COMP 435 Quiz 5 Study Guide

Computer Security Concepts · UNC Fall 2026 · Everything since Quiz 4: digital signatures, software security, network security, HTTPS and PKI, XSS (Lectures 28–34 and WA5)

0. Quiz logistics and how to use this guide

When. Friday, November 20, 2026, in class (10:10–11:00am, Bingham 3014). It is the fifth and last quiz of the semester.

Weight. The five quizzes together are 40% of the course grade. If you take all five quizzes and your final exam score is higher than a quiz score, the lowest quiz is replaced by the final exam score.

Scope. Material since Quiz 4 (Oct 30): L28 digital signatures and intro to software security, L29 buffer overflows, L30 secure coding and intro to network security, L31 packet filters and IDS, WA5 packet sniffing and spoofing, L33 IDS math, tunnels, SSH, TLS and PKI, and the L34 browser / same-origin / XSS material that framed Lab 5. L27 was a guest lecture on multi-party computation with no posted slides.

What is in scope

Lecture	Date	Core ideas you must be able to explain
L28 Digital signatures; intro to software security	Nov 4	Definition, properties and requirements of a digital signature; Sign/Verify with a private signing key and public verification key; RSA signatures and why raw RSA is forgeable; hash-then-sign; encrypt-then-sign and which key does what; "note of caution"; quantum computers, Shor's algorithm and the post-quantum replacements; the stack we trust; integer overflow (the login counter and the `malloc(len * sizeof(int))` example); integer underflow.
L29 Buffer overflow and more software security	Nov 6	Error / fault / failure / flaw; buffer overflow vs overrun vs attack vs stack smashing; process memory layout; stack frames, return address, saved base pointer; the `strcpy` attack and shellcode; countermeasures (canary, W⊕X, ASLR) and the attacks that answer them (return-into-libc, ROP gadgets); backdoors; privilege escalation; guidelines for writing secure code.

L30 Secure coding wrap-up; intro to network security	Nov 9	Least privilege, sandboxing, type safety, memory safety; Heartbleed; network security challenges; security perimeter; transmission media; the four threats mapped to CIA; Dolev-Yao; firewalls, IDS, tunnels; firewall definition and properties; packet filters vs proxies; TCP/IP layers, IPv4 header, TCP segment and the three-way handshake.
L31 Network defenses	Nov 11	Packet filter definition; what rules can match; positive/negative, stateless/stateful, shallow/deep; firewall limitations; the mail rule set and its two fixes (source port, ACK flag); why a high port or port 25 or ACK proves nothing; signature-based filters and the split / reorder / TTL evasions; intrusion, IDS vs IPS, NIDS vs HIDS.
WA5 Packet sniffing and spoofing	Nov 13	Why the attacker sees the victim's pings; the forged ICMP echo reply (which fields are reversed, which are copied); why changing the ICMP identifier breaks it; what this says about source IP as authentication.
L33 From network defenses to HTTPS	Nov 16	IDS confusion matrix, TPR/FPR/TNR/FNR and alarm precision; base rate; the 50,000-event worked examples; tunneling and VPNs; SSH layers, server authentication (CA, key database, TOFU), user authentication; HTTP, TLS, encapsulation; PKI, certificates, chains, what the browser checks; the TLS handshake phases; "a certificate is not a safety rating".
L34 XSS (Lab 5 framing)	Nov 18	What the browser does with a page; DOM; origin = scheme + host + port; same-origin policy and origin matching; stored XSS; what injected script can read and do (cookies, GET via image, authenticated requests with the page's token, self-copying worm); defenses: output encoding, sanitizers, CSP, HttpOnly.

How to study with this guide

- **Each lecture section** follows the slides in order, expands the terse bullets into full explanations, and ends with "Check yourself" questions. Answers are hidden under a toggle so you can self-test.
- **Boxes:** blue "Key idea" boxes quote slide definitions nearly word for word. Orange "Watch out" boxes flag the mistakes the practice solutions warn about. Green "Exam tip" boxes tell you how a question on that topic was phrased in class.
- **Every in-class practice sheet since Quiz 4** (Nov 4, 6, 9, 11, 16), the WA5 reflection questions, and all 14 questions of Extra Practice 5 are worked in full here, using the instructor's posted solutions where they exist.
- **Practice exam** at the end mixes every topic in the formats used in class: fill in a packet-filter table, trace a stack diagram, compute IDS alarms and precision, complete a TCP handshake, name who signs a certificate, and explain a definition.
- **Cram sheet** at the very end is one screen per lecture. Read it last.

EXAM TIP. Extra Practice 5 says it covers "material since Quiz 4, through Lecture 33" and warns that it "is not an announcement of quiz scope", so treat L34's XSS material as fair game but secondary. The in-class sheets show the question styles the instructor likes: short calculations (ACK numbers, IDS alarm counts, RSA toy numbers), "which rule allows this packet", "where does execution jump", and multiple choice where the wrong options are tempting half-truths ("the server should never echo back a user-provided string"). Expect to be asked *why*, not just *what*: every posted solution ends with the reason.

L28 1. Digital signatures and the quantum footnote

L28 (Nov 4) finished the cryptography unit. The guiding question on the first slide: *what cryptographic tool(s) give us non-repudiation (aka accountability)?* The answer is the digital signature, the last box in the roadmap: hash functions and MACs on the integrity side, symmetric and public-key encryption on the confidentiality side, and digital signatures giving integrity plus authentication plus accountability.

1.1 Definition, properties, requirements

DIGITAL SIGNATURE. A means to ensure a given message comes from a given source.

Properties of a digital signature.
Authentication, integrity, accountability. Accountability (non-repudiation) is the one a MAC cannot give: with a MAC, the verifier shares the key and could have produced the tag; with a signature only the holder of the private key could have.

Requirements.
Unforgeable, authentic, unalterable, not reusable.

The algorithm pair, as drawn on the slides:

- `tag = Sign_Ks(msg)` with the **signing key Ks, kept private**.
- `isValid = Verify_Kv(msg, tag)` with the **verification key Kv, made public**. Output is valid or invalid.

Protocol: Alice publishes Kv ("Hello, World. This is my public key."). She computes the tag over the message with Ks, sends `(tag, msg)`, and Bob runs Verify with her Kv. Note the direction compared with encryption: the *sender* uses the private key, and anyone with the public key can check.

1.2 RSA signatures and why raw RSA is forgeable

RSA FOR DIGITAL SIGNATURES. sign: `sig = m^s mod N` using the private key `(N, s)`; verify: check `m == sig^v mod N` using the public key `(N, v)`.

In-class worked example (Nov 2 sheet, solutions posted): private exponent s = 3, public exponent v = 7, N = 33, message m = 2.

- Signature: $2^3 \bmod 33 = \mathbf{8}$.
- Verification: $8^7 \bmod 33$. $8^2 = 64 \equiv 31 \equiv -2$; $8^4 \equiv 4$; $8^7 = 8^4 \cdot 8^2 \cdot 8 \equiv 4 \cdot (-2) \cdot 8 = -64 \equiv \mathbf{2}$. Equals m, so the signature is valid.
- Does this give confidentiality? **No.** The message travels in plaintext; signing does not encrypt it.

RSA DOES NOT PROVIDE UNFORGEABILITY. The slide shows the attack: pick any `sig := Rand(N)` and compute `msg := sig^v mod N`. Now `(msg, sig)` verifies, even though nobody signed anything. The forged message is gibberish, but the requirement "unforgeable" is violated. This is called an existential forgery.

The fix: hash, then sign. Alice sends `tag := Sign_Ks(h(msg))` and Bob checks `Verify_Kv(h(msg), tag)`. The random-signature trick now yields a hash value the attacker would have to invert (pre-image resistance) to find a message. Hashing also makes signing fast and lets you sign a message longer than N.

1.3 Encrypt, then sign

Two key pairs are in play. Alice has signing/verification keys (Ks private, Kv public). Bob has encryption/decryption keys (Ke public, Kd private). Both publish their public keys first.

1. `c := Enc_Ke(msg)` — Alice encrypts with **Bob's public key**.
2. `tag := Sign_Ks(c)` — Alice signs the ciphertext with **Alice's private key**.
3. Send `(c, tag)`.
4. `isValid := Verify_Kv(c, tag)` — Bob verifies with **Alice's public key**.
5. `msg := Dec_Kd(c)` — Bob decrypts with **Bob's private key**.

EXAM TIP. The "Old WS Q3" revisited in L28 asked: if Mallory swaps the ciphertext for a different one but keeps Alice's original tag, does the check pass? **No.** The signature is bound to the original ciphertext; a different c fails Verify, and Mallory cannot make a new valid tag without Ks. Be ready to list which of the four keys is used at each of the four steps.

A NOTE OF CAUTION. Beware of determinism. Use unique keys for encryption, signing, and generating MACs. Use appropriate algorithms for the desired property. Do not roll your own.

1.4 "Before we move on... what about quantum?"

Added because of a Quiz 4 question. The points on the slides:

- A **quantum computer** uses **qubits**, which can be in a superposition of 0 and 1; measuring a qubit gives 0 or 1. Quantum algorithms use **interference** to make useful outcomes more likely.
- Why we care: **Shor's algorithm** could break RSA and Diffie-Hellman on a sufficiently powerful quantum computer. That affects **key establishment and digital signatures**. Example: recorded encrypted traffic could be decrypted later ("harvest now, decrypt later").
- Recall RSA security: given N, e, msg^e, find msg; if the attacker can factor N they win. Recall DH security: given g, p, A, B, find K; if the attacker can compute the discrete log of A or B they win.
- Recall one-way functions: easy to compute, hard to invert. Multiplying two large primes is one-way (factoring is hard); modular exponentiation is one-way (discrete log is hard).

- "Hard to reverse" **depends on the computing model**. Shor's algorithm efficiently factors integers and solves discrete logarithms on a quantum computer, undermining both hardness assumptions. It requires a sufficiently large, reliable quantum computer (which does not exist yet).

POST-QUANTUM REPLACEMENTS. ML-KEM: establish a shared secret (NIST FIPS 203). ML-DSA: create digital signatures (NIST FIPS 204). SLH-DSA: another signature option (NIST FIPS 205).

Extra context beyond the slides: ML-KEM is the standardized form of Kyber (lattice-based), ML-DSA of Dilithium (lattice-based), SLH-DSA of SPHINCS+ (hash-based). The slide title says "Schorr's Algorithm" but means Shor's algorithm (Peter Shor, 1994).

