Code Hardening: C/C++

CEN429 Secure Programming — Week 4

Asst. Prof. Dr. Uğur CORUH · 09.10.2026

Today's Plan (3 Hours)

Hour	Section	Topic
1	0–2	Basic concepts · layers of hardening · SEI CERT · input validation
2	3–4	Format string · UAF · integer/UB · error/signal · static analysis · sanitizers · fuzzing
3	5–7	Compiler/OS protections · secure build pipeline · introduction to code obfuscation · project

A Brief History — the Race Between Memory Bugs and Defences

- **1988** — **Morris Worm** announces the buffer overflow to the world
- **1996** — *Smashing the Stack for Fun and Profit* teaches exploitation to everyone
- **1998 → 2004** — **canary** (StackGuard) · **ASLR** (PaX) · **DEP/NX** · FORTIFY
- **2012–13** — **AddressSanitizer** and **fuzzing** (AFL) catch bugs automatically

> This course follows this race: write the bug → let the tool catch it → let the compiler/OS protect.

Where Does This Week Fit?

- **Weeks 1–3:** security principles, threat model, cryptography.
- **This week (4): hardening** C/C++ code — write it bug-free, limit the damage if a bug slips through, make it harder to read.
- **Week 5:** the same topic for Java/interpreted languages.

Learning Outcome

This week is about **LO.3** (binary application protections).

By the end, you will be able to:

- Recognise and fix common C/C++ vulnerabilities
- Use static analysis, sanitizers, fuzzing
- Turn on and verify compiler/OS protections

Three Layers, the Right Order

Code hardening is three layers. The order matters:

1. **First, bug-free code** (secure coding)
2. **Then, limit the damage** (compiler/OS protections)
3. **Finally, make it harder to read** (obfuscation)

> ⚠️ Obfuscation does **not fix** the bug.

Why This Order?

- Adding protection to buggy code is like painting over a rotten foundation.
- First **remove** the bug; then **catch** what slips through; finally **slow down** reverse engineering.
- We will see all three this week, in this order.

0. Basic Concepts (From Scratch)

Memory: Stack and Heap

- **Stack:** automatically managed memory that holds the local variables of function calls.
- **Heap:** memory allocated **manually** with `malloc` / `new`, released with `free` / `delete`.

Both can be a source of overflows and bugs.

Pointer

- **Pointer:** a variable that holds a memory **address**.
- `p` is an address; `*p` is the value at that address.
- Wrong address → crash or wrong data.

Buffer and Overflow

- **Buffer:** a contiguous block of memory (e.g., `char ad[16]`).
- **Overflow (buffer overflow):** writing more data than the buffer can hold → corrupts neighbouring memory.
- A classic and dangerous class of bug.

Why Is an Overflow Dangerous?

- It can corrupt neighbouring variables or the return address.
- An attacker can use this to hijack **control flow**.
- This is why **bounds checking** is vital.

What Is Undefined Behaviour (UB)?

- **Undefined behaviour:** situations the C/C++ standard calls "the result is unspecified."
- Example: signed integer overflow, accessing past the end of an array.
- The compiler may handle this **however it likes** — it may even **delete** the corresponding check.

Compiler Flag

- **Flag:** an option passed to the compiler.
- Example: `-O2` (optimisation), `-Wall` (warnings), `-fsanitize=address`.
- The right flags catch many bugs **at compile time**.

Warning vs Error

- **Error:** compilation stops.
- **Warning:** compilation continues but a problem is reported.
- Rule: **turn warnings into errors** (`-Werror`) — an ignored warning is a future vulnerability.

What Is a CWE?

- **CWE (Common Weakness Enumeration):** a numbered catalogue of software weaknesses.
- Example: CWE-416 = "use-after-free."
- Naming a finding with its CWE number makes it **searchable**.

What Is CERT?

- **SEI CERT C/C++:** the **rulebook** of secure coding.
- Every rule: a noncompliant example + a compliant solution + risk + a CWE link.
- Example: `STR31-C` = allocate sufficient memory for the string.

Static vs Dynamic Analysis

- **Static analysis:** examining code **without running** it (compiler warnings, clang-tidy).
- **Dynamic analysis:** observing code **while it runs** (sanitizers).
- The two complement each other.

What Is a Sanitizer?

