the "no write down" rule) enforces that a user with a particular security level cannot write information to an object at a lower security level. This prevents information at a higher security level from being inadvertently or intentionally disclosed to a lower security level, thus preserving data confidentiality. The other rule in BLP is the *"no read up"* rule (Simple Security Property), which prevents users from reading information above their security level.

10. In RBAC, a *role* can be described as:
    (a) A set of individual permissions to objects granted to a user.
    (b) Restrictions placed on a user based on information they previously accessed.
    (c) A temporary access level that overrides a user's normal access permissions.

(d) A group of access permissions associated with a particular job function.

In Role-Based Access Control (RBAC), a role is a collection of access permissions associated with a specific job function or responsibility within an organization. Instead of assigning permissions to each user individually, users are assigned to roles, and each role has predefined permissions. This simplifies access management and ensures that users have access only to the resources necessary for their roles, enhancing security. A user may take on different roles over time (even during the course of a day)

11. What is the primary focus of the *Biba Model*?
    (a) Enforcing confidentiality by restricting access to classified information.
    (b) Preventing unauthorized users from gaining access to a network.
    (c) Ensuring data integrity by preventing users from writing to a higher integrity level.
    (d) Maximizing availability by ensuring all processes have access to system resources.

The Biba Model focuses on data integrity rather than confidentiality. It enforces rules that prevent data at a higher integrity level from being compromised by data or actions at a lower integrity level. Key rules include the *"no write up"* rule, which prevents users from writing to a higher integrity level, and the *"no read down"* rule, which prevents users from reading information from a lower integrity level. These rules are designed to maintain the trustworthiness and accuracy of data within the system.

12. What is a typical use of *fuzzing*?
    (a) To ensure all code paths are tested.
    (b) To randomize pointers to ensure that they cannot be modified.
    (c) To provide runtime detection of buffer overflows. (partial credit: 3 points)
    (d) To find parts of code where inputs are not being validated.

Fuzzing is a testing technique used to identify vulnerabilities by providing unexpected, random, long, or malformed inputs to a program. This method helps discover areas in the code where input validation is inadequate, which can lead to issues like crashes, buffer overflows, or other security vulnerabilities. Fuzzing is particularly useful for uncovering weaknesses that may not be easily detected through standard testing approaches.

13. The standard C library *gets* function, which reads a line of input, is considered *unsafe* because:
    (a) It does not validate the format of the input.
    (b) It can allow users to execute arbitrary shell commands.
    (c) It runs at a higher privilege level.
    (d) It does not know the size of the array that will hold the data.

The *gets* function in the standard C library is considered unsafe because it does not limit the amount of data it reads, since it is unaware of the size of the array (buffer) where the input will be stored. This lack of bounds checking can lead to buffer overflows, where input data exceeds the allocated memory space, potentially overwriting adjacent memory. This vulnerability can be exploited by attackers to manipulate program behavior or execute arbitrary code, which is why gets has been deprecated in favor of safer functions like fgets.

14. A key part of most *code injection* attacks is their ability to:
    (a) Take advantage of parsing errors to get a program to treat an input string as a command.
    (b) Replace parts of the program's code with code provided by the attacker.
    (c) Replace the local data in a function with executable code.
    (d) Overwrite the return address of a function.

In many code injection attacks the attacker's goal is to overwrite the return address on the stack with a pointer to malicious code they have injected (or existing code in the case of return-to-libc or ROP). By doing so, the program's execution flow is redirected to the injected code when the function returns.

Choice (a) specifically mentions a *command injection* attack and not a *code injection*. The common attack vector for both code and command injection attacks is to exploit improper input handling by injecting malicious data through user inputs, which is treated as commands (in the case of command injection) or overflows a buffer (in the case of code injection).

15. A *return-to-libc* attack was designed to deal with:
    (a) Stack canaries.
    (b) NOP slides.
    (c) Data Execution Prevention .
    (d) Return Oriented Programming.

A *return-to-libc* attack is a type of attack used to bypass Data Execution Prevention (DEP), a security feature that marks certain memory regions (such as the stack) as non-executable to prevent code injection. Instead of injecting new code, a return-to-libc attack reuses existing, legitimate code, often in the C standard library, such as the *system()* function in libc, to execute malicious actions. By redirecting the flow to these pre-existing functions, attackers can achieve their objectives without injecting new, executable code, effectively circumventing DEP.

16. The purpose of *stack canaries* is to:
    (a) Detect if it is likely that a return address has been modified.
    (b) Keep any buffer overflow attack from overwriting local data.
    (c) Protect buffer overflows from leaving the stack segment.
    (d) Create a protected copy of the stack to guard against data corruption.

Stack canaries are special values placed between the local variables and the return address on the stack. Their purpose is to allow the code to detect buffer overflow attacks that attempt to overwrite the return address. If an overflow modifies the canary value (which it has to in order to reach the return address on the stack), the program detects this change before returning from the function and can terminate execution, preventing the attacker from hijacking the program's control flow. This makes stack canaries an effective safeguard against certain types of buffer overflow attacks.

17. Address Space Layout Randomization (ASLR):
    (a) Uses the memory management unit (MMU) to randomly change the virtual to physical memory mapping.
    (b) Makes it difficult for an attacker to inject valid addresses in a code injection attack.
    (c) Enables attackers to insert code into arbitrary regions of memory.
    (d) Hides the location of the program's heap from attackers.