1.5 Check yourself

1. Give the slide definition of a digital signature, its three properties, and its four requirements.

A means to ensure a given message comes from a given source. Properties: authentication, integrity, accountability. Requirements: unforgeable, authentic, unalterable, not reusable.

2. Which key signs and which key verifies? Which is public?

Ks (signing key) signs and is kept private; Kv (verification key) verifies and is made public. `tag = Sign_Ks(msg)`, `isValid = Verify_Kv(msg, tag)`.

3. Which cryptographic tool gives non-repudiation, and why can a MAC not?

The digital signature. A MAC key is shared by sender and receiver, so the receiver could have produced the tag; a signature can only be produced by the holder of the private signing key.

4. With s = 3, v = 7, N = 33, sign m = 2 and verify. Does the recipient get confidentiality?

sig = $2^3 \bmod 33 = 8$. $8^7 \bmod 33 = 2 = m$, valid. No confidentiality: the message is sent in the clear.

5. Show that textbook RSA signatures are forgeable and name the fix.

Choose a random sig, compute msg = sig^v mod N; the pair verifies although nothing was signed. Fix: sign the hash, `Sign_Ks(h(msg))`, so the attacker would need to invert the hash to find a message for a chosen signature.

6. In encrypt-then-sign, list the key used at each step. If Mallory replaces c with c' but keeps the tag, what happens?

Encrypt with Bob's public key; sign with Alice's private key; verify with Alice's public key; decrypt with Bob's private key. The verification fails: the signature is bound to c, and Mallory cannot forge a tag for c' without Alice's private key.

7. What does Shor's algorithm do, which two systems does it undermine, and what replaces them?

It efficiently factors integers and solves discrete logarithms on a quantum computer, breaking RSA and Diffie-Hellman (key establishment and signatures). Replacements: ML-KEM (shared secret, FIPS 203), ML-DSA (signatures, FIPS 204), SLH-DSA (signatures, FIPS 205).

8. Why does the quantum threat matter for traffic encrypted today?

Recorded encrypted traffic could be decrypted later once a large enough quantum computer exists, so long-lived secrets need post-quantum key establishment now.

L28 · L29 2. Software security: integer overflow and buffer overflow attacks

The second half of L28 and all of L29 (Nov 6) cover the classic memory-corruption vulnerabilities. Guiding question: *how can a memory error let an attacker take control of a program, and what can we do to prevent or limit the damage?*

THE STACK WE ARE TRUSTING. Every software security mechanism rests on assumptions about the layers below it: Applications → Operating System → ISA / Processor → Hardware.

2.1 Integer overflow

An unsigned 8-bit integer holding 1111 1111 = 255, plus 1, needs a ninth bit. The carry is lost and the result is **0**, not 256. The same wraparound happens at every width, and in the classroom model for signed integers too.

The login counter. `attempts` is a 4-bit signed integer (range −8 to 7); access is granted only while `attempts <= 6`.

```
1. int handle_login(userid, passwd) {
2.   attempts = attempts + 1;        // 4-bit signed int
3.   if (attempts <= 6) {            // max attempts allowed is 6
4.     if passwdValid(userid, passwd) {
5.       attempts = 0;
6.       return (1);                 // access granted
       }
     }
7.   return (0);                     // what happens on the next attempt?
   }
```

Trace (posted solution): attempt 6 is checked; attempt 7 makes `attempts` = 7, which fails `<= 6`, so it is blocked; attempt 8 increments 7 to **−8** (wraparound), `-8 <= 6` is true again, and the password is checked. The lockout is defeated: an attacker just keeps guessing. Fixed version from the slides: check first, then increment.

```
1. int handle_login(userid, passwd) {
2.   if (attempts >= 6) {
3.     return (0);
4.   }
5.   attempts = attempts + 1;
6.   if passwdValid(userid, passwd) {
7.     attempts = 0;
8.     return (1);
     }
9.   return (0);
   }
```

If locked out, reject *without* changing the counter. The practice sheet also asked what the lockout protects against: an **online** guessing attack against the live service. It does nothing against offline guessing on a stolen password database. (Real C signed overflow is undefined behaviour; wraparound is the classroom model.)

Integer overflow as a stepping stone to a buffer overflow.

```
1. int *vulnerable(int *inparray, int len) {
2.   int *myarray;
3.   myarray = malloc(len * sizeof(int));
4.   if (myarray == NULL) {
5.     return (-1);
     }
6.   for (i = 0; i < len; i++) {
7.     myarray[i] = inparray[i];
     }
8.   return myarray;
9. }
```

- Line 3 asks for `len × 4` bytes. For len = 5 that is 20 bytes, fine. If `len` is close to MAX_INT, multiplying by 4 overflows and malloc allocates a **small** number of bytes.
- Lines 6–7 still copy `len` values. The space actually allocated is tiny, so memory past the end of `myarray` is overwritten: **integer overflow used as a stepping stone for a buffer overflow attack**.
- Reflections from the slide: catching this by manual review is very hard; on first read it looks like it does the right checks (it even tests for NULL).
- **Integer underflow** also causes problems: a negative value converted to an unsigned malloc parameter is interpreted as a very large unsigned int. Attempting to allocate huge blocks can trigger out-of-memory conditions or return NULL pointers.

EXAM TIP. Extra Practice 5 Q1 uses the same bug with 8-byte pointers and `count = 2^29 + 1`: the product $2^{32} + 8$ wraps modulo 2^{32} to just **8 bytes** (room for one pointer) while 4,294,967,304 bytes were needed; the loop still writes $2^{29} + 1$ pointers. The fix is a check *before* the multiplication: reject `count > SIZE_MAX / sizeof(*response)` (here $\lfloor (2^{32} - 1)/8 \rfloor$), plus a practical limit and a check for allocation failure.

2.2 Terminology: error, fault, failure, flaw; overflow, overrun, attack, smashing

Error.
A human mistake. Ex: a designer fails to consider a possible input value. (These three come from software engineering.)

Fault.
An encoding of the error. Ex: the program has no code for handling unexpected inputs.

Failure.
Deviation from desired behaviour. Ex: the program crashes when it receives certain inputs. Existence of a failure implies existence of a fault (not the other way around).

Flaw.
Fault or failure. Umbrella term for instances in a system susceptible to exploit by a malicious actor, even though under normal operating conditions a fault would never exhibit as a failure.

The ten-element example: `char sample[10]` filled with ten A's, then `sample[10] = 'B'`. The B lands one past the end, in whatever lives at the next address. On the memory picture the array grows toward higher addresses, so the B sits just above the last A.

Buffer overflow.
Writing past the end of an allocated array.

Buffer overrun.
Reading past the end of an allocated array. (Heartbleed, section 3.4.)

Buffer overflow attack.
Exploiting a buffer overflow flaw to gain control of the system.

Stack smashing.
A particular type of buffer overflow attack (the one that overwrites a stack frame's return address).

2.3 Process memory and the stack frame

Each process has its own code and data, isolated from other users' processes and from the operating system. Inside a process, from high addresses (0xFF…F) down to low (0x00…0): **Stack** (grows downward ↓), free space, **Heap** (grows upward ↑), Local Data, Text (local code), System Data, System Code.

When procedure p1 calls p2 the stack grows downward through these pieces, which the worksheet labels A–E:

```
higher addresses
┌──────────────────────────────────────┐
│ A  p1 local vars          │ p1's stack frame
│ B  p2 parameter(s)        │
├──────────────────────────────────────┤
│ C  return instruction     │ where p2 returns to: the
│    pointer (RIP)          │ instruction after "call p2" in p1
│ D  saved base pointer     │ p2's stack frame
│ E  p2 local vars, incl.   │
│    buffer[100]            │ strcpy writes upward from here
└──────────────────────────────────────┘
lower addresses
```

- **F** on the worksheet is procedure p1's code; **G** is the instruction right after `call p2;`. Without an attack, the address stored at C points to **G**.
- A has a **higher** address than E: the stack grows toward lower addresses, from A toward E.
- A buffer is filled from its low address upward, so a long input marches from E through D and into C.

2.4 The attack: strcpy, a corrupted return address, shellcode

```
(1) int p2(char *input){
(2)   char buffer[100];
(3)   strcpy(buffer, input);
(4)   return 0;
    }
```

`strcpy` copies until it sees a terminating NUL and never checks the destination size. Input `0xCAFECAFECAFECAFE…EEEEEE` overwrites the buffer, the saved base pointer, and then the return pointer with `0xEEEEEEEE`. When p2 returns, execution jumps to 0xEEEEEEEE, which the attacker chose to point back into the buffer, where the rest of the input is machine code. On the worksheet diagram, C has been overwritten with `0x00FFEE`, which points into the injected bytes at E (`0x90 0x90 0x90` is a NOP sled; `0x80 0xcd 0xe1 0x89 …` is the start of the shellcode).

SHELLCODE.C. The injected code typically just launches a shell: `name[0] = "/bin/sh"; name[1] = NULL; execve(name[0], name, NULL);`. If the vulnerable program runs as root (a Set-UID program, as in Lab 4), the shell runs as root.

Worksheet (Nov 6) answers, posted: 1.1 after the overwrite p2 returns to the attacker-chosen address at C (0x00FFEE), pointing into the injected bytes near E. 1.2 a canary goes between the buffer and the saved control information, i.e. between C and D in the class model. 1.3 without an attack C points to G. 1.4 A has the higher address. 1.5 the vulnerable code is in **p2** (the `strcpy`). 1.6 the injected executable is in p2's buffer near E.

2.5 Countermeasures and the attacks that answer them

CANARY. A secret value added to the stack. Before returning from a procedure, check that the canary has the expected value. It sits between the buffer and the return pointer, so a contiguous overwrite that reaches the return address must change the canary first.

W⊕X. Allow regions of memory to be writable or executable, but not both. E.g., the stack must be writable, but does not need to be executable. (Also called NX, DEP, non-executable stack.)

Attack: return into libc. Instead of writing attack code in the buffer, point the return address at existing code in the system, e.g. the C library's `system()`. Nothing on the stack has to execute, so W⊕X does not stop it.

Attack: return-oriented programming (ROP). Point to **gadgets** within the existing code.

GADGET. A short sequence of instructions already present in executable memory, typically ending in `ret`, that an attacker chains with other gadgets to perform an operation. The overflow writes a list of gadget addresses onto the stack (`sub r3, r2; ret` → `add r1, r2; ret` → …); each `ret` pops the next one.