- **Sanitizer:** a compiler tool that catches memory/UB bugs while the program runs.
- Example: **ASan** (address), **UBSan** (undefined behaviour).
- It does not ship in the release build; it is used in **testing/CI**.

What Is Fuzzing?

- **Fuzzing:** feeding a program **random/unexpected** inputs and looking for crashes.
- It finds inputs a human would never think of.
- Very powerful when combined with a sanitizer.

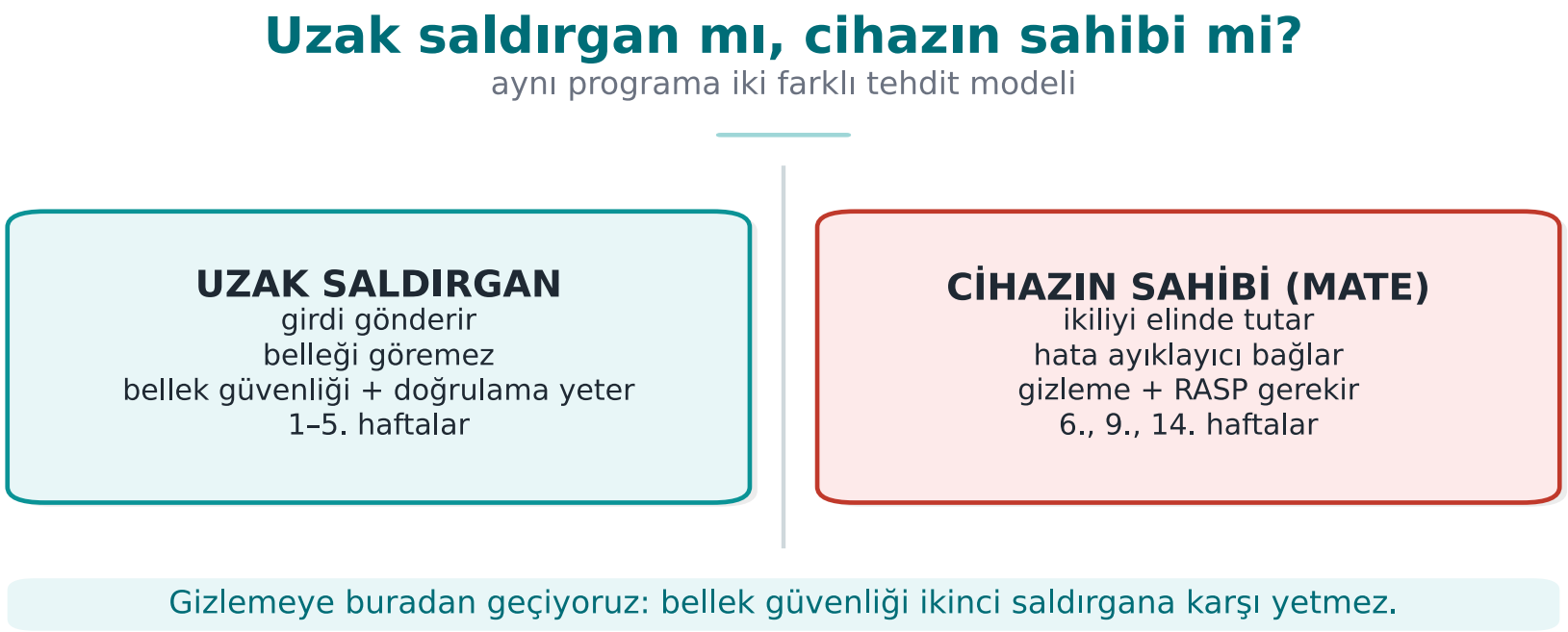
ASLR, NX/DEP, Canary

- **ASLR: randomizes** memory addresses (so the attacker cannot guess an address).
- **NX/DEP:** prevents bytes in a data region from being **executed as code**.
- **Stack canary:** a **sentinel value** just before the return address; if an overflow corrupts it, the program halts.

White-Box Attacker (Reminder)

- An attacker who possesses the program (Week 1's MATE).
- Reads strings, finds functions by name, bypasses checks.
- We will return to this in the obfuscation section (later today).

Two Attacker Models — Diagram



CI (Continuous Integration)

- **CI (Continuous Integration):** a system that runs an automatic build + tests on every code change.
- It runs security tools (warnings, static analysis, sanitizers, fuzzing) **on every merge**.

