Unit 3 Notes

Managing a Penetration test

Planning a Penetration Test

Penetration testing is a methodical process of evaluating the security of computer systems, networks, or applications by simulating attacks to identify vulnerabilities. Effective planning is essential for successful penetration testing and involves considering multiple factors that influence the test's approach, scope, and execution.

Types of Penetration Tests

White Box Testing

- Full Information Access: Testers receive complete internal information including network diagrams, source code, and configuration details
- Optimal Use Cases: Rapid examinations, compliance verification, secure-development validation, and code reviews
- Primary Advantages: High efficiency with no discovery phase, deep technical analysis, and effective remediation testing
- Key Limitations: Less realistic attack simulation, potential oversight of blind discovery vulnerabilities and social engineering risks
- Typical Outputs: Detailed vulnerability reports, source-level security issues, configuration errors, and prioritized remediation guidance

Black Box Testing

- No Prior Knowledge: Testers simulate external attackers with zero internal information
- Realistic Scenarios: External threat modeling, detection and response capability testing, red-team exercises
- Core Strengths: Authentic attack simulation, thorough evaluation of organizational visibility and defense posture
- Primary Challenges: Time-intensive execution, higher costs, potential missing of deep internal vulnerabilities

- Engagement Requirement: Longer testing periods necessary for comprehensive assessment depth

Gray Box Testing

- Limited Information Access: Testers receive partial credentials, network maps, or architectural insights
- Balanced Approach: Combines realism of black box with efficiency of white box testing
- Ideal Applications: Web application assessments, internal network testing with limited credentials
- Key Benefits: More realistic than white box, faster than black box, effective for insider threat scenarios
- Potential Issues: Possible oversight of blind attack paths, risk of testing bias from provided information
- Blind Attack Context: Refers to attacks where attackers have minimal or no prior target knowledge

Scope Definition

- Critical Planning Element: Defines exactly what systems, networks, and applications will be tested
- Financial Impact: Directly influences pricing and resource allocation for the engagement
- Client Inquiry Topics: Number and types of network devices, operating systems, web applications, evaluation duration, and physical locations
- Importance: Prevents scope creep and ensures clear expectations between tester and client

Location Planning

- Impact Factors: Affects travel requirements, effort allocation, and legal compliance considerations
- Site Selection Criteria: Based on critical systems location, user density, high-value targets, wireless exposure, and social engineering targets
- Practical Scoring: Evaluation of sites based on criticality, user population, exposure, security maturity, network footprint, regulatory requirements, third-party presence, operational impact, safety, and cost

- Sampling Strategies: Representative sampling across security levels, cluster sampling of nearby sites, critical-only focus, or rotating quarterly testing
- Logistics Requirements: Written authorization, permitted hours identification, local law compliance, travel arrangements, emergency planning, and insurance verification
- Deliverables: Scope matrices, prioritization spreadsheets, logistics plans, legal documentation, and sampling schedules

Testing Team Organization

- Team Leader (Engagement Lead): Manages scoping, legal approvals, stakeholder communication, scheduling, quality control, and final reporting with project management and security expertise
- Physical Security Expert: Conducts perimeter reconnaissance, lock testing, covert entry attempts, and CCTV validation using specialized physical security tools
- Social Engineering Expert: Executes phishing campaigns, vishing attacks, pretexting scenarios, and role-play assessments with strong interpersonal and legal compliance skills
- Wireless Security Expert: Performs war driving, rogue access point detection, encryption testing, and wireless mapping using RF analysis tools
- Network Security Expert: Conducts vulnerability scanning, firewall testing, exploit verification, and lateral movement analysis with network protocol and security tool expertise
- Operating System Expert: Focuses on host hardening, privilege escalation, patch management, and system configuration review across Windows, Linux, and macOS platforms

Testing Methodologies and Standards

- OWASP Framework: Focuses on web application security with the Top Ten vulnerabilities list, testing tools like ZAP, dependency checking, and secure development guidelines
- OSSTMM Manual: Provides standardized, measurable security testing methodology covering information, physical, human, wireless, and process security
- ISSAF Framework: Offers step-by-step structured security assessment approach for consistent, repeatable vulnerability identification and risk assessment

Testing Phases

- Phase I - External Testing: Includes footprinting, social engineering, and port scanning of external-facing assets
- Phase II - Internal Testing: Evaluates internal security capabilities once inside the network perimeter
- Phase III - Quality Assurance and Reporting: Involves validation of findings and comprehensive report generation