ADDRESS SPACE LAYOUT RANDOMIZATION (ASLR). Change the layout of objects in memory. E.g., the base addresses of the stack, heap, and libraries are chosen at process start-up, so the attacker cannot predict where the buffer, libc or gadgets live. (Lab 4 turned it off with `sysctl -w kernel.randomize_va_space=0`.)

"We have trained the attacker well!": each defense pushed attackers to a cleverer technique. Extra Practice 5 Q3.3 summarizes the four defenses: bounds checking prevents the write; a canary detects many contiguous overwrites before the return; a non-executable stack blocks injected stack bytes but not code reuse; ASLR makes useful addresses hard to predict. **The last three do not fix the unchecked write.**

WATCH OUT. Overwriting C (the return address) changes where execution goes. Overwriting D (the saved base pointer) corrupts frame-management state and may crash or be exploited further, but D is not the saved return instruction pointer. Keep C and D straight.

2.6 Backdoors and privilege escalation

BACKDOOR. A point of entry that bypasses existing access control measures. Added for testing/maintenance purposes → grants complete access to the system.

```
if (userid = "admin" && passwd = "t3st1ng")
    allow_access();
else
    check_credentials(userid, passwd);
```

The password is hardcoded into the program → it will exist in the binary. Anyone who runs `strings` on it finds it. (The demo in class pulled the password out of a compiled binary.)

PRIVILEGE ESCALATION. Change the execution environment to increase privileges: escaping fixed functionality of a program; escaping a sandbox to access the full filesystem; gaining root-level privileges; gaining kernel-mode privileges.

Attacks happen in stages. Ex: first a buffer overflow may be exploited → leads to privilege escalation. **Vulnerabilities we have seen all lead to some form of privilege escalation.** Textbook 6.7 calls the first foothold "base-camp code".

2.7 Check yourself

1. In the 4-bit signed login counter, what happens on attempts 6, 7 and 8, and how do you fix it?

6: incremented to 6, `6 <= 6`, password checked. 7: incremented to 7, blocked. 8: 7 + 1 wraps to −8, `-8 <= 6`, password checked again; lockout defeated. Fix: test `attempts >= 6` first and return 0 without incrementing; otherwise increment, check the password, reset on success.

2. Why is `myarray = malloc(len * sizeof(int))` followed by a copy loop dangerous even though it checks for NULL?

A huge `len` makes the multiplication overflow, so malloc returns a small block, but the loop still copies `len` ints and writes past the end: integer overflow as a stepping stone to a buffer overflow. Check `len` against `MAX / sizeof(int)` before multiplying.

3. Define error, fault, failure and flaw. Which implies which?

Error: human mistake. Fault: the encoding of the error in the program. Failure: deviation from desired behaviour. Flaw: fault or failure, the umbrella term for exploitable instances. A failure implies a fault; a fault need not produce a failure.

4. Distinguish buffer overflow, buffer overrun, buffer overflow attack and stack smashing.

Overflow: writing past the end of an array. Overrun: reading past the end. Attack: exploiting an overflow flaw to gain control. Stack smashing: the particular attack that overwrites a stack frame's return address.

5. In the worksheet diagram (A local vars, B parameters, C return pointer, D saved base pointer, E buffer): where does a canary go, which label has the highest address, where is the shellcode, and where should C point without an attack?

Canary between the buffer and the saved control data (between D and C in the class model). A has the highest address (stack grows down toward E). Shellcode is in the buffer at E. Without an attack C points to G, the instruction after `call p2` in p1.

6. Define canary, W⊕X and ASLR, and name the attack that defeats W⊕X.

Canary: secret value on the stack checked before return. W⊕X: memory is writable or executable, never both. ASLR: randomize the base addresses of stack, heap and libraries at start-up. Return-into-libc and ROP defeat W⊕X by executing existing code instead of injected code.

7. What is a gadget?

A short instruction sequence already present in executable memory, typically ending in `ret`, chained with other gadgets to perform an operation.

8. What is a backdoor and why is a hardcoded test password "very bad"?

A point of entry that bypasses existing access control, often added for testing or maintenance and granting complete access. A hardcoded password exists in the binary, so anyone with the executable can extract it.

9. List the four forms of privilege escalation on the slide.

Escaping a program's fixed functionality (to a shell); escaping a sandbox to the full filesystem; gaining root privileges; gaining kernel-mode privileges.

L29 · L30 3. Writing secure software and Heartbleed

The end of L29 and the first part of L30 (Nov 9) turn from attacks to guidance. The list on the slide, which you should be able to reproduce:

GUIDELINES FOR WRITING SECURE CODE. Check bounds · Validate and sanitize input · Use safe utilities · Grant least privilege · Sandbox · Use type safe languages.

3.1 Bounds checking, static vs dynamic analysis

BOUNDS CHECKING. Check whether a read or write extends beyond the bounds of an allocated object. Done by manual code review, static analysis, or dynamic analysis.

Static analysis (examine the code without running it)	**Dynamic analysis (observe runs)**
+ No runtime overhead	− Runtime overhead
+ Analysis of the full program	− Analysis of individual runs only
+ Bugs are caught early	− Bugs are caught during execution
− False positives	+ No false positives

3.2 Input validation and sanitization

Never trust user provided input. E.g., validate buffer access against the buffer size; use a template to accept user input.

TEMPLATE. Define acceptable input strings. E.g., a phone number: `(919) 555-1234`, `1-919-555-1234`, `+1 919 555 1234`, `919 555 1234`, `919.555.1234`. Anything not matching is rejected: **prescribe good behaviour (default deny)**, instead of trying to enumerate bad inputs.

3.3 Use safe utilities

Use bounded string manipulation functions from the standard C library, libc: `strcpy` vs `strncpy`, `strcmp` vs `strncmp`, `sprintf` vs `snprintf`.

```
char str1[] = "hello";          // h e l l o \0  (6 bytes)
char str2[5];
strcpy (str2, str1);            // copies to the terminating null: 6 bytes into 5 ->
overflow
strncpy(str2, str1, 5);         // copies up to 5 chars: h e l l o, no overflow
```

Extra context beyond the slides: `strncpy` with n = 5 here leaves `str2` without a terminating NUL, which is its own classic pitfall; the slide's point is only that the bounded version cannot write past 5 bytes.

3.4 Least privilege and sandboxing

Grant least privilege (Principle of Least Privilege). A subject should be granted only the privileges necessary to complete its work. E.g., your browser does not need admin privileges. It **limits the damage that occurs when security is compromised**.

- Give code only the access required to complete the work: a calculator does not need access to the filesystem.
- Build modular code: separate HTML rendering from filesystem access in your browser, so a bug in the renderer cannot read your files.

Sandboxing. Allow code access to only a portion of the filesystem. E.g., the browser can access only files in `/tmp`.

3.5 Type safety and memory safety

Type safety. Memory is only ever accessed according to its type. E.g., memory allocated as an int is only ever used as an int; indexing into an array can only ever access the memory allocated as part of the array.

Memory safety. A program is memory safe if it never accesses undefined memory. Violations: buffer overflows, null pointer dereference, use-after-free, use of uninitialized memory, illegal free. **Type safe languages produce memory safe programs.**

3.6 Heartbleed: a buffer overrun

Heartbleed is the famous **overrun** (read past the end), as opposed to an overflow (write).

- **TLS (Transport Layer Security)**: a cryptographic protocol that secures communication over a network. **OpenSSL** is the library implementing TLS.
- TLS heartbeat protocol: the client sends a request with a payload and a claimed length ("3 bytes, `cat`"); the server echoes it back ("`cat`").
- Attack (xkcd 1354): request "64 kbytes, `cat`". The server trusts the claimed length and replies with `cat` followed by whatever 64 KB happened to be next in its memory: "`cat0xcafep2ssw0rd!`", i.e. passwords, keys, other users' data.

- History: bug introduced Dec 2011; RFC published Feb 2012; included and enabled by default in OpenSSL March 2012; first discovered **March 2014**. Two years in the wild.

EXAM TIP. Nov 4 practice Q2, multiple choice: "The lesson from the Heartbleed vulnerability is…" Correct: **the server failed to check the length of the user-provided string**. Not "the server should never echo back a user-provided string" (echoing is fine when bounded), not "failed to sanitize" (the content was harmless; the length was the problem), not "stale heartbeat". Extra Practice 5 Q2 adds: it is an out-of-bounds **read** from the request buffer; the direct fix is checking the claimed length against the available payload; a stack canary or non-executable memory would not prevent it, since nothing is overwritten or executed.

3.7 Check yourself

1. List the six guidelines for writing secure code.

Check bounds; validate and sanitize input; use safe utilities; grant least privilege; sandbox; use type safe languages.

2. Give two advantages of static analysis and the one disadvantage; then the reverse for dynamic analysis.

Static: no runtime overhead, analyzes the full program, catches bugs early; but false positives.
Dynamic: no false positives; but runtime overhead, analyzes individual runs only, catches bugs during execution.

3. What is a template, and which design principle does it embody?

A definition of the acceptable input strings (e.g. the allowed phone-number formats). It prescribes good behaviour: default deny.

4. Why is `strcpy(str2, str1)` with a 5-byte `str2` and `"hello"` a bug, and what is the safe call?

`"hello"` is six bytes including the terminating NUL; strcpy copies all six into five bytes.
`strncpy(str2, str1, 5)` copies at most five characters.

5. State the principle of least privilege and what it buys you.

A subject should be granted only the privileges necessary to complete its work. It limits the damage when security is compromised.

6. Define type safety and memory safety. List five memory-safety violations. How are the two related?

Type safety: memory is only accessed according to its type. Memory safety: the program never accesses undefined memory. Violations: buffer overflow, null pointer dereference, use-after-free, uninitialized memory, illegal free. Type safe languages produce memory safe programs.

7. Is Heartbleed an overflow or an overrun? What was the bug, what was leaked, and what is the fix?

An overrun (out-of-bounds read). The server trusted the claimed heartbeat length instead of checking it against the bytes actually received, so it echoed adjacent memory (keys, passwords). Fix: check the claimed length against the available payload before copying.

8. For how long was Heartbleed in OpenSSL before discovery?

Roughly two years: enabled by default in March 2012, discovered March 2014.

L30 · L31

4. Network security concepts and TCP/IP background

The second half of L30 (Nov 9) opens the network unit. Guiding question: *how do we secure our communication and data over a network?* L31 re-taught the TCP/IP background before the packet-filter examples.