Now We're Ready

Terms:

stack/heap · pointer · buffer/overflow · UB · flag · warning/error · CWE · CERT · static/dynamic · sanitizer · fuzzing · ASLR/NX/canary · CI

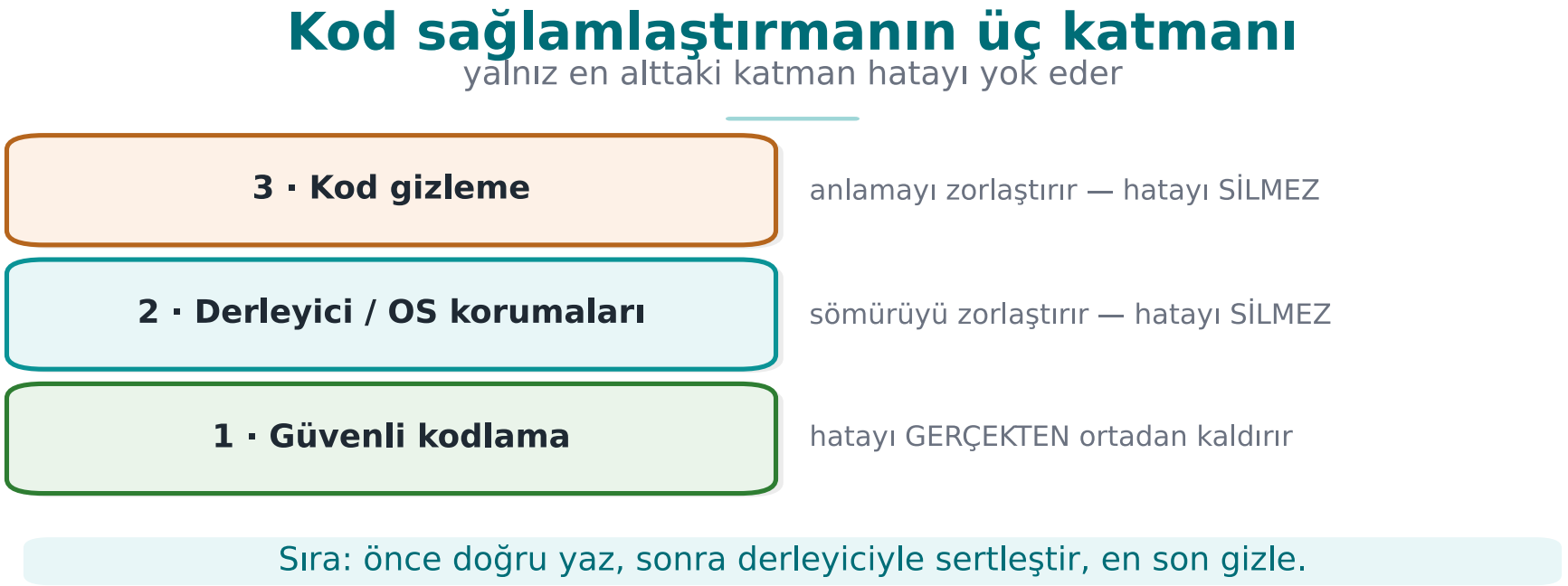
Now: the layers of code hardening.

1. Layers and SEI CERT

Anatomy of a CERT Rule — Diagram

Kuralı ezberlemeyin: hatalı ile uyumlu kodun FARKINI okuyun; öğretici olan orasıdır.

Three Layers — Diagram



Layer 2 · Secure Coding

- **Question:** is there a bug in my code?
- **Tools:** CERT rules, sanitizers, fuzzing.
- **Goal:** not writing the bug in the first place.

Layer 3 · Compiler/OS

- **Question:** if a bug slips through, is it hard to exploit?
- **Tools:** canary, FORTIFY, ASLR, NX, RELRO, CFI.
- **Goal:** making exploitation of the remaining bug expensive.

Layer 4 · Obfuscation

- **Question:** does someone reading the code understand it?
- **Tools:** symbol/string hiding, removing logging, flattening.
- **Goal:** slowing down reverse engineering.

From the Field: Scheme Requirements

In a certified product:

- "Common vulnerabilities must be addressed: sensitive data, injection, overflow, language-specific weaknesses"
- "Defensive programming **consistent across the entire codebase**"
- "Protection against static and dynamic reverse engineering"

➡️ Eighteen hardening measures on the native side **alone**.

