Class

1

CSC 472 Software Security

Introduction

Dr. Si Chen (schen@wcupa.edu)

Tue / Thu · 2:00 – 3:15 PM · 25 University Ave., Room 158
Fall 2026 · Week 1 · Tuesday, August 25, 2026
Prereq: CSC 231/242 and CSC 302 · wcu.ninja/csc472

PART 1

What is software security?

Correctness prevents accidents. Security prevents adversaries.

The one-sentence version

- CSC 472 is for people interested in **software security, reverse engineering, and low-level software**.
- We study **vulnerabilities and the attacks that exploit them** — buffer overflows, ROP, format strings, heap & kernel bugs, SQL injection, session hijacking —
  - ...and the **defenses**: ASLR, NX/DEP, PIE, stack canaries, program-analysis techniques.
- Hands-on in the **Badger CTF** environment; scripting in **Python / pwntools**.
- It culminates in an **integrated team Capture-The-Flag** project.

Grade breakdown

5 Hacking Labs — 50%

5 short quizzes — 25%

Final team CTF — 20%

Attendance / participation — 5%

Labs: Stack, Overflow, ROP, Multi-stage, Kernel

Correctness vs. security

Correctness

- Most developers pursue **correctness**: the right behavior on **expected** input.
- A working bank site, editor, or blog is "correct."
- Correctness assumes a **cooperative** user who, on hitting a bug, works around it.
- Companies fix the bugs that affect **normal users** — the rest ship.

Security

- **Security** prevents **undesired behavior** even against a **motivated adversary** —
- ...who **actively** hunts rare feature interactions and edge cases, then **exploits** them.
- **Key difference:** an adversary is not a normal user. A bug is not an annoyance to them — it is an opportunity.
- To be secure we must **eliminate flaws, or make them harder to exploit**.

The CIA triad — what we protect

Confidentiality

Information stays private.

Secrets, credentials, PII, source code.

Attack: steal data, sniff, leak.

Integrity

Data & behavior stay trustworthy.

No unauthorized modification.

Attack: tamper, inject code, forge, destroy records.

Availability

The system stays usable when needed.

Service responds; resources are free.

Attack: denial-of-service, resource exhaustion.

Vulnerabilities and exploits

- A **vulnerability** is a specific flaw or oversight that lets an attacker cause undesired behavior.
- It is a **subset of software bugs** — the **security-relevant** ones. Most bugs are not exploitable; the skill is telling which are.
- An **exploit** is the concrete input/technique that *uses* a vulnerability. A **PoC** proves it can be done; an exploit weaponizes it.
- A **0-day** is a vulnerability with no patch yet — "zero days" for the vendor to respond.
  - "A complex program, written by experts and deployed for a decade, can suddenly be co-opted by attackers — some form of digital voodoo."

This still happens — recent breaches (2021–2024)

Log4Shell · 2021

Apache Log4j RCE — CVE-2021-44228.

One crafted string, ${jndi:ldap://…}, logged anywhere = remote code execution.

Trivial to exploit, in software everywhere.

MOVEit · 2023

SQL-injection zero-day — CVE-2023-34362.

Cl0p ransomware mass-exploited it: data from thousands of orgs, tens of millions of people.

A course topic — SQL injection — at global scale.

XZ Utils backdoor · 2024

Supply-chain backdoor in liblzma targeting SSH — CVE-2024-3094.

A ~2-year social-engineering campaign by "Jia Tan."

Caught by luck: a 500 ms SSH slowdown.

regreSSHion · 2024

OpenSSH signal-handler race — CVE-2024-6387.

Unauthenticated remote root; 14M+ servers exposed.