4.1 Why networks are hard to secure

Challenges. Anonymity · Many points of attack · Sharing · Complexity · Unknown perimeter · Unknown path. (From Pfleeger, *Security in Computing*.)

Security perimeter. The boundary that separates the trusted part of the network from untrusted areas.

Nov 9 practice Q2.1 asked how the perimeter can be broken. Posted answer: an infected laptop or USB drive brought inside; phishing; stolen VPN credentials; a malicious insider; an unauthorized wireless connection; or exploitation of a vulnerable service that the firewall allows.

Medium	Strengths	Weaknesses
Wire	Cheap, ubiquitous	Signal emanation, physical wiretapping
Optical fiber	No emanation, no wiretapping	Weak at connection points
Microwave	Strong signal	Interception possible, line of sight needed, needs repeaters
Wireless	Ubiquitous	Interception possible, short range
Satellite	Strong signal	Delay (long distance), interception possible

4.2 The four threats and Dolev-Yao

Threat	Property attacked
Interception (eavesdropping)	Confidentiality
Modification	Integrity
Fabrication (forging messages)	Integrity
Interruption	Availability

DOLEV-YAO MODEL. (IEEE Transactions on Information Theory, 1983.) An **active attacker** who is a legitimate user of the network, can obtain any message on the network, and can be a receiver to any user. In short: **the attacker carries the message.** Every protocol we study is analyzed against this adversary.

4.3 Defense strategies

Three pictures, each a LAN talking to a LAN across the Internet: a **firewall** at the edge of each LAN, an **intrusion detection system** watching the traffic, and a **tunnel** (VPN) wrapping the traffic between the two LANs. Sections 5, 7 and 8 take them in turn.

FIREWALL. A gateway between two networks that can allow, deny, or modify data passing from one to the other. Named after the wall between buildings that stops a fire spreading. Typical placement: Internet → firewall → web server (perimeter / DMZ) → second firewall → database server and the internal network.

FIREWALL PROPERTIES. Security perimeter · Defense in depth (used in conjunction with other methods) · Chokepoint for security analysis (all communication funnels through) · Reference monitor: complete mediation, immune to attack, proven correct.

Types of firewalls: **packet filters** (decide per packet from header fields; section 5) and **proxies** (terminate the connection and speak the application protocol on the client's behalf).

Nov 9 practice Q2.2, why is a firewall alone insufficient? Posted answer: a firewall enforces traffic rules, but **permitted traffic is not necessarily safe**. Attacks can use allowed services, begin inside, or bypass the boundary. Add layers: patching, strong authentication, least privilege, endpoint protection, segmentation, monitoring. A firewall does not fix vulnerable software or prevent all social engineering.

4.4 TCP/IP background

The network protocol stack is layered: Application, Transport (TCP), Network (IP), Data Link, Physical. The bottom layer dictates how bits are transmitted over a wire; the top layer dictates interfaces to applications; between them, protocols dictate how information is routed from one location on the Internet to another.

Layer	Identifier	Unit
Link	MAC address	Frame
Network (IP)	IP address	Packet
Transport (TCP/UDP)	Port number	Segment

IPv4 packet header (20 bytes, options 0–40 bytes, data up to 65,536 bytes): Version, IHL, TOS, Total length; Identification, Flags, Fragment offset; **TTL** (Time To Live: hops remaining, decremented by each router; the instructor circled it because of the TTL evasion trick in section 5.5), Protocol, Header checksum; **Source address**; **Destination address**; Options; Data.

TCP segment header (20–60 bytes): **Source port** (16 bits), **Destination port** (16 bits); **Sequence number** (32); **Acknowledgement number** (32); HLEN, reserved, flags **URG ACK PSH RST SYN FIN**, window size; checksum, urgent pointer; options. Packet filters look at ports and flags; encapsulation puts the TCP segment inside the IP packet's data field.

Three-way handshake. Each SYN consumes one sequence number; a pure ACK consumes none. The ACK number is the *next* sequence number the sender expects.

Message	Flags	Sequence number	ACK number
A → B	SYN	x (e.g. 4335)	—
B → A	SYN + ACK	y (e.g. 1003)	x + 1 (4336)
A → B	ACK	x + 1	y + 1 (**1004**)

Nov 9 practice Q1: with Device A's SYN = 4335 and Device B's SYN = 1003 / ACK = 4336, the third message carries **ACK = 1004**. "The ACK flag is never set in the first message initiating a TCP handshake": **True** for a normal handshake (first message has SYN = 1, ACK = 0; the second is SYN-ACK; the third has ACK set). Arbitrary attacker-crafted packets need not follow the pattern. The figure's "SYN=4335" labels are sequence numbers, not the one-bit SYN flag.

EXAM TIP. Extra Practice 5 Q4.2 is the same exercise with client ISN 100 and server ISN 700: server → client SYN+ACK, seq 700, ack 101; client → server ACK, seq 101, ack 701. Q4.3: the third message acknowledges the server's SYN and initial sequence number. Completing the handshake does **not** prove the server's real-world identity; only cryptographic authentication (TLS, section 8) does.

4.5 Check yourself

1. List the six network security challenges and define the security perimeter.

Anonymity, many points of attack, sharing, complexity, unknown perimeter, unknown path.
Security perimeter: the boundary separating the trusted part of the network from untrusted areas.

2. Map interception, modification, fabrication and interruption to C, I, A.

Interception → confidentiality; modification and fabrication → integrity; interruption → availability.

3. Describe the Dolev-Yao attacker in three clauses.

An active attacker who is a legitimate user of the network, can obtain any message on the network, and can be a receiver to any user. The attacker carries the message.

4. Define a firewall and list its four properties. Which three sub-properties make it a reference monitor?

A gateway between two networks that can allow, deny or modify data passing between them. Properties: security perimeter; defense in depth; chokepoint for analysis; reference monitor, i.e. complete mediation, immune to attack, proven correct.

5. Which medium has no emanation and no wiretapping, and where is it weak?

Optical fiber; weak at connection points.

6. Match MAC address, IP address and TCP port to link, network and transport.

MAC: link. IP: network. Port: transport.

7. Device A sends SYN with sequence 4335; B answers SYN 1003, ACK 4336. What are the flags, sequence and ACK of the third message?

ACK flag only, sequence 4336, ACK 1004 (B's SYN consumed one sequence number).

8. Does a completed handshake authenticate the server? Give a reason.

No. It only confirms both sides received each other's sequence numbers; anyone who can answer the SYN can complete it. Identity needs TLS certificates and a handshake signature.

9. Name three ways a correctly configured firewall can still be bypassed.

Any three of: infected laptop or USB brought inside, phishing, stolen VPN credentials, malicious insider, rogue wireless access point, exploiting a vulnerable service the firewall allows.

L30 · L31 5. Packet filters and signature-based filters

L31 (Nov 11). Guiding question: *how do we encode rules about what traffic we do and don't want to allow into a network?* This section is the most "worked example" part of the quiz material; practice filling in the tables until it is automatic.

5.1 Definitions and categories

PACKET FILTER. IP packets are compared to a set of rules and either allowed, dropped, or rejected. (Dropped = silently discarded; rejected = discarded with an error sent back.)

Filtering rules can look at.
TCP/IP header fields; IP header protocol; packet size, flags; application protocol; user ID; network activity.

Positive vs negative.
Default deny (allow only what is listed) vs default allow (block only what is listed). The example rule sets end with "Either Any Any Any Any Deny": default deny.

Stateless vs stateful.
Decide each packet on its own vs remember connections (handshake state, addresses, ports) and check whether the packet fits a permitted flow.

Shallow view vs deep view.
Headers only vs inspecting the payload (deep packet inspection, section 5.5).

FIREWALL LIMITATIONS. Reliance on topology · Untrustworthy insiders · Tunneling · Encrypted content.

5.2 The mail example, version 1: destination port only

Goal: people inside the organization can receive mail (rules A and B) and send mail (rules C and D); everything else is denied (E). SMTP mail servers listen on TCP port 25; clients use an ephemeral port above 1023. **First matching rule wins.**

Rule	Dir	Src addr	Dest addr	Protocol	Dest port	Action
A	In	External	Internal	TCP	25	Allow
B	Out	Internal	External	TCP	>1023	Allow
C	Out	Internal	External	TCP	25	Allow
D	In	External	Internal	TCP	>1023	Allow
E	Either	Any	Any	Any	Any	Deny

Incoming mail (external client 121.12.1.2 port 5150 → internal mail server 222.22.2.2 port 25): packet 1, In, dest port 25 → **Allow (A)**; the reply, Out, dest port 5150 → **Allow (B)**. Outgoing mail (internal client port 5150 → external server port 25): packet 3, Out, dest port 25 → **Allow (C)**; reply In, dest port 5150 → **Allow (D)**.

A HIGH DESTINATION PORT DOES NOT PROVE A PACKET IS A REPLY. Rule D lets an attacker connect from outside to any internal service on a port above 1023, e.g. a web proxy on **8080**: packet In, dest 8080 → Allow (D); and rule B lets the server reply (Out, dest 5150 → Allow (B)). The attacker has a full connection to an internal service.

5.3 Version 2: add the source port

Rule	Dir	Src addr	Dest addr	Prot	Src port	Dest port	Action
A	In	External	Internal	TCP	>1023	25	Allow
B	Out	Internal	External	TCP	25	>1023	Allow
C	Out	Internal	External	TCP	>1023	25	Allow
D	In	External	Internal	TCP	25	>1023	Allow
E	Either	Any	Any	Any	Any	Any	Deny

Now the proxy attack (In, src 5150, dest 8080) matches nothing but E → **Deny**, and so does its reply.

SOURCE PORT 25 DOES NOT PROVE A PACKET IS A MAIL SERVER'S REPLY. The attacker simply *chooses* source port 25: In, src 25, dest 8080 → Allow (D); the reply Out, src 8080, dest 25 → Allow (C). Anyone can set any source port on their own machine.

5.4 Version 3: add the ACK flag

In TCP, only the very first packet of a connection (the SYN) has ACK clear; every later packet has ACK set. So a rule that is meant to match *replies* can demand ACK = Yes, and an attacker's connection-opening SYN (ACK = No) will not match it.