SEI CERT: What Does a Rule Look Like?

Parts of every CERT rule:

- **Identifier:** `STR31-C`, `MEM50-CPP`
- **Title:** one sentence
- **Noncompliant example** and **compliant solution**
- **Risk level** (L1/L2/L3) and a **CWE** link

Rule or Recommendation?

- **Rule:** violating it is a **vulnerability**, and it can be checked automatically.
- **Recommendation:** good practice.
- Start a review with **L1** (highest risk) rules.

How Is the Risk Level Determined?

Risk = **severity × likelihood × cost to fix**.

- **L1:** high risk, these first.
- **L2 / L3:** progressively lower.

CERT Categories (1)

Abbreviation	Category	Example
INT	Integers	INT32-C: do not allow signed overflow
ARR/STR	Array/string	STR31-C: allocate sufficient space
MEM	Memory	MEM30-C: do not access freed memory
FIO	File I/O	FIO30-C: exclude input from the format string

CERT Categories (2)

Abbreviation	Category	Example
ENV	Environment	ENV33-C: do not call `system()`
SIG	Signals	SIG30-C: only safe fn. in the handler
ERR	Error handling	ERR33-C: detect a library error
CON/MSC	Concurrency/misc	CON43-C, MSC32-C

C++: MEM50-CPP, CTR50-CPP, STR50-CPP, EXP53-CPP, ERR50-CPP.

Automated CERT Auditing

- Warnings: `-Wall -Wextra -Wformat=2 -Wconversion`
- `clang-tidy cert-*`
- `cppcheck --addon=cert`
- MSVC `/analyze`

> The evaluator runs this scan first, then reads by hand.

Input Validation

Input Validation — Diagram

Kara liste değil İZİN listesi: neyin yasak olduğunu değil, neyin serbest olduğunu yazın.

Where Does Input Come From?

From more places than you'd think. And **all of it is untrusted**:

- Command line, environment variables
- File, network
- IPC (another process on the same machine)
- Database, library interface (JNI)

Commonly Forgotten Inputs

- Even `argv[0]` can be the attacker's; `PATH`, `LANG`.
- **Length fields** inside a file format.
- Network: says "16 bytes," sends 1 byte.
- "Our own data" may have changed since.

Five Principles of Input Validation

1. **Allow-list** (enumerate what's permitted), not a deny-list.
2. **Canonicalize first**, then validate.
3. **Length → type → range → format**, in that order.
4. At the **trust boundary**, as early as possible.
5. **Reject**, don't "clean up" and use it anyway.

Principle 1 · Allow-List

- **Deny-list:** "these are forbidden" — something always gets forgotten.
- **Allow-list:** "only these are permitted" — a safe default.

Example: a username may only contain `[a-z0-9_]`.

Principle 2 · Canonicalize

- The same thing, spelled differently: `/tmp/../etc`, `%2e%2e/`.
- Reduce it to **one form** (canonicalize) first, then check it.
- Otherwise the check can be bypassed.

Principle 3 · `strtol`, Not `atoi`

```
errno = 0;
long v = strtol(s, &son, 10);
if (son == s)        return -1;  /* rakam yok */
if (*son != '\0')    return -1;  /* "12abc" */
if (errno == ERANGE) return -1;  /* taştı */
if (v < en_az || v > en_cok) return -1;  /* iş kuralı */
```

Why Is `atoi` Dangerous?

- `atoi("abc")` = 0 (the error is silent)
- `atoi("99999999999")` = **undefined** (overflow)
- `atoi("12abc")` = 12 (the extra characters are ignored)

> `strtol` reports **every error** to you; `atoi` does not.

Principle 3 · a Length-Prefixed Record

```
if (kalan < 3) return -1;                  /* başlık yok */
uint16_t uzunluk = tampon[1] << 8 | tampon[2];
if (uzunluk > kalan - 3)      return -1;   /* KAYNAK: bildirilen > gelen */
if (uzunluk > sizeof k->veri) return -1;   /* HEDEF: sığmıyor */
memcpy(k->veri, tampon + 3, uzunluk);
```

Two Separate Checks, the Right Order

- **Source check:** is the declared length greater than the data that arrived?
- **Destination check:** does it fit in the destination buffer?
- Order: `kalan < 3` **first**, so `kalan - 3` does not wrap.
- In the field: every array coming from JNI is **re-validated** on the native side.