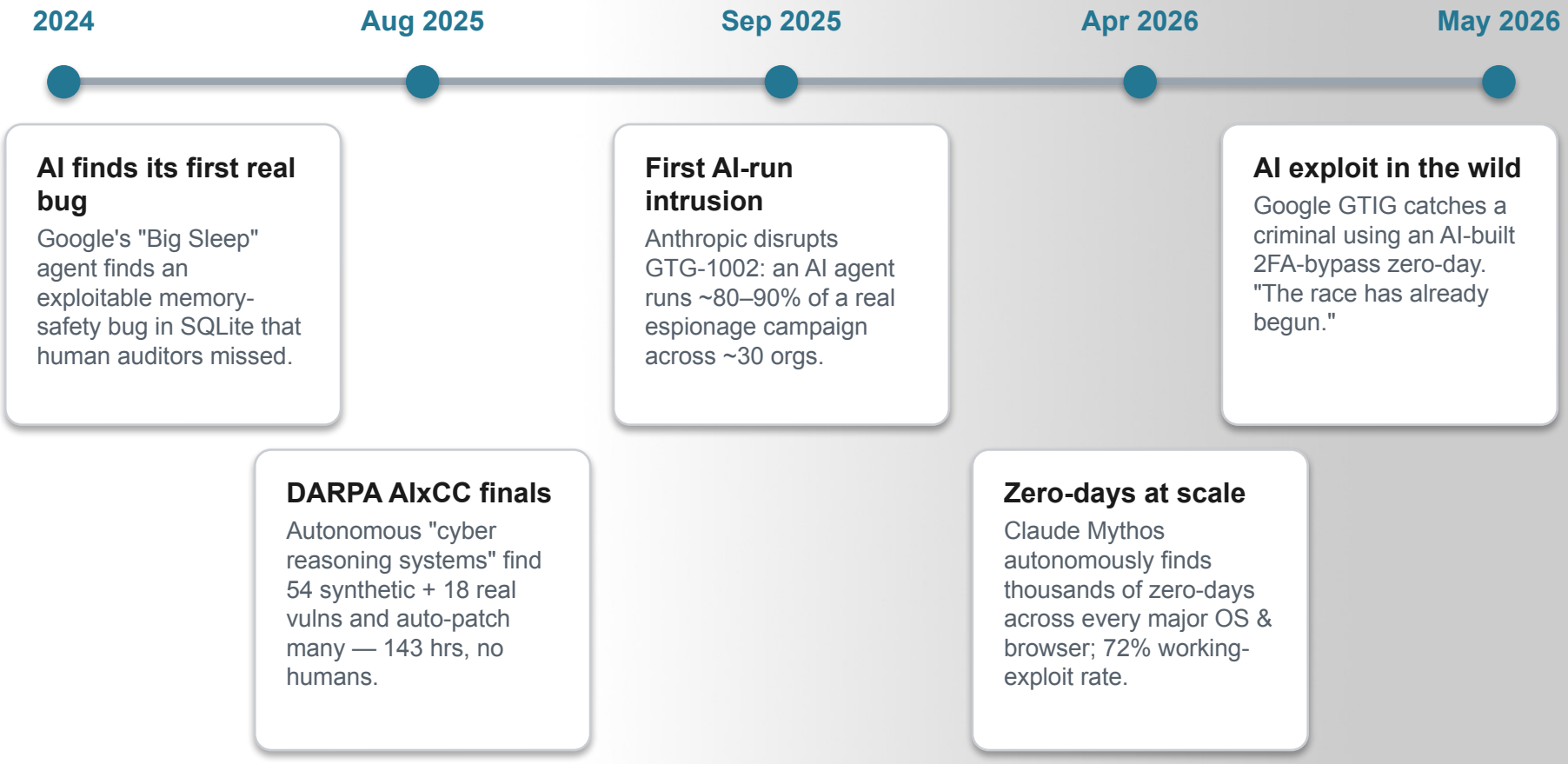
A memory-safety bug — exactly what we study.