Rule	Dir	Src addr	Dest addr	Prot	Src port	Dest port	ACK set	Action
A	In	External	Internal	TCP	>1023	25	Any	Allow
B	Out	Internal	External	TCP	25	>1023	Yes	Allow
C	Out	Internal	External	TCP	>1023	25	Any	Allow
D	In	External	Internal	TCP	25	>1023	Yes	Allow
E	Either	Any	Any	Any	Any	Any	Any	Deny

The fake-reply attack (In, src 25, dest 8080, ACK **No**) now falls through to **Deny (E)**. Rules A and C say "Any" because the connection-opening SYN legitimately has no ACK.

Nov 11 practice sheet, Q1 (posted solutions):

Packet	Dir	Src	Dest	Prot	Src port	Dest port	ACK	Action (rule)
1	In	121.12.1.2 (ext)	222.2.2.2 (int)	TCP	25	8080	Yes	**Allow (D)**: external source port 25, internal dest port >1023, ACK set.
2	In	121.12.1.2	222.2.2.2	TCP	5051	25	Yes	**Allow (A)**: source port >1023, dest port 25; A accepts either ACK value.
3	In	121.12.1.2	222.2.2.2	TCP	25	5055	Yes	Allow (D). Reasonable assumption: **someone inside is sending mail to someone outside**: this looks like the external mail server (port 25) answering the internal client (port 5055), most likely the server's SYN-ACK.

ACK SET STILL PROVES VERY LITTLE. Extra Practice 5 Q8: an incoming packet with src 25, dest 5055, ACK set passes rule D and *could* be the SYN-ACK of an outgoing mail connection, but ACK alone does not tell you SYN is also set; it could be a later packet or a fabricated one. ACK set does not prove an internal client opened a connection, because an attacker can set the flag and choose the source port. A **stateful** firewall can additionally track addresses, ports, protocol and handshake state and check that the packet fits a permitted flow. That is stronger filtering, not cryptographic authentication.

5.5 Signature-based filters and how to evade them

"Problem: packet filters have an imperfect view." Header fields say nothing about what the payload is doing.

SIGNATURE-BASED FILTER. A packet filter that inspects the packet payload to find patterns of known attacks. E.g., look for "root"; look for "/etc/passwd".

Three evasions, each followed by its fix:

1. **Split across segments.** Segment 1 ends "...ro", segment 2 begins "ot...". Neither segment contains "root" but the reassembled stream does. Inspecting packets individually misses content split across boundaries. Fix: track sequences of packets. Downside: more state to keep.

2. **Send out of order.** The attacker can easily make "ot…" arrive before "…ro", so a filter that only concatenates consecutive packets still misses it. Fix: **reassemble the TCP stream** using sequence numbers, the way the receiving endpoint does.
3. **TTL games.** Send `r`(TTL 25) `i`(TTL 19) `o`(25) `p`(19) `o`(25) `t`(25). The filter, close to the sender, sees "riopot". The segments with the short TTL expire before they reach the destination, which sees "root". Fix: the filter must interpret the stream *consistently with the receiving endpoint* (know which packets will actually arrive).

EXAM TIP. Extra Practice 5 Q8.3 phrases it as: "A content filter checks each TCP segment separately for the text root. One segment ends in ro; the next begins with ot. Why can the filter miss the word? What should it inspect instead?" Answer: neither segment contains the whole word; inspect the reassembled TCP byte stream, interpreted as the receiving endpoint will.

5.6 Writing a rule set yourself (Extra Practice 5 Q6)

Internal network 10.20.30.0/24, web server 10.20.30.10. Policy: permit incoming HTTPS; permit SSH to that server only from 203.0.113.0/24; permit outgoing TCP/UDP DNS queries; deny everything else. Single-packet rules, return traffic omitted. The blanks resolve as: **A dest port 443**; **B protocol TCP**; **C action Deny** (plain HTTP on port 80 is not in the policy); **D dest port 53**.

Rule	Dir	Source	Destination	Protocol	Dest port	Action
A	In	External	10.20.30.10	TCP	443	Allow
B	In	203.0.113.0/24	10.20.30.10	TCP	22	Allow
C	In	External	10.20.30.10	TCP	80	Deny
D	Out	10.20.30.0/24	External	TCP/UDP	53	Allow
E	Either	Any	Any	Any	Any	Deny

Packets: (1) In 198.51.100.9 → .10 TCP 443: Allow/A. (2) In 203.0.113.55 → .10 TCP 22: Allow/B. (3) In 198.51.100.7 → .10 TCP 80: Deny/C. (4) In 198.51.100.7 → .10 TCP 22: Deny/E (wrong source network for B). (5) Out 10.20.30.42 → 198.51.100.53 UDP 53: Allow/D. (6) same with TCP 53: Allow/D. (7) Out 10.20.30.42 → 198.51.100.9 TCP 443: Deny/E (outgoing HTTPS was never permitted). If E were moved to the top, every packet would match E first and be denied. Rule C can be deleted without changing which packets are allowed, since they fall through to E anyway; only the matching rule and perhaps logging would differ.

5.7 Check yourself

1. Define a packet filter and give its three possible actions.

IP packets are compared to a set of rules and either allowed, dropped, or rejected.

2. Name the three ways to categorize packet filters and the four firewall limitations.

Positive vs negative (default deny vs default allow); stateless vs stateful; shallow vs deep view. Limitations: reliance on topology, untrustworthy insiders, tunneling, encrypted content.

3. In the version-1 mail rule set, why is rule D ("In, dest port >1023, Allow") dangerous?

A high destination port does not prove a reply. An outsider can connect to any internal service above 1023 (e.g. a proxy on 8080) under rule D, and rule B lets the service answer.

4. After adding source ports, how does the attacker get through, and what is the final fix?

The attacker sets source port 25 so the connection request matches D and the reply matches C. Fix: require ACK set on the reply rules B and D; the attacker's opening SYN has no ACK and falls to E.

5. Version-3 rule set: incoming TCP, src port 5051, dest port 25, ACK set. Action and rule?

Allow (A): external source port >1023 to internal port 25; A accepts any ACK value.

6. Version-3 rule set: incoming TCP, src port 25, dest port 8080, ACK not set. Action and rule?

Deny (E). D requires ACK set; nothing else matches.

7. Does an incoming packet with source port 25 and ACK set prove an internal client opened a mail connection? What could do better?

No; the attacker can set both. A stateful firewall that tracked the outgoing SYN and checks that the packet fits a permitted flow does better, though it is still not cryptographic authentication.

8. Define a signature-based filter and describe the three evasions with their fixes.

A packet filter that inspects payloads for patterns of known attacks ("root", "/etc/passwd"). Evasions: split the string across segments (fix: track packet sequences), send segments out of order (fix: reassemble the TCP stream), and insert decoy segments with a TTL too small to reach the destination (fix: interpret the stream as the endpoint will).

9. Why are rules written "first match wins" dangerous if the default-deny rule is moved to the top?

Every packet matches the catch-all first and is denied; the allow rules below are never consulted.

WA5 6. Why a source address cannot authenticate a sender

WA5 (Nov 13) was an in-class exercise on an isolated three-container Docker network: a **victim** (10.34.50.2), an **attacker** (10.34.50.3) and a **target** (10.34.50.20). The lab's rules were explicit: use the supplied isolated network only, never any real network. The written part is four short reflection questions, and the point the quiz will test is the *lesson*, not the tooling: a claimed source address is not proof of identity.

6.1 The observation

- The victim has a route that sends traffic for 10.34.50.20 through the attacker container, which has IP forwarding turned off.
- When the victim pings the target, the attacker can **observe the echo requests** going by, but the target's replies **do not come back**, because the attacker does not forward them.
- So the attacker sits on the path and can read the request fields: source, destination, ICMP identifier, sequence number, payload.

6.2 What makes a forged reply "look right"

A ping program decides a reply belongs to its outstanding request by matching a few plaintext fields. An ICMP echo reply that a pinger will accept simply reverses the two addresses (source = the address it was waiting to hear from, destination = the victim), carries ICMP type 0 (echo reply), and copies the request's identifier and sequence number and payload unchanged. None of those fields is secret, and the attacker read all of them off the request it observed.

WA5 REFLECTION ANSWERS. 1.1 The initial capture showed only echo requests and no replies, because the attacker container does not forward the victim's traffic to the real target, so nothing legitimately answers. 1.2 In Wireshark the forged reply is the echo reply whose claimed source is the target's address (10.34.50.20) even though it was produced by the attacker. 1.3 After the ICMP identifier is changed, the ping program no longer associates the reply with its outstanding request, so ping does not count it even though the packet still appears in the capture. 1.4 A source IP address proves nothing about who sent a packet; it is a value the sender writes, so it cannot serve as authentication.

6.3 Why changing one field breaks it

Extra Practice 5 Q5 restates the idea without any lab steps:

- **5.1** The victim's ping can count a forged reply as a response because it matches the fields the program checks (addresses, identifier, sequence), and those fields are observable, not secret.
- **5.2** If only the ICMP identifier is changed, a capture still records the packet (it is on the wire) but ping does not count it, because a mismatched identifier prevents ping from associating the reply with its outstanding request. Capturing a packet and *accepting* it are different things.

- **5.3** A matching source IP and identifier do **not** authenticate the sender. The source IP is a claimed address and the identifier only helps match requests to replies; neither is cryptographic authentication.

EXAM TIP. This ties directly to the network-security theme (section 4) and to TLS (section 8): completing a TCP handshake or echoing the right ICMP fields shows a party can see and copy plaintext values, not that it is who it claims to be. Only a signature over the handshake, verified against a trusted public key, authenticates identity. Expect a question like "does a matching source IP prove who sent the packet?" — the answer is no, with that reason.

6.4 Check yourself

1. Why could the attacker see the victim's echo requests but the victim see no replies at first?

The victim's route sends target-bound traffic through the attacker, which has IP forwarding disabled, so it observes the requests but does not pass them on, and nothing legitimately answers.

2. Which fields does an accepted echo reply reuse from the request, and which does it reverse?

It reverses source and destination addresses (claiming to come from the target) and copies the ICMP identifier, sequence number and payload unchanged, with type set to echo reply.

3. If only the ICMP identifier is changed, why might a capture show the reply while ping ignores it?

The packet is still on the wire so Wireshark records it, but ping matches replies to outstanding requests by identifier, so a changed identifier means ping does not associate or count it.

4. What does this activity show about using a source IP address as authentication?

Nothing authenticates a sender by its claimed source address; it is a value chosen by whoever sends the packet. Identity requires cryptographic authentication, not matching plaintext header fields.