CERT Pair · STR31-C

```
strcpy(kopya, ad);                                  /* HATALI */
int n = snprintf(kopya, sizeof kopya, "%s", ad);/* UYUMLU */
if (n < 0 || (size_t)n >= sizeof kopya) { /* kesildi */ }
```

`strcpy` does not know bounds; `snprintf` knows the size and reports **truncation**.

CERT Pair · INT30-C

```
size_t kalan = toplam - okunan;   /* HATALI: okunan>toplam → dev sayı */
if (okunan > toplam) return HATA; /* UYUMLU: önce sırala */
size_t kalan = toplam - okunan;
```

Unsigned subtraction **wraps**; check the logic first.

CERT Pair · MEM30-C

```
for (d = bas; d; d = d->sonraki) free(d);       /* HATALI */
while (d) { Dugum *s = d->sonraki; free(d); d = s; }  /* UYUMLU */
```

Reading `d->sonraki` after `free(d)` = accessing freed memory.

CERT Pair · ERR33-C

```
FILE *f = fopen(yol,"rb"); fread(t,1,n,f);       /* HATALI */
if (!f) return HATA;
if (fread(t,1,n,f) < n && ferror(f)) return HATA;/* UYUMLU */
```

Check **every** return value; `fopen` can return NULL.

Rule: Tie the Finding to an Identifier

Match every finding to a CERT identifier:

> "This line is a MEM30-C violation (CWE-416)."

This makes the finding **objective** and **searchable**.

Section 1 — Quick Check

1. The three layers, and the right order?
2. Difference between a rule and a recommendation?
3. Why is `atoi` more dangerous than `strtol`?

Section 1 — Answers

1. **Order:** secure code (source) → **compiler/tool** protections → **OS/runtime** protections. First write it correctly, then harden it, then defend it at runtime.
2. A **rule** always applies, and violating it is a defect; a **recommendation** is good practice depending on context.
3. `atoi` does **not report** errors (undefined/0 on overflow/invalid input); `strtol` lets you check **both the error and the bound** via `endptr` + `ERANGE`.

2. Format String Vulnerability

Format String Vulnerability — Diagram

Biçim dizisi açığı

tek fark: biçim dizgesini kim belirliyor

printf(girdi)
girdi BİÇİM DİZGESİ sayılır

%x → yığını okur
%n → belleğe YAZAR
→ bilgi sızıntısı, kod akışı

printf("%s", girdi)
biçim dizgesi SABİT

girdi yalnız VERİ

güvenli

En tehlikelisi %n: yazdığı bayt sayısını belleğe yazar → seçili adrese yazma.

The Problem · `printf(girdi)`

```
printf(kullanici_girdisi);         /* HATALI */
printf("%s", kullanici_girdisi);   /* DOĞRU */
```

In the first case, user input is interpreted as a **format string**.

Why Is It Dangerous?

`printf` treats the `%` markers in the format string as a **command**.

If the user supplies `%x`, `%s`, `%n`, they make the program do work.

Marker · `%x` , `%p`

- **Reads** values off the stack and prints them.
- → **information leak** (addresses, secret values).
- Weakens ASLR (if an address leaks).

Marker · `%s`

- Treats a value on the stack as a **pointer** and reads the string at that address.
- Random address → crash or memory read.

Marker · `%n`

- **Writes** the number of characters printed so far to an address.
- → **writes to memory** → corruption, hijacking control.
- The most dangerous one.

The Same Bug Elsewhere

Not just `printf`:

`syslog`, `fprintf`, `snprintf`, `err` / `warn` — they all take a format string.

Rule: the format string is **always constant** (FIO30-C).