Agreement Structuring

- Statement of Work (SOW): Documents purpose, scope, limitations, time constraints, communication protocols, incident handling, deliverables, budget, and emergency contacts
- Get-Out-of-Jail-Free Letter: Provides written authorization for high-risk activities like physical penetration, wireless testing, dumpster diving, and social engineering, signed by senior organizational authority

Test Execution

- Kickoff Meeting: Aligns expectations, builds client relationships, and establishes testing ground rules
- Resource Requirements: Secure workspace, network access, identification credentials, and necessary equipment
- Expectation Management: Continuous communication, verified finding sharing, and cautious conclusion drawing
- Problem Resolution: Immediate issue reporting, collaborative problem-solving, and recurrence prevention
- Coordination: Contact list maintenance, off-hours communication, and real-time team information sharing

Results Reporting

- Report Structure: Includes table of contents, executive summary, methodology description, prioritized findings, and detailed appendices
- Vulnerability Prioritization: Critical (immediate remediation), High (six-month resolution), Medium (one-year fix), Low/Informational (optional addressing)
- Finding Organization: Grouped by business units or departments to enable targeted sharing and sensitive information protection

Basic Linux Exploits

Linux exploits are techniques or programs used to take advantage of vulnerabilities in Linux systems to gain unauthorized access, execute arbitrary code, escalate privileges, or disrupt services. These exploits typically abuse weaknesses in Linux applications, the Linux kernel, or system configurations to make the system behave in unintended ways.

Stack Operations

1. Stack Definition: A memory structure following LIFO (Last In, First Out) order used during program execution
2. Storage Contents: Function parameters, local variables, and return addresses
3. Key Stack Registers:
    - ESP (Extended Stack Pointer): Points to the top of the stack
    - EBP (Extended Base Pointer): Points to the base of the current stack frame
4. Stack Frame: Section of stack containing function parameters, return addresses, and local variables
5. Memory Direction: Stack grows downward (from higher to lower addresses)

Buffer Overflows

1. Definition: Occurs when a program writes more data into a buffer than it can hold
2. Mechanism: Excess data overwrites adjacent memory, including control information like return addresses
3. Consequences:
    - Program crashes
    - Arbitrary code execution
    - Unauthorized access
    - Privilege escalation
4. Example: Copying 35 bytes into a 10-byte buffer causes segmentation fault

Local Buffer Overflow Exploits

1. Attack Scope: Performed by users with local access to the system
2. Primary Targets: setuid/setgid programs (allow privilege escalation)
3. Exploit Components:
    - NOP Sled: Sequence of No Operation instructions (0x90) for reliable execution landing
    - Shellcode: Binary machine code performing attacker commands
    - Return Address: Overwritten to redirect execution flow
4. Key Challenge: ASLR (Address Space Layout Randomization) must be disabled for consistent exploitation
5. EIP Control: Attackers overwrite the saved EIP to redirect execution to malicious code

Exploit Development Process

Step 1: Control EIP

- Objective: Gain control over the Instruction Pointer
- Method: Send oversized input to trigger buffer overflow
- Verification: Use debugger (GDB) to observe crash behavior
- Confirmation: EIP contains attacker-controlled data (e.g., 0x41414141)
- Significance: Proves vulnerability is exploitable

Step 2: Determine the Offset(s)

- Goal: Find exact number of bytes required to overwrite EIP
- Tools: Metasploit pattern creation utilities
- Process:
    - Send unique cyclic pattern as input
    - Record overwritten EIP/ESP values during crash
    - Use pattern_offset to calculate precise overwrite position
- Result: Clean and accurate EIP control

Step 3: Determine the Attack Vector

- Analysis: Examine relationship between EIP and ESP
- Objective: Identify where shellcode can be reliably placed
- Common Strategy: Use instructions like "jmp esp" to jump to shellcode
- Consideration: Works even with stack randomization (ASLR)

Step 4: Build the Exploit Sandwich

- Layer 1: Padding/Junk bytes to reach EIP
- Layer 2: Overwrite EIP with attack vector address
- Layer 3: Shellcode placed where ESP points
- Format Requirements: No null bytes in addresses, little-endian format
- Construction: Structured payload assembly