L31 · L33 7. Intrusion detection, tunnels, SSH, TLS and PKI

The IDS material started at the end of L31 and the math was worked in L33 (Nov 16). Guiding question for L33: *how do we protect a connection, and know who is at the other end?*

7.1 Intrusion detection systems

Intrusion.
An event on a host or network that violates the desired security policy (a.k.a. incident). E.g., an unauthorized connection, illegal program execution.

Intrusion detection system (IDS).
An automated process that monitors events and alerts a human to potential harm. A detection is **evidence to investigate, not proof of an intrusion.**

IDS (detection)	IPS (prevention)
Supports forensic analysis; monitors and logs events; alerts a human to adverse events	Responds to events; blocks or alters events; alerts a human to adverse events

NIDS (network-based)	HIDS (host-based)
Network-wide viewpoint; does not touch endpoints; events are packets	Single-host viewpoint; not blocked by encryption; events are log updates, filesystem changes, system calls, CPU use

7.2 The confusion matrix and the five rates

	Intrusion	No intrusion
Alarm	True Positive (TP)	False Positive (FP)
No alarm	False Negative (FN)	True Negative (TN)

The rates (exactly as on the slides). False Positive Rate FPR = FP / (FP + TN).
True Negative Rate TNR = TN / (FP + TN).
False Negative Rate FNR = FN / (TP + FN).
True Positive Rate TPR (detection rate) = TP / (TP + FN).
Alarm Precision AP = TP / (TP + FP) = TP / (all alarms).

Watch the denominators. FPR and TNR are conditioned on the **non-intrusions** (their denominators are FP + TN). TPR and FNR are conditioned on the **intrusions** (denominators TP + FN). Alarm precision is conditioned on the **alarms** (denominator TP + FP). Mixing these up is the easiest way to lose the points.

7.3 The base rate, with the worked examples

WHY THE BASE RATE MATTERS. When attacks are rare, even a small false-positive rate can produce many false alarms. A high detection rate does not mean most alarms are real attacks.

Worked example (Nov 11 practice Q2, revisited L33). TPR = 99%, TNR = 98%, 50,000 events, 2 of them intrusions. Benign events = 50,000 − 2 = 49,998. FPR = 1 − TNR = 0.02.

- Expected true alarms = 2 × 0.99 = 1.98.
- Expected false alarms = 49,998 × 0.02 = 999.96.
- Total ≈ **1001.94 ≈ 1002 alarms.** (Keep the fractions as expected counts; do not round individual events.)

Alarm precision (Nov 11 practice Q3). A separate scenario: 50,000 events, 2 intrusions, the IDS raises 100 alarms and both intrusions are detected. Then TP = 2, FP = 98, and precision = TP / (TP + FP) = 2 / 100 = **2%.** The denominator is alarms, not all events. So 98% of alarms are false even though every intrusion was caught: that is the base-rate fallacy.

Detection has tradeoffs: too many **false negatives** and intrusions go undetected; too many **false positives** and you waste investigation time and cause **alert fatigue**.

EXAM TIP. Extra Practice 5 Q9 drills all of it: Q9.1 with 10 intrusions detected and 50 total alarms gives TP 10, FP 40, FN 0, TN 99,950, precision 10/50 = 20%. Q9.2 with TPR 90% and FPR 1% on the same 100,000 events (10 intrusions) gives expected TP 9, FP 999.9, total ≈ 1008.9, precision ≈ 0.892%. Q9.3: intrusions are rare, so a small FPR over a large benign population produces more false than true alarms; TPR conditions on intrusions, precision conditions on alarms.

7.4 Tunnels, VPNs and SSH

Tunneling.
Carrying traffic inside another connection or packet stream. The tunnel endpoints add and remove the outer wrapper. **A tunnel is not automatically encrypted.**

Virtual Private Network (VPN).
A protected tunnel that connects hosts or networks across a public network; it protects traffic between its endpoints (network-to-network or network-to-host).

Encapsulation.
Putting one protocol's data inside another protocol's payload (e.g. an IP packet carries a TCP segment). A VPN adds an **outer** IP header; the inner header holds the original destination, the outer header holds the receiving tunnel endpoint.

Secure Shell (SSH). An encrypted, authenticated connection for remote access (log in to a remote machine and run commands); it can also forward other applications' traffic. **The client must recognize the server before sending credentials.**

SSH protocol layers, bottom to top: **TCP/IP → SSH Transport Layer → SSH User Authentication → SSH Connection.** The client initiates; the server authenticates with its public key; parameters are negotiated; then the user authenticates.

- **Server authentication** uses public-key cryptography: the SSH host (server) issues a signature; the client verifies it. How does the client get the right host key? A CA, a database of host-to-public-key pairs, or **Trust On First Use (TOFU)**: accept the key on the first connection and warn if it ever changes.
- **User authentication**: client password (static or one-time); a **Kerberos ticket** (a trusted Kerberos server issues tickets granting access to services); or client public key (the server checks the key is on its list of pre-registered keys and the signature is valid for this public key and the current session id).

Exam tip. Extra Practice 5 Q10: inner vs outer destination (original destination vs tunnel endpoint); encapsulation is wrapping, not encryption. In SSH, the server signs with its host private key and the client verifies with the trusted host public key; for user public-key auth the client signs with the user's private key and the server checks it against a pre-registered public key. TOFU detects a *change* of host key later but does not establish on the first connection that the key really belongs to the intended server; an independently verified fingerprint or a trusted CA does.

7.5 HTTP, TLS, certificates and PKI

HTTP.
The request/response protocol: the browser sends "GET /index.html"; the server returns a status code, headers and content. Plain HTTP does not encrypt or authenticate the exchange.

TLS (Transport Layer Security).
A protocol that protects communication between applications, providing confidentiality, integrity, and usually server authentication. **HTTPS = HTTP carried over TLS.**

Encryption alone does not tell Alice *whose* public key she received: a man in the middle can offer his own key and say "I am Bob." That is the problem PKI solves.

Public Key Infrastructure (PKI). The technologies, organizations and rules for managing public keys and binding them to identities. What makes it work: certificates bind identities to public keys; certificate authorities (CAs) validate and sign those bindings; trust stores identify trusted root CAs; issuance, renewal and revocation manage certificates over time.

PUBLIC KEY CERTIFICATE. Contains a **subject**, the subject's **public key**, an **issuer**, and the **issuer's signature**. All the fields are hashed with a cryptographically secure hash function, and the hash is signed with the issuer's (CA's) private key: `SignKs(h(subject, public key, issuer, …))`. The signature attests to the contents.

Certificate chain (the DigiCert teaching example):

```
(1) subject: www.amazon.com      issuer: DigiCert Global CA G2   <- leaf
(2) subject: DigiCert Global CA G2   issuer: DigiCert Global Root G2  <- intermediate
(3) subject: DigiCert Global Root G2 issuer: DigiCert Global Root G2  <- root (self-signed)
```

Each certificate is signed by the next one up; the root signs itself. Signature (1) on the leaf is created with **DigiCert Global CA G2's private key** and verified with its public key, which certificate (2) supplies. The chain supplies the intermediate and leaf public keys; the browser must **already know and trust the root's public key** (from its trust store). The root's self-signature alone does not establish trust: anyone can self-sign.

WHAT MUST THE BROWSER CHECK?. Does the certificate cover the requested hostname? Are the certificates valid for this use and this time? Do the signatures chain to a trusted root? Apply revocation checks according to client policy.

Nov 16 practice (posted): for cs.unc.edu the leaf's common name was actually chem.unc.edu with cs.unc.edu in the Subject Alternative Names (one certificate can cover several hostnames), issued by an intermediate chaining up to ISRG Root X1 (the trust anchor). For the DigiCert figure: the user was browsing **www.amazon.com** (named by the leaf); the browser must already know **DigiCert Global Root G2's public key**; the trust anchor is **DigiCert Global Root G2**, trusted because it is in the trust store, not merely because it signed itself.

7.6 The TLS handshake and its limits

"Put the pieces together": key exchange → shared secrets; certificate + handshake signature → server authentication; finished checks → confirm the handshake; session keys → protect application data.

1. **Phase 1 (key exchange):** Client Hello (nonce, protocol versions, cipher suites, DH key-share) and Server Hello (nonce, chosen parameters, DH key-share).
2. **Phase 2 (server parameters):** extensions, certificate request.
3. **Phase 3 (authentication):** certificate, certificate verify (the handshake signature), finished MAC from each side.
4. Then HTTP flows inside the TLS channel.

A CERTIFICATE IS NOT A SAFETY RATING. Domain validation checks **control of the named domain**, not that the site is honest or bug-free. An attacker's own website can have a perfectly valid certificate. And HTTPS protects data in transit to the authenticated endpoint; it does not fix vulnerable application code and does not make downloaded scripts or page content safe. That is the bridge to XSS (section 8).

EXAM TIP. Extra Practice 5 Q11–Q13 hit this repeatedly. A valid certificate sent by the server is not enough, because certificates are public and copyable: the **handshake signature** proves possession of the matching private key. Application data is protected by **symmetric session keys**; the CA's public key only verifies certificate signatures. Validation failing on hostname coverage means the browser must reject the certificate even if every signature verifies. And a valid certificate for `amazon.money.example` authenticates *that* hostname, not amazon.com; if the attacker controls that domain, TLS faithfully protects Alice's password all the way to the attacker's server, which can then read it.

7.7 Check yourself

1. Define intrusion and IDS. How does an IDS differ from an IPS, and NIDS from HIDS?

Intrusion: an event violating the desired security policy. IDS: monitors events and alerts a human; evidence, not proof. IPS also responds, blocking or altering events. NIDS has a network-wide view of packets and does not touch endpoints; HIDS has a single-host view (logs, filesystem, syscalls, CPU) and is not blocked by encryption.

2. Write the five rates with their denominators, and give the alarm-precision formula.

FPR = FP/(FP+TN); TNR = TN/(FP+TN); FNR = FN/(TP+FN); TPR = TP/(TP+FN); AP = TP/(TP+FP).

3. TPR 99%, TNR 98%, 50,000 events of which 2 are intrusions. How many alarms?

True alarms $2 \times 0.99 = 1.98$; false alarms $49{,}998 \times 0.02 = 999.96$; total ≈ 1002.

4. Same 50,000 events, 2 intrusions, 100 alarms raised, both intrusions detected. Alarm precision?

TP 2, FP 98, precision $2/100 = 2\%$. The denominator is alarms.