Demo 1 — Format String

`code/week-04/01-format-string` · CWE-134

Step	What Is Seen?
0	GCC `-Wformat-security` warns; MSVC `/analyze` catches it
2 `%x.%x...`	The stack is dumped, the **secret (synthetic) value** appears
3 `AAAA%n`	Unprotected Linux crashes; Windows CRT rejects `%n`
4	`_FORTIFY_SOURCE`：`%n in writable segment`
5	Markers appear only as text

The Fix · Layer by Layer

1. The format string is always **constant** (FIO30-C).
2. Warning: `-Wformat -Wformat-security -Werror=format-security`.
3. Have the compiler **check** your own function too:

```
void gunluk_yaz(int duzey, const char *bicim, ...)
    __attribute__((format(printf, 2, 3)));
```

4. Runtime: `_FORTIFY_SOURCE=2`; MS CRT has `%n` disabled.

3. Memory: Use-After-Free

Stack and Heap — Diagram

Yığın (stack) ve öbek (heap)

taşmanın sonucu, belleğin hangi bölgesinde olduğuna göre değişir

YIĞIN
fonksiyonla açılır/kapanır
otomatik, hızlı
boyut derleme anında belli

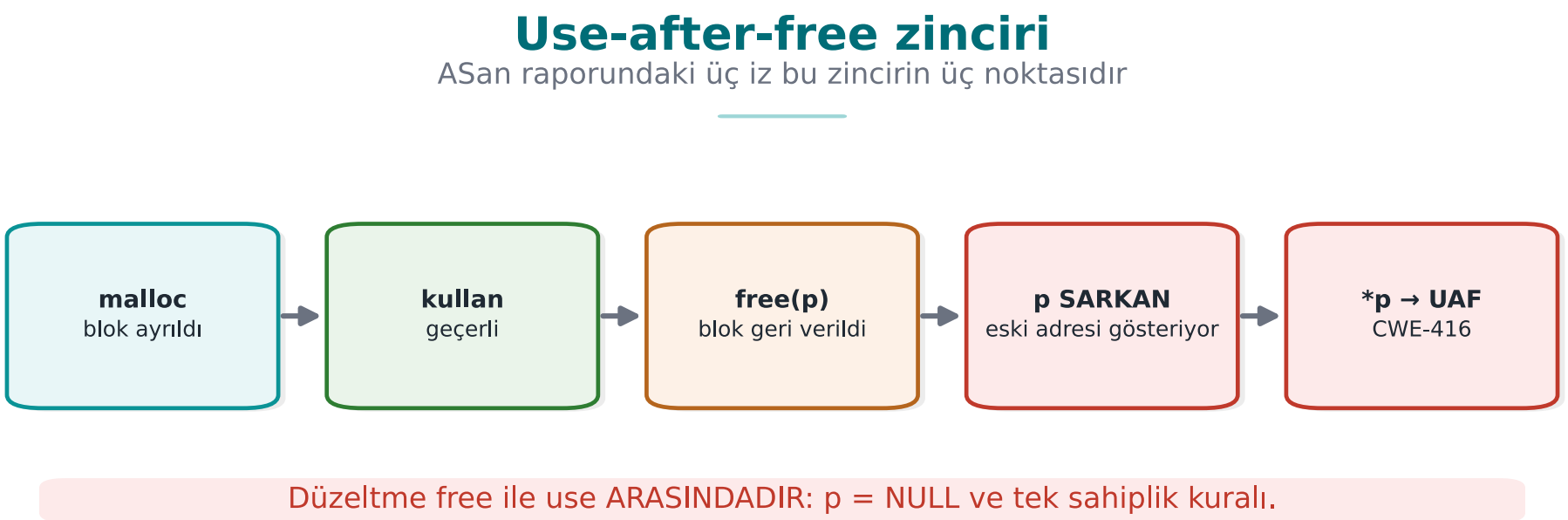
taşma → dönüş adresi tehlikede

ÖBEK
malloc/free ile yönetilir
yaşam süresi size bağlı
boyut çalışma anında

taşma → komşu blok / meta veri

Kanarya yığını korur; öbek taşmasını yakalamaz — bu yüzden ASan gerekir.

Use-After-Free — Diagram



What Is UAF? · Step 1

- An object is on the heap.
- **Two pointers** point to the same object.

UAF · Step 2

- One of them `free`s it.
- The other still holds that address → a **dangling** pointer.

UAF · Step 3

- The allocator gives the same block to **another object**.
- The dangling pointer now reads/writes **foreign data**.
- If the object has a **function pointer** (C++ virtual table), the risk grows.

Why Is UAF So Common?

Most of the "memory corruption fixed" notes in browser/OS updates fall into this class.

Complex ownership → a dangling pointer.