PART 2

The AI moment in security

In 24 months, AI went from "helps write scripts" to "autonomously finds and exploits zero-days"

The exploit gap collapsed — fast



The numbers that should get your attention

Thousands

of zero-day vulnerabilities autonomously discovered by one AI model across every major OS & browser

Claude Mythos, Apr 2026

72%

working-exploit success rate — not just finding bugs, but weaponizing them

Anthropic / Help Net Security

80–90%

of a real-world espionage campaign executed autonomously by an AI agent

Anthropic GTG-1002, 2025

A 27-year-old OpenBSD bug, a 16-year-old FFmpeg bug, a 17-year-old FreeBSD RCE — all surfaced by AI for under $20K of compute. The vulnerabilities were always there. What changed is who can find them, and how fast.

Two sides of the same coin

AI as attacker

- **Offense got cheap.** Finding and weaponizing bugs, once elite and slow, is increasingly automated.
- Agents do recon, exploit-dev, lateral movement — in parallel, across many targets.
- A 27-year-old OpenBSD DoS found in ~1,000 runs for <$20K.
- The "exploit gap" — bug to working exploit — is collapsing.

AI as defender

- **Defense got a new weapon too.** AIxCC proved AI can find *and patch* at scale.
- 4 of 7 winning cyber-reasoning systems were **open-sourced** for defenders.
- Anthropic launched **Project Glasswing** — mass vulnerability coordination.
- The winners: **humans who can direct and verify** these systems.

> *AI output is not evidence by itself. Security claims must be validated against the actual target — debugger output, program behavior, a successful exploit. Treat generated output as a hypothesis to test, not as ground truth.*

— CSC 472 Syllabus — Generative AI policy (paraphrased)

Why AI makes deep security skills MORE valuable

AI is confidently wrong

Hallucinates offsets, gadget addresses, mitigation behavior.

In exploitation, wrong = crash. Verify in the debugger.

You cannot supervise what you do not understand.

The bugs are still written

AI code fails security checks ~44% of the time.

"Vibe coding" → 35+ CVEs in one month (Georgia Tech, 2026).

More code, more bugs, more surface to defend.

Someone accountable is needed

Attackers automate; defenders need people who direct AI and own the result.

CTF-grade skills = that supervisory capability.

A growth field: AI-security roles face a 63% talent shortage.

How we use AI in CSC 472 — encouraged, but on a leash

- **Encouraged:** explain C/asm/pwntools, generate & troubleshoot scripts, suggest exploit hypotheses, interpret crashes and register/memory state.
- **You remain responsible** for every command, script, and payload — be able to explain and modify all of it.
- **Validate everything** against the actual course target. A plausible explanation is not a working exploit.
- **Never** give external AI systems course credentials, private CTF flags, or another student's work.
  - Quizzes: no AI unless explicitly open-AI. Labs & CTF: AI allowed under this policy.

PART 3

The landscape of this course

Both hats: how vulnerabilities are exploited, and how they are prevented

Black hat & white hat — we wear both

Attacker

- **Black hat (offense):**
- What security-relevant defects are vulnerabilities?
- How are they exploited?
- You cannot defend what you do not understand attacking.

Defender

- **White hat (defense):**
- How do we prevent defects before deploying?
- How do we make the ones we miss harder to exploit?
- Design, implement, review, and test for security.

Low-level vulnerabilities we will study

- Programs in **C/C++** are susceptible to dangerous memory-safety flaws:
  - **Buffer overflows** — on the stack and the heap; from integer overflow; over-reading (Heartbleed).
  - **Format-string** mismatches; **dangling-pointer** / stale-memory dereferences.
- These enable attacks: **stack smashing**, **format-string attacks**, **return-oriented programming (ROP)**.
  - All are violations of **memory safety** — accessing memory via pointers that do not own it.