5. In one sentence, why can a detector catch most intrusions yet have mostly false alarms?

Intrusions are rare, so even a small false-positive rate applied to the large benign population yields more false alarms than true ones; TPR conditions on intrusions, precision on alarms.

6. Is a tunnel encrypted? What do the inner and outer destination addresses identify in a VPN?

Not automatically; a VPN adds the cryptographic protection. Inner destination = the original packet's destination; outer destination = the receiving tunnel endpoint.

7. In SSH server authentication, who signs and who verifies, and what are the three ways a client can trust the host key?

The server signs with its host private key; the client verifies with the host public key. Trust via a CA, a database of host/key pairs, or Trust On First Use.

8. What four things must a browser check about a certificate chain?

Hostname coverage; validity for this use and time; that signatures chain to a trusted root; revocation per client policy.

9. In the DigiCert chain, whose key signs the leaf, and which key must the browser already trust?

The intermediate (DigiCert Global CA G2) signs the leaf with its private key; the browser must already trust DigiCert Global Root G2's public key.

10. TLS validation succeeds for amazon.money.example, controlled by an attacker. Does HTTPS stop the site from reading Alice's password?

No. TLS protects the data in transit to that endpoint; since the endpoint is the attacker's server, it decrypts and can read the password. A valid certificate proves domain control, not honesty.

L34 8. Browsers, the same-origin policy and cross-site scripting

L34 (Nov 18) introduced the Lab 5 XSS activity. Extra Practice 5 stops at Lecture 33, so this is likely lighter on the quiz, but the definitions are quotable and the "HTTPS does not make the page safe" link from section 7 makes it fair game. The framing: the server sends a page and the browser runs its JavaScript, so if attacker text becomes part of the page, the browser runs it.

8.1 What the browser does, and the DOM

- HTML describes the page; JavaScript can read and change it. A **session cookie** lets the server recognize a logged-in browser.

> **Document Object Model (DOM).** The internal, client-side representation of an HTML document. The tree runs `window → document → (location → href, URL, domain, cookie) → element`. Script reads and writes the page through the DOM (`document.cookie`, `document.getElementById`, …).

8.2 Origins and the same-origin policy

> **Origin.** The scheme + host + port of a page. A script in a page runs with that page's privileges.

> **Same-Origin Policy (SOP).** A web page from one host should not be able to read or modify content from another host. E.g., a script from thenews.com should not have access to content loaded from mybank.com.

How origins are assigned: (1) the base HTML document is assigned an origin from its URL; (2) scripts and images are assigned the origin of the loading document; (3) scripts can access content whose assigned origin matches their own.

> **Origin matching.** Two origins match if and only if the origin triplets (scheme, host, port) match. E.g., `https://accounts.mybank.com:443` matches only another page with the same scheme, host and port.

"A strict isolation policy is easy to define and enforce, but it breaks usability and features. We want a policy that allows interaction but 'safely' — this is hard." That tension is exactly what XSS exploits: injected code runs *inside* the trusted origin, so the SOP treats it as belonging there.

8.3 Cross-site scripting (XSS)

XSS exploits code running inside the trusted origin. The Lab 5 target is Elgg, a social-networking app with its XSS countermeasures deliberately disabled, re-creating the 2005 MySpace "Samy worm".

STORED XSS. The attacker's script is saved on the server (e.g. in a profile field) and delivered to every visitor. When another user views the infected profile, the profile data becomes **code in the visitor's browser**, running with the site's origin and the visitor's session.

What injected script can do, in increasing severity (the Lab 5 checkpoints, stated conceptually):

- Run at all: a stored `<script>alert("XSS")</script>` runs when anyone views the profile, without the viewer typing anything.
- Read page-accessible data: `document.cookie` shows cookies the page can see. **HttpOnly** blocks a script from reading a cookie directly, but does not stop injected code from taking authenticated actions.
- Exfiltrate: setting an image's `src` to an attacker URL with the data in the query string makes the browser send a GET request, no click needed.
- Act as the viewer: the script can read the page's security token and timestamp and issue an authenticated request (add a friend, edit the profile) that the viewer's session authorizes.
- Self-propagate: a worm copies its own source into the viewer's profile, so the next person who views *that* profile is infected too.

WHAT WOULD PREVENT THIS?. Keep untrusted profile content from becoming executable code: **encode** untrusted text for the context where it is displayed; if rich HTML is allowed, **sanitize** it with a vetted HTML sanitizer; use **CSP** (Content Security Policy) as another layer; HttpOnly protects direct cookie access. HTTPS protects transit; the application must still handle untrusted content safely.

EXAM TIP. The clean one-liner the slides keep returning to: HTTPS protects the *connection*, the same-origin policy protects *across* origins, but XSS runs code *inside* the trusted origin, so neither stops it; the fix is at the application, by treating stored user content as data, not code.

8.4 Check yourself

1. Define the DOM.

The internal, client-side representation of an HTML document, through which JavaScript reads and changes the page.

2. What is an origin, and when do two origins match?

Scheme + host + port. They match if and only if all three of scheme, host and port are equal.

3. State the same-origin policy and how a script gets its origin.

A page from one host should not read or modify content from another host. The base document takes its origin from its URL; scripts and images take the origin of the loading document; a script can access only content whose origin matches its own.

4. Define stored XSS and explain why the same-origin policy does not stop it.

The attacker's script is saved on the server and served to every visitor; it runs inside the site's own origin with the viewer's session, so the SOP treats it as belonging there.

5. Does HttpOnly stop every consequence of XSS?

No. It blocks a script from reading the cookie directly, but injected code can still make authenticated requests using the viewer's session.

6. List four defenses against XSS and the one-line relationship between HTTPS, SOP and XSS.

Output-encode untrusted text for its display context; sanitize allowed HTML with a vetted sanitizer; apply CSP; set HttpOnly on cookies. HTTPS protects the connection and SOP protects across origins, but XSS runs inside the trusted origin, so only safe handling of untrusted content at the application prevents it.

9. Practice exam (mixed, all topics)

Aim for 50 minutes without notes, then check. The styles mirror the in-class sheets: short calculations, "which rule / where does it jump", fill-in tables, and "define X and say why".

Part A: definitions and short answer

A1. Give the slide definition of a digital signature, its three properties, and why it provides non-repudiation where a MAC does not. [L28]

> A means to ensure a given message comes from a given source. Properties: authentication, integrity, accountability. A MAC key is shared, so the verifier could have made the tag; only the signer holds the private signing key, so a signature is non-repudiable.

A2. Define error, fault, failure and flaw. Which implies the existence of which? [L29]

> Error = human mistake; fault = its encoding in the program; failure = deviation from desired behaviour; flaw = fault or failure. A failure implies a fault; a fault need not cause a failure.

A3. Define buffer overflow, buffer overrun, buffer overflow attack and stack smashing. Which is Heartbleed? [L29, L30]

> Overflow = writing past an array's end; overrun = reading past it; attack = exploiting an overflow to gain control; stack smashing = the attack that overwrites a stack frame's return address. Heartbleed is an overrun.

A4. Map interception, modification, fabrication, interruption to CIA, and describe the Dolev-Yao attacker. [L30]

> Interception → confidentiality; modification and fabrication → integrity; interruption → availability. Dolev-Yao: an active attacker who is a legitimate network user, can obtain any message, and can be a receiver to any user; the attacker carries the message.

A5. Define a packet filter, name its three actions, and give the four firewall limitations. [L31]

> IP packets are compared to rules and allowed, dropped or rejected. Limitations: reliance on topology, untrustworthy insiders, tunneling, encrypted content.

A6. Write the five IDS rates with denominators and the alarm-precision formula. [L33]

> FPR = FP/(FP+TN); TNR = TN/(FP+TN); FNR = FN/(TP+FN); TPR = TP/(TP+FN); AP = TP/(TP+FP).

A7. Define origin, origin matching, and the same-origin policy. [L34]

Origin = scheme + host + port. Two origins match iff all three match. SOP: a page from one host should not read or modify content from another host.

A8. What does Shor's algorithm break, and name the three NIST post-quantum standards by purpose. [L28]

RSA and Diffie-Hellman (factoring and discrete log). ML-KEM for shared-secret establishment (FIPS 203), ML-DSA for signatures (FIPS 204), SLH-DSA an alternative signature (FIPS 205).

Part B: calculations and tracing

B1. With RSA signing exponent s = 3, verification exponent v = 7, N = 33, sign m = 2 and verify. Is the message confidential? [L28]

sig = $2^3 \bmod 33 = 8$. Verify: $8^7 \bmod 33 = 2 = m$, valid. Not confidential; the message is in plaintext.

B2. Count = $2^{29} + 1$, pointers are 8 bytes, unsigned 32-bit multiply. How many bytes are allocated vs needed, and what check prevents it? [L28]

Needed: $(2^{29}+1) \cdot 8 = 2^{32}+8 = 4{,}294{,}967{,}304$ bytes. The multiply wraps mod 2^{32} to 8 bytes. Check count $\leq \lfloor (2^{32}-1)/8 \rfloor$ (i.e. count ≤ SIZE_MAX/sizeof) before multiplying.

B3. Stack diagram: A = p1 local vars, B = p2 parameters, C = return pointer, D = saved base pointer, E = buffer; F = p1's code, G = the instruction after `call p2`. Answer: (a) where does execution jump after the illustrated overwrite? (b) where does a canary go? (c) where does C point with no attack? (d) higher address, A or E? (e) is the vulnerable code in p1 or p2? (f) where is the injected shellcode? [L29]

(a) to the attacker's address written into C, pointing into the injected bytes near E; (b) between the buffer and the saved control data, i.e. between D and C; (c) G; (d) A; (e) p2; (f) in p2's buffer at E.

B4. Complete the TCP handshake: client ISN 100, server ISN 700, no data. Fill the two missing rows, then say what the third message acknowledges and whether it authenticates the server. [L30]

Server → client: SYN+ACK, seq 700, ack 101. Client → server: ACK, seq 101, ack 701. The third acknowledges the server's SYN and ISN; it does not authenticate the server's identity.

B5. Version-3 mail rule set (source port + ACK). Classify each: (1) In, src 25, dest 8080, ACK not set. (2) In, src 5051, dest 25, ACK set. (3) In, src 25, dest 5055, ACK set. [L31]

(1) Deny (E): rule D needs ACK set. (2) Allow (A): external src >1023 to port 25, any ACK. (3) Allow (D): looks like an external mail server (port 25) answering an internal client, most likely the server's SYN-ACK of outgoing mail.