Demo 2 — UAF and Double Free

`code/week-04/02-kullanim-sonrasi` · CWE-416/415 · MEM30-C

Step	What Is Seen?
1	The freed block gets reused → role `user` → `admin`
2 ASan	`heap-use-after-free` + locations
3	Double `free` → undefined behaviour
4 ASan	`attempting double-free`
5	`free` + `NULL`, single owner → safe

ASan Report · Three Stack Traces

```
ERROR: heap-use-after-free
  #0 oturum_kullan uaf.c:48   <- 1. KULLANIM
freed here:
  #1 oturum_kapat  uaf.c:31   <- 2. SERBEST BIRAKMA
allocated here:
  #1 oturum_ac     uaf.c:22   <- 3. AYIRMA
```

The fix is usually at **location 2**: the ownership decision was wrong there.

Fix · the C Side

- Set the pointer to `NULL` after `free`.
- Establish a **single owner**, a single release point.
- Write down the ownership rule: "who releases this memory?"

Fix · C++ Smart Pointers

Need	C++
Single owner	`std::unique_ptr`
Shared	`std::shared_ptr`
Non-owning observation	`std::weak_ptr` → `lock()`
Array	`std::vector` , `std::array`

```
if (auto o = onbellek.lock()) kullan(*o);  // yoksa eski belleğe erişilmez
```

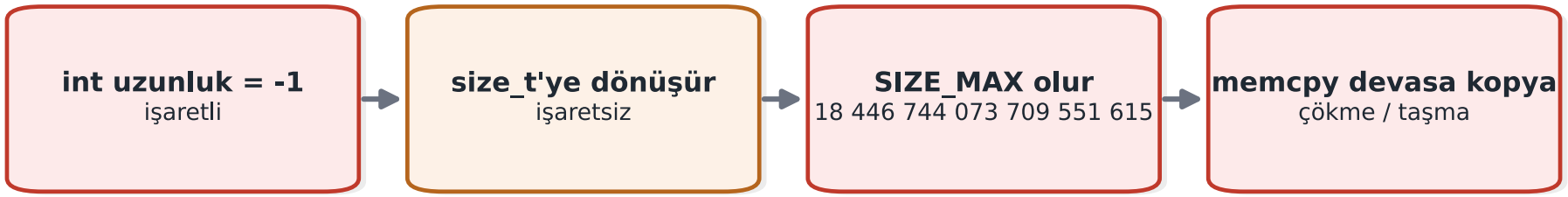
⚠️ `get()` is a raw pointer; when a `vector` grows, its iterators are invalidated.

Integers and UB

Integer Overflow — Diagram

-1 ne zaman dev olur?

işaretli → işaretsiz dönüşümü sessizce gerçekleşir



Kural: uzunlukları size_t ile tutun; dışarıdan gelen sayıyı hem alt hem üst sınırla denetleyin.

Four Integer Bugs

Bug	Example	Result
Unsigned wraparound	`0u - 1` = 4,294,967,295	Logic error
Signed overflow	`INT_MAX + 1`	**Undefined**
Conversion loss	`long long` → `int`	Truncation
Shift	`1 << 31`	Undefined

Why Does It Matter? · Multiplication Overflow

```
uint32_t boyut = adet * 4;    /* adet = 0x40000001 → boyut = 4 */
dizi = malloc(boyut);         /* küçük blok */
/* döngü adet kez yazar → öbek taşması */
```

Writing too much into a small allocated block = an overflow.

UB · the Compiler Can Delete the Check

```
if (x + 100 < x)   /* "taşarsa küçülür" sanısı */
    return -1;     /* işaretli taşma UB → derleyici bu dalı SİLEBİLİR */
```

- It "works" at `-O0`, at `-O2` the check **disappears**.
- UB cannot be relied on.

Correct: Check First

```
if ((b > 0 && a > INT_MAX - b) ||
    (b < 0 && a < INT_MIN - b)) return false;   /* taşma öncesi */
if (__builtin_mul_overflow(a, b, &sonuc)) return false;  /* GCC/Clang */
/* C23: <stdckdint.h> ckd_add/ckd_mul · MSVC: <intsafe.h> */
```

Catch the overflow **before it happens**.