Address Space Layout Randomization (ASLR) randomizes the memory locations of key data areas, such as the stack, heap, and libraries, each time a program is run. This makes it challenging for attackers to predict the memory addresses needed for a successful code injection or buffer overflow attack, as they cannot reliably guess where specific code or data is located. By disrupting predictable memory layouts, ASLR significantly increases the difficulty of crafting successful exploits based on memory addresses.

18. What is the purpose of *privilege separation*?
    (a) To separate high- and low-privilege functions into distinct processes to reduce the risk of exploits.
    (b) To separate users into distinct groups and define access policies for each group.
    (c) To grant multiple root-level (administrative) access to a system.
    (d) To assign each user a unique ID so that access can be managed on a per-user basis.

Privilege separation is a security design principle that involves separating high-privilege and low-privilege functions into different processes or components. This separation reduces the attack surface by limiting the amount of code that runs with high privileges. If a low-privilege component is compromised, the attacker gains only limited access, making it harder to escalate privileges and exploit the system. This approach is commonly used in applications and operating systems to contain potential security breaches.

19. *Command injection* can best be avoided by:
    (a) Using the principles of privilege separation.
    (b) Sanitizing the input.
    (c) Running the application in a sandbox.
    (d) Ensuring that buffer overflows are not possible.

Command injection vulnerabilities occur when untrusted input is passed to a system command without proper validation, allowing attackers to execute arbitrary commands. The best way to prevent this is by sanitizing the input—ensuring that input data is validated and stripped of any potentially dangerous characters or patterns. Additionally, using parameterized commands or safe APIs that do not allow direct command execution can help avoid command injection risks. While other security practices (like privilege separation or sandboxing) add layers of protection, input sanitization directly addresses the root cause of command injection vulnerabilities.

20. What is an attacker's goal in exploiting a *path traversal* vulnerability?
    (a) To trigger a bug in the pathname parser that will result in the program crashing.
    (b) To embed special characters that will result in parts of the name being treated as a shell command.
    (c) To gain unauthorized access to files outside of the allowed directory subtree.
    (d) To create a pathname so long that it overflows the allocated buffer.

A path traversal vulnerability occurs when an attacker manipulates input paths to access files or directories outside the intended or restricted directory. By using special characters like `../` in the input, an attacker can navigate up the directory structure, potentially gaining unauthorized access to sensitive files or system configuration data. Preventing this vulnerability involves properly validating and sanitizing file paths to ensure they remain within allowed directory boundaries.

21. Function *interposition* attacks involve:
    (a) Loading dynamic libraries that implement replacement function calls.
    (b) Generating user input that results in code injection.
    (c) A buffer overflow that transfers control to existing code in the application.
    (d) A heap overflow that overwrites the code in the running process.

Function interposition attacks exploit the ability to override or replace standard function calls by loading dynamic libraries that provide alternative implementations. In systems that support dynamic linking, an attacker can interpose on certain functions by creating custom versions, effectively altering the program's behavior by redirecting these calls. This technique is often used to intercept or manipulate function calls for malicious purposes, such as modifying program output or capturing sensitive data.

22. Which technique can help mitigate *TOCTTOU* (time of check to time of use) vulnerabilities?
    (a) Keeping clocks synchronized across all systems that interact with each other.
    (b) Allowing the process to recheck permissions.
    (c) Keeping any accessed files read-only.
    (d) Ensuring that checks and uses happen in an atomic operation.

TOCTTOU (Time of Check to Time of Use) vulnerabilities arise when there is a delay between checking a resource (like a file's permissions) and using that resource, during which an attacker can alter the resource. Mitigating TOCTTOU vulnerabilities involves ensuring that the check and use occur as a single atomic operation, meaning they are performed without interruption. This prevents an attacker from modifying the resource between the check and the use, reducing the risk of exploitation.

23. Linux *capabilities* are used to:
    (a) Allow a process to run with limited root-level privileges.
    (b) Let a user run programs with full root-level privileges even if the user is not root.
    (c) Restrict the amount of system resources a process can use.
    (d) Define which files a user is allowed to access.

Linux *capabilities* allow finer-grained control over privileges by breaking down root-level privileges into separate capabilities that can be independently assigned to processes. Instead of giving a process full root access, specific capabilities (such as the ability to bind to low-numbered ports or change file ownership) can be granted, limiting the process to only the privileges it actually needs. This minimizes security risks by reducing the number of processes running with full root-level permissions.

24. The UNIX *chroot* system call:
    (a) Assigns root privileges to a given user ID.
    (b) Changes the root directory of a process to a given subtree in the file system.
    (c) Restricts the allowable set of system calls a process can make.
    (d) Gives a process its own IP address & hostname.

The *chroot* system call in UNIX changes the root directory (`/`) for a process to a specified directory, effectively creating a "*chroot jail*." This limits the process's view of the file system, confining it to the specified directory and its subdirectories. By restricting file system access in this way, *chroot* can enhance security by isolating the process from other parts of the system, although it is not a complete security mechanism on its own.

25. *Sandboxing* allows:
    (a) Granting unprivileged applications limited access to privileged system calls.
    (b) Creating isolated environments with private network interfaces and process IDs. (partial credit: 2 points)
    (c) Automatically encrypting all data accessed by the application.
    (d) Restricting access to specific unprivileged system calls.

Sandboxing often involves filtering and restricting access to specific unprivileged system calls to limit what actions an application can perform. By controlling which system calls are available, the sandbox can prevent an application from performing operations outside its permissions or potentially harmful actions, thereby containing its behavior within safe bounds. This approach is commonly used in security-focused sandboxes, like those in web browsers or mobile operating systems, to enforce strict controls on application behavior. Choice (b) applies to containers and virtual machines, providing isolation but not restricting operations.

— The end —