Step 5: Test the Exploit

- Execution: Launch vulnerable application with crafted payload
- Observation: Monitor for successful shell or controlled execution
- Validation: Verify exploit reliability and consistency
- Next Step: Debug if exploit fails

Step 6: Debug and Refine

- Tools: GDB for detailed inspection
- Examination Areas:
    - Stack layout
    - Register values
    - Execution flow
- Adjustments: Modify offsets, NOP sleds, or addresses
- Goal: Achieve stable and reliable exploitation

Windows Exploit Development

Windows exploit development involves creating code that takes advantage of vulnerabilities in Windows systems to perform unauthorized actions like executing arbitrary code, privilege escalation, or bypassing security controls. This process uses specialized tools and follows a structured methodology to transform known vulnerabilities into working proof-of-concept exploits.

Types of Windows Exploits:

1. Buffer Overflow Exploits: Target stack or heap memory corruption vulnerabilities.
2. SEH Exploits: Abuse Structured Exception Handling mechanisms.

3. Kernel Exploits: Target the Windows operating system core.
4. DLL Hijacking: Replace or manipulate dynamic link libraries.
5. Exploit Chains: Combine multiple vulnerabilities for greater impact.

Tools for Windows Exploit Development:

1. Immunity Debugger: A variant of OllyDbg for analyzing and debugging Windows applications during exploitation.
2. Metasploit Framework: Provides exploit development modules, payload generation, and testing utilities.
3. Ruby Scripting: Used for writing exploit scripts and automating exploitation steps.
4. Microsoft Visual C++ Compiler: Compiles vulnerable programs for testing and analysis.

Exploit Development Methodology:

1. Control EIP: Gain control over the Instruction Pointer to redirect execution.
2. Determine Offsets: Calculate exact memory positions for reliable exploitation.
3. Identify Attack Vectors: Find suitable instructions or registers to execute payloads.
4. Build Exploit Payload: Construct the complete exploit with padding, return address, and shellcode.
5. Test and Debug: Validate the exploit and troubleshoot issues like bad characters.

Case Study: ProSSHD Vulnerability

- Target: ProSSHD SSH server with a 2010 buffer overflow vulnerability.
- Vulnerability Trigger: Sending more than 500 bytes in an SCP GET command.
- Exploitation Steps:
    1. Environment Setup: Configure vulnerable VM, start server, and launch debugger.
    2. EIP Control Verification: Use Ruby script to send oversized request and confirm EIP overwrite.
    3. Offset Calculation: Replace pattern with Metasploit-generated sequence to find exact offset (492 bytes).
    4. Attack Vector Selection: Use non-ASLR DLL (MSVCR71.dll) to find jump/push ESP instructions.

5. Payload Construction: Build exploit with padding, return address, NOP sled, and shellcode.
6. Debugging: Test for bad characters and refine payload until successful.

Key Windows Memory Protections & Bypasses:

1. /GS Stack Protection: Uses stack cookies to detect overflows. Bypass methods include cookie guessing, SEH overwrites, or function pointer manipulation.
2. SafeSEH: Validates exception handlers. Bypassed using modules without SafeSEH.
3. ASLR: Randomizes memory addresses. Bypassed by targeting non-ASLR modules.
4. DEP: Prevents code execution from data areas. Bypassed using ROP chains and VirtualProtect.
5. SEHOP: Validates SEH chain integrity. Bypassed by creating fake SEH chains.

Best Practices for Exploit Development:

1. Use Virtual Machines: Isolate testing environments to prevent system damage.
2. Take Snapshots: Save VM states for quick restoration during testing.
3. Validate Offsets Carefully: Use pattern tools for precise offset calculation.
4. Check for Bad Characters: Identify and exclude bytes that break shellcode.
5. Test Incrementally: Verify each exploitation step before proceeding.
6. Document Everything: Record offsets, addresses, and payload details for reproducibility.

Legal and Ethical Considerations:

1. Authorization Required: Only test systems you own or have explicit permission to test.
2. Responsible Disclosure: Report discovered vulnerabilities to vendors.
3. Educational Purpose: Exploit development should be for learning and improving security.
4. Compliance: Adhere to relevant laws and regulations regarding security testing.
    1.

Windows Memory Protection Mechanisms

Memory protection mechanisms are security features implemented in Windows operating systems to prevent and detect exploitation of software vulnerabilities. These protections work together to create multiple layers of defense against memory corruption attacks.