B6. IDS: TPR 90%, FPR 1%, 100,000 events, 10 of them intrusions. Expected true alarms, false alarms, total, and alarm precision. [L33]

True alarms = $10 \times 0.90 = 9$; false alarms = $99{,}990 \times 0.01 = 999.9$; total ≈ 1008.9; precision $\approx 9/1008.9 \approx 0.892\%$.

B7. Fill the packet-filter blanks: policy permits incoming HTTPS to 10.20.30.10, SSH to it only from 203.0.113.0/24, outgoing DNS; deny else. Give the action for: (a) In 198.51.100.7 → .10 TCP 80; (b) In 198.51.100.7 → .10 TCP 22; (c) Out 10.20.30.42 → external TCP 443. [L31]

(a) Deny (plain HTTP not permitted); (b) Deny (SSH allowed only from 203.0.113.0/24); (c) Deny (outgoing HTTPS was never permitted). Only incoming 443, SSH from the one subnet, and outgoing DNS are allowed.

Part C: explain why

C1. Why is textbook RSA signing forgeable, and what single change fixes it? [L28]

Pick a random sig and compute msg = sig^v mod N; the pair verifies with nothing signed. Signing the hash instead (Sign(h(msg))) forces the attacker to invert the hash to find a matching message.

C2. A content filter checks each TCP segment for "root"; one ends "...ro", the next starts "ot...". Why is the word missed, and what should the filter inspect? [L31]

Neither segment contains the whole word; inspect the reassembled TCP byte stream, interpreted the way the receiving endpoint will (also defeating out-of-order and TTL tricks).

C3. Why can a ping accept a forged echo reply, and why does changing only the ICMP identifier stop it? What does this say about source IPs? [WA5]

The reply matches the plaintext fields ping checks (addresses, identifier, sequence), all observable. A changed identifier breaks the match, so ping does not associate the reply even though a capture still shows it. A source IP is a value the sender writes, so it is not authentication.

C4. TLS validation succeeds for amazon.money.example (attacker-controlled). Does HTTPS stop the operator reading Alice's password? Does a valid certificate prove she is at amazon.com? [L33]

No and no. TLS protects data to the authenticated endpoint, which is the attacker's own server, so it decrypts the password. Domain validation proves control of that hostname, not honesty or any relation to amazon.com.

C5. Why does the same-origin policy fail to stop stored XSS, and what actually prevents it? [L34]

Injected code runs inside the site's own origin with the viewer's session, so the SOP treats it as legitimate. Prevention is at the application: output-encode or sanitize untrusted content, add CSP, set HttpOnly.

C6. Name the four buffer-overflow defenses and, for each, say whether it stops the unchecked write itself. [L29]

Bounds checking stops the write. A canary only detects a contiguous overwrite before return. A non-executable stack blocks injected-stack execution but not code reuse. ASLR only makes addresses hard to predict. Only bounds checking prevents the write.

10. One-screen cram sheet

Read this the night before. Every item is a term or fact stated on a slide or in a posted solution.

L28 Signatures. Digital signature = ensure a message comes from a given source. Properties: authentication, integrity, accountability (non-repudiation; MAC can't, key shared). Requirements: unforgeable, authentic, unalterable, not reusable. tag = Sign_Ks(msg); isValid = Verify_Kv(msg, tag); Ks private, Kv public; sender uses private key. RSA sign sig = m^s mod N, verify m == sig^v mod N; raw RSA forgeable (pick sig, msg = sig^v); fix = hash-then-sign. Encrypt-then-sign: enc w/ Bob pub, sign w/ Alice priv, verify w/ Alice pub, dec w/ Bob priv. Caution: beware determinism; unique keys per purpose; don't roll your own. Quantum: qubits, superposition, interference; Shor's breaks RSA + DH (factoring, discrete log); harvest-now-decrypt-later. PQ: ML-KEM (FIPS203, secret), ML-DSA (204, sig), SLH-DSA (205, sig).

L28/L29 Memory bugs. Trusted stack: App → OS → ISA → HW. Integer overflow: 255+1 = 0; 4-bit signed login counter wraps 7→−8, defeats lockout (fix: check before increment; protects online guessing only). malloc(len*4) overflow → small alloc, copy loop overruns = stepping stone. Underflow → huge unsigned alloc. Error/fault/failure/flaw. Overflow=write past end; overrun=read past; attack=gain control; stack smashing=overwrite return addr. Frame A local vars, B params, C return ptr, D saved base ptr, E buffer; A highest addr, buffer fills up toward C. strcpy no bounds check → overwrite C → jump to shellcode (/bin/sh). Defenses: canary (secret between buffer & return), W⊕X (writable xor executable), ASLR (randomize base addrs). Attacks back: return-into-libc, ROP (gadgets end in ret). Backdoor = hardcoded entry in binary. Priv-esc: fixed-fn→shell, sandbox→FS, non-root→root, user→kernel.

L30 Secure code + net intro. Guidelines: check bounds, validate/sanitize, safe utilities, least privilege, sandbox, type safety. Static (no overhead, full program, early, but false positives) vs dynamic (overhead, per-run, late, no false positives). Template = allowed inputs, default deny. strcpy→strncpy, sprintf→snprintf. Least privilege limits damage. Sandbox = partial FS (/tmp). Type safety: memory used per its type; memory safety: never touch undefined memory (overflow, null deref, use-after-free, uninit, illegal free); type-safe → memory-safe. Heartbleed = overrun: trust claimed heartbeat length, leak adjacent memory (keys); OpenSSL = TLS lib; 2012→2014. Net challenges: anonymity, many points, sharing, complexity, unknown perimeter/path. Threats: intercept=C, modify+fabricate=I, interrupt=A. Dolev-Yao active attacker carries the message. Firewall = gateway allow/deny/modify; reference monitor (complete mediation, immune, proven).

L30/L31 TCP + filters. Layers: App, Transport(TCP, port, segment), Network(IP, addr, packet), Data link(MAC, frame), Physical. IPv4 hdr 20B: TTL, protocol, src, dst. TCP flags URG ACK PSH RST SYN FIN. Handshake: SYN(x) → SYN+ACK(y, ack x+1) → ACK(ack y+1); SYN eats 1 seq; handshake ≠ identity. Packet filter = compare to rules, allow/drop/reject. Categories: default deny/allow, stateless/stateful, shallow/deep. Mail rules A–E, first match wins, E=default deny. Pitfalls: high dest port ≠ reply (proxy 8080 attack) → add src port; src port 25 chosen by attacker → add ACK (only opening SYN has ACK=0); ACK set still ≠ connection exists → stateful. Signature filter inspects payload ("root", /etc/passwd); evade: split, reorder, TTL decoys; fix: track/reassemble TCP stream as endpoint sees it.

WA5 + L33 IDS. WA5: attacker on path sees victim's ICMP echo requests (no forwarding, so no real reply); forged reply reverses src/dst, copies id+seq+payload, type 0; change id → ping won't match (capture still shows it). Source IP ≠ authentication. IDS: intrusion = policy-violating event; IDS alerts (evidence, not proof), IPS also blocks; NIDS packets/network-wide, HIDS logs+syscalls, not blocked by encryption. Matrix TP FP / FN TN. FPR=FP/(FP+TN), TNR=TN/(FP+TN), FNR=FN/(TP+FN), TPR=TP/(TP+FN), AP=TP/(TP+FP). Base rate: rare intrusions + small FPR → mostly false alarms. 50k events, 2 intr, TPR99 TNR98 → ~1002 alarms; 100 alarms, 2 real → precision 2%. Tradeoff: FN miss intrusions, FP alert fatigue.

L33 Tunnels, SSH, TLS, PKI. Tunnel = traffic inside another stream; not auto-encrypted; VPN = protected tunnel; encapsulation = outer IP (tunnel endpoint) wraps inner IP (real dest). SSH layers TCP/IP → Transport → User Auth → Connection; server auth: host signs, client verifies; trust via CA / host-key DB / TOFU (detects change, not first-time identity). User auth: password, Kerberos ticket, client public key. HTTP = request/response, no protection; TLS adds confidentiality+integrity+server auth; HTTPS = HTTP over TLS. PKI binds identity ↔ key: certs, CAs sign, trust stores, revocation. Cert = subject, public key, issuer, issuer signature over hash(fields) w/ CA private key. Chain: leaf←intermediate←root(self-signed); browser must already trust root. Check: hostname, validity, chains to trusted root, revocation. Handshake: key exchange→secrets, cert+sig→auth, finished→confirm, session keys→data. Cert ≠ safety rating: domain validation only; attacker site can have valid cert; HTTPS protects transit, not app code.

L34 XSS. Browser runs the page's JS; session cookie = logged-in marker. DOM = client-side representation of the HTML doc (window→document→cookie/location/element). Origin = scheme+host+port; SOP: page from one host can't read/modify another host's content; origins match iff triplet matches. Stored XSS: script saved on server, served to every viewer, runs in site's origin with viewer's session. Script can alert, read document.cookie, exfil via image src GET, make authenticated requests with the page's token, self-copy (worm). HttpOnly blocks direct cookie read, not authenticated actions. Prevent: output-encode for context, sanitize allowed HTML, CSP, HttpOnly. HTTPS protects connection, SOP protects across origins, XSS runs inside the origin → fix at the app.

Numbers, dates, gotchas. Quiz 5: Fri Nov 20, in class, last quiz; 5 quizzes = 40%; final can replace lowest quiz. RSA toy: s3 v7 N33, m2 → sig8. Login counter 4-bit signed −8..7. BUF_SIZE lab 97, str 517. Heartbleed 2012→2014. Handshake SYN eats 1 seq. IDS 50k/2: ~1002 alarms; precision 2%. Mail port 25, client >1023. Lab5 Elgg users admin/alice/boby/charlie/samy. Watch wrong-option MC traps: "never echo a string", "server should never respond". Always answer WHY.

Built November 20, 2026 from the Lecture 28–34 slide decks, in-class practice sheets and posted solutions, WA5, and Extra Practice 5 of COMP 435 (Fall 2026, UNC). Personal study material.
© 2026 Deven Jadhav. Not affiliated with or endorsed by UNC-Chapel Hill; course material © its respective owners. Willful large-scale copyright infringement may be a federal crime (17 U.S.C. §506).