Defenses — and beyond memory bugs

Memory-safety defenses

- **Make exploitation harder:**
- Stack canaries; non-executable data (NX / DEP).
- Address space layout randomization (ASLR); PIE.
- Control-Flow Integrity (CFI); safe libraries.
- Key idea: **validate untrusted input.**

Web security

- **Web security** — a whole other surface:
- SQL injection, cross-site scripting (XSS),
- cross-site request forgery (CSRF), session hijacking.
- Same theme: **be careful what you trust; reduce the damage.**

PART 4

A taste of the target

The kind of bug we will spend the semester weaponizing — and defending

Why C is dangerous: the classic overflow

```
#include <string.h>

void vulnerable(char *input) {
    char buffer[64];        // 64 bytes on
the stack
    strcpy(buffer, input);  // no bounds
check!
    printf("Got: %s\n", buffer);
}

int main(int argc, char **argv) {
    vulnerable(argv[1]);    // attacker
controls input
    return 0;
}
```

- `buffer` holds **64 bytes** — but `strcpy` copies **whatever** the attacker sends.
- Send 100 bytes → **overwrite the saved return address**.
- Control the return address → **control what the CPU runs next**.
- This one pattern (CWE-121) underlies decades of real compromises — including Heartbleed's cousin, the over-*read*.

You do not need every detail today — this is the destination, not the starting point.

Warm-up: think like an attacker

IN-CLASS ⏱ 6–8 min

- No computer needed — reason it out, discuss with a neighbor.
- **1. Spot the assumption:** what did the programmer *assume* about `input` (previous slide) that an attacker will violate?
- **2. Estimate:** `buffer` is 64 bytes. Roughly how many bytes before you reach the saved return address? What sits in between?
- **3. Defense:** name one change — in the *code*, the *compiler*, or the *OS* — that makes this attack harder.

Hints:

• Assumption: input fits in 64 bytes.

• Between buffer & return addr:

saved frame pointer + padding.

• Defenses: strncpy / bounds checks,

stack canaries (compiler),

NX + ASLR + PIE (OS).

No single "right" number —

reasoning about the layout is the skill.

The mindset — and where Class 2 goes

- **Work from evidence.** When an exploit fails, inspect registers, memory, stack, addresses, protections, exact output — *before* changing the payload.
- **Build incrementally.** Prove control, then the primitive, then the next stage.
- **Use AI aggressively — then verify in the target.** A plausible explanation is not a working exploit.
- **Thursday (Class 2):** we drop to the machine — IA-32 **registers** (ESP/EBP/EIP) and **byte ordering** — the facts every exploit on the previous slide depends on.

Before Thursday

- Confirm you can **SSH into the course Linux environment** and launch **`gdb`** on a small binary.
- Review your **CSC 231/242** x86 basics and **CSC 302** systems background.
- Read the **syllabus** (course site & D2L): the AI + credential/flag policy, lab-scope rules, grading.
- Course site: **wcu.ninja/csc472**. Badger CTF credentials come in class / on D2L. No graded work is due yet.

Sources & further reading

- Security model, CIA, vulnerabilities — adapted from the CSC 472 course materials (ch01)
- Log4Shell — CVE-2021-44228 (Apache Log4j); MOVEit — CVE-2023-34362 (Cl0p; CISA AA23-158A)
- XZ Utils backdoor — CVE-2024-3094 (Andres Freund disclosure); regreSSHion — CVE-2024-6387 (Qualys, Jul 2024)
- Google "Big Sleep" — SQLite bug (2024); 20 zero-days (Aug 2025) — Project Zero / DeepMind
- DARPA AIxCC final results — darpa.mil (2025); AIxCC SoK, USENIX Security 2026
- Anthropic GTG-1002 — first AI-orchestrated espionage campaign — anthropic.com (2025)
- Claude Mythos — thousands of zero-days, 72% exploit rate — Help Net Security (Apr 2026)
- Google GTIG — first AI-built zero-day in the wild — GTIG (May 2026)
- Veracode 2026 GenAI Code Security Report — veracode.com; Georgia Tech Vibe Security Radar (2026)
- CSC 472 Fall 2026 Syllabus; Bryant & O'Hallaron, CS:APP, 3rd Ed.