Structured Exception Handling (SEH)

1. Purpose: Gracefully handles runtime errors to prevent abrupt program termination
2. Structure: EXCEPTION_REGISTRATION record (8 bytes) containing two pointers:
    - Previous SEH record pointer
    - Exception handler function pointer
3. Storage: SEH chain stored in Thread Information Block (TIB), accessed via FS:[0] register
4. Termination: Chain ends with system default handler (Prev = 0xFFFFFFFF)
5. Handling Process:
    - Exceptions occur (access violation, divide-by-zero, etc.)
    - CPU transfers control to Windows kernel exception dispatcher
    - Dispatcher walks SEH chain from FS:[0]
    - Handler filter functions evaluated for each exception
    - If no handler catches exception, system terminates process

Windows Memory Protection Mechanisms

/GS Protection (Stack Canary)

1. Function: Detects stack-based buffer overruns using security cookies
2. Implementation: Random cookie placed between local variables and return address
3. Verification: Cookie checked before function return; program terminates if modified
4. Improvements:
    - Enabled by default in later Visual Studio versions
    - Buffers moved away from sensitive variables
    - Function arguments copied to top of stack frame

5. Limitations:
    - Doesn't protect functions without buffers
    - Doesn't protect buffers smaller than 4 bytes
    - Doesn't protect naked functions or functions with variable arguments
    - Disabled when optimizations are disabled

Safe Structured Exception Handling (SafeSEH)

1. Purpose: Prevents malicious overwrite of SEH handlers
2. Activation: Enabled via /SafeSEH compiler option
3. Mechanism: Binary contains table of valid exception handlers
4. Runtime Verification:
    - Handler address must not be located on stack
    - Handler must exist in pre-compiled allowed list
    - Handler must be in executable memory region
5. Effect: Prevents execution of attacker-injected exception handlers

SEH Overwrite Protection (SEHOP)

1. Introduction: Windows Server 2008 (later backported to other versions)
2. Purpose: Validates integrity of entire SEH chain
3. Mechanism: Ensures chain properly reaches FinalExceptionHandler
4. Validation: Walks SEH chain checking structural integrity
5. Detection: Identifies corrupted or fake SEH chains
6. Effectiveness: Prevents sophisticated fake chain constructions

Heap Protections

1. Target: Vulnerabilities in dynamic memory allocation
2. Historical Exploits: Heap metadata overwrites enabling arbitrary memory writes
3. Protection Mechanisms:
    - Safe Unlinking: Validates forward/backward heap pointers (XP SP2+)
    - Heap Cookies: One-byte random values in heap headers (XP SP2+)
    - XOR Encryption: Critical heap metadata encryption (Vista+)
4. Impact: Significantly increases difficulty of heap corruption attacks

Data Execution Prevention (DEP)

1. Principle: W^X (Write XOR Execute) - memory cannot be both writable and executable
2. Function: Prevents code execution from stack, heap, and data sections
3. Operating Modes:
    - OptIn: Only opted-in applications protected (default for XP, Vista, 7)
    - OptOut: All applications protected except exclusions
    - AlwaysOn: Cannot be disabled
    - AlwaysOff: Completely disabled
4. Response: ACCESS_VIOLATION exception on execution attempts
5. Effect: Prevents direct execution of shellcode in memory buffers

Address Space Layout Randomization (ASLR)

1. Introduction: Windows Vista with /DYNAMICBASE linker option (default enabled)
2. Function: Randomizes memory addresses to break exploit predictability
3. Randomized Components:
    - Executable images (EXE base addresses)
    - DLL images (library base addresses)
    - Stack memory regions
    - Heap memory regions
    - Process Environment Block (PEB)
    - Thread Environment Block (TEB)
4. Limitations:
    - 64KB page size alignment reduces entropy
    - Some sections vulnerable to brute-force attacks
    - Not all modules are ASLR-compatible
5. Impact: Forces address guessing rather than using hardcoded addresses

Protection Synergy

1. Defense in Depth: Multiple layers create comprehensive protection
2. Complementary Functions: Each protection addresses different attack vectors
3. Modern Exploitation: Requires bypassing multiple simultaneous protections
4. Evolution: Continuous improvement across Windows versions since XP SP3

Bypassing Windows Memory Protections

Memory protection bypass techniques are methods used by attackers to circumvent the security mechanisms implemented in Windows operating systems. These techniques demonstrate how determined attackers can overcome even sophisticated protections through creative exploitation strategies.

Bypassing /GS Protection Methods

Guessing the Cookie Value

- Vulnerability: Stack cookies sometimes use weak entropy sources
- Attack Type: Primarily effective in local attacks only
- Method: Attacker calculates or guesses the cookie value
- Execution: Overwrites stack while preserving correct cookie value
- Result: /GS verification passes despite successful buffer overflow

Overwriting Calling Function Pointers

- Target: Virtual table (vtable) pointers used by object-oriented functions
- Bypass Logic: /GS protects return addresses but not function pointers
- Attack: Attacker overwrites vtable pointer to redirect execution
- Result: Achieves code execution while completely avoiding cookie check

Replace the Cookie with Known Value

- Cookie Location: Stored in writable .data section of memory
- Prerequisite: Requires ability to perform arbitrary memory writes
- Method: Attacker replaces actual cookie with known value before overflow
- Advantage: Stack overflow uses predictable cookie, passing /GS validation
- Challenge: Requires initial memory corruption capability

Overwriting an SEH Record

- Key Insight: /GS does not protect Structured Exception Handling records
- Strategy: Trigger exception before function returns (before cookie check)
- Method: Attacker overwrites SEH handler pointer on stack
- Result: Exception handler executes, bypassing /GS entirely
- Follow-up: Leads directly into SafeSEH bypass techniques

Bypassing SafeSEH Protection

- Target Selection: Focus on modules compiled without SafeSEH protection
- Attack Technique: Use "POP POP RET" instruction sequence exploitation
- Method: Corrupt SEH record to point to non-SafeSEH DLL containing POP POP RET
- Execution Flow: Stack manipulation redirects to attacker-controlled shellcode
- Result: Bypasses SafeSEH's handler validation through module selection

Bypassing ASLR

- Core Challenge: Randomized memory addresses break hardcoded exploit values
- Attack Strategies:
    - Target modules not compiled with ASLR (/DYNAMICBASE)
    - Use non-ASLR enabled DLLs with predictable base addresses
    - Employ information leaks to reveal memory layouts
    - Implement brute-force attacks when entropy is limited
- Modern Approach: Combine multiple techniques to defeat ASLR progressively

Bypassing DEP

VirtualProtect/VirtualAlloc Method

- Concept: Use legitimate Windows API functions to mark memory as executable
- Functions: VirtualProtect() or VirtualAlloc()
- Key Parameters:
    - lpAddress: Memory region starting address
    - dwSize: Size of region to modify
    - flNewProtect: New protection flags (PAGE_EXECUTE_READWRITE)
    - lpflOldProtect: Storage for original protection
- Result: Attacker-controlled memory becomes executable legally

Return Oriented Programming (ROP)

- Core Principle: Reuse existing code fragments (gadgets) ending in RET
- Chain Construction: Connect multiple gadgets to perform complex operations

- Advantage: No new code injected; only legitimate code reused
- DEP Compliance: Execution occurs in already-executable regions
- Complexity: Requires finding and chaining multiple suitable gadgets

Bypassing SEHOP

- Validation Target: SEHOP checks chain integrity and termination
- Bypass Strategy: Create fake but structurally valid SEH chain
- Technique: Build chain ending with legitimate system default handler
- Instruction Use: XOR, POP, POP, RET sequences from non-SafeSEH modules
- Conditional Jump: Force JE instruction to redirect to attacker code
- Final Layer: NOP sleds guide execution to shellcode
- Appearance: Fake chain appears completely valid to SEHOP validation

Conclusion

1. Reality Check: No security protection is completely invulnerable
2. Defense Layers: Multiple protections work together to increase attack complexity
3. Attack Evolution: As protections improve, attack techniques evolve in response
4. Continuous Learning: Security professionals must stay updated on bypass techniques
5. Practical Application: Understanding bypass methods informs better defense design
6. Ethical Use: Knowledge should be applied to improve security, not compromise it

Key Takeaways

- Modern Windows protections significantly raise the bar for exploitation
- Bypasses often require multiple steps and sophisticated techniques
- Attackers increasingly combine methods to defeat layered defenses
- Security is an ongoing process requiring continuous adaptation
- Understanding bypass techniques is essential for effective defense design
- Responsible disclosure and ethical application of this knowledge is crucial