Demo 3 — UB and UBSan

`code/week-04/03-tanimsiz-davranis` · CWE-190/758

- Signed overflow, invalid shift, misaligned access
- On x86, most produce a wrong result **silently**
- `-fsanitize=undefined` → `signed integer overflow: 2147483647 + 1 ...`
- Windows: no UBSan → `/RTC`, `/analyze`, CERT

⚠️ `-fwrapv` **hides** the logic error; do not use it for diagnosis.

Error Handling and Signals

Signal Handler — Diagram

Sinyal işleyicide ne yapılır?

sinyal, kesilen kodun ortasında gelebilir

YANLIŞ
printf, malloc, free
kilit alan her şey

yeniden girişte iç yapı bozulur
→ kilitlenme / çökme

DOĞRU
yalnız bayrak kur:
volatile sig_atomic_t

asıl işi ana döngü yapar
(write() gibi güvenli çağrılar)

Async-signal-safe olmayan fonksiyon çağırmak tanımsız davranıştır.

Unchecked Return Value

Call	If Not Checked
`malloc`	Writing to NULL
`setresuid`	Stays privileged
`RAND_bytes`	The "random" key is zero
`EVP_DecryptFinal_ex`	Tampered data is assumed verified

Rules for Error Handling

1. Check every return value (ERR33-C); `[[nodiscard]]`.
2. On error, return to a **safe state** (`goto cleanup`) — **fail-closed**.
3. The message must not give a hint: "no such user" ≠ "wrong password."

From the Field: Opaque Return Values

- Native functions do not return a detailed **error code**.
- Only success/failure, and even that is **opaque** (Week 9).
- A detailed code = a map for the attacker of "which check you tripped."
- A separate, secure channel for diagnosis → recorded as a **trade-off**.

Signal Handler · Rule

```
static volatile sig_atomic_t durdur = 0;
static void isleyici(int s){ (void)s; durdur = 1; }  /* YALNIZ bayrak */
```

- No `printf`, `malloc`, `free` in the handler (SIG30-C).
- Do not call functions that are not async-signal-safe.
- `sigaction()` instead of `signal()`.

Real Incident: regreSSHion

- 2024 CVE-2024-6387: the timeout **signal handler** called an unsafe logging function.
- Result: remote code execution risk.
- Lesson: only set a flag inside a handler.

Section 2–3 — Quick Check

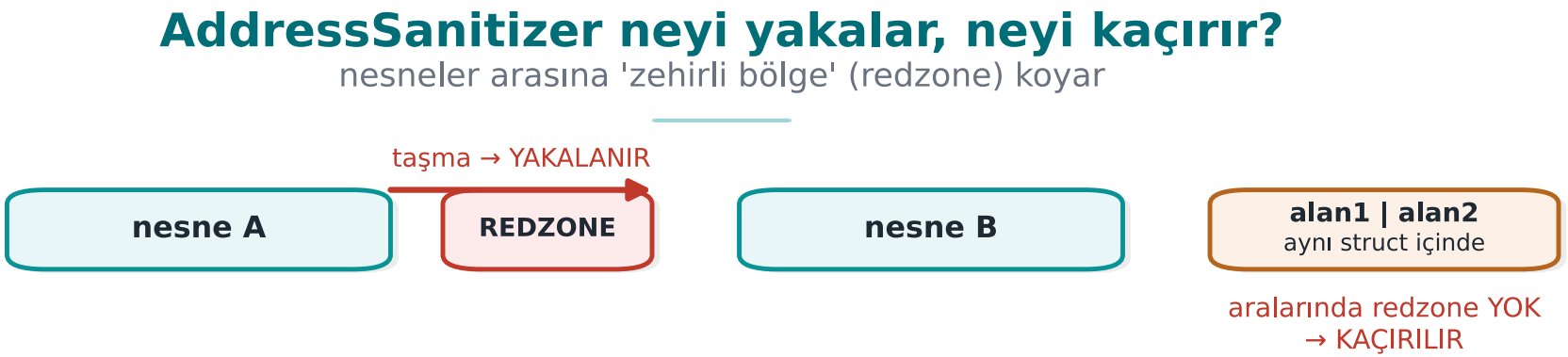
1. Why is `%n` the most dangerous?
2. The three traces in a UAF report? Which one gets the fix?
3. Why can `if (x+100 < x)` be deleted?
4. Why is there no `printf` in a signal handler?

Section 2–3 — Answers

1. `%n` **WRITES** the number of bytes printed so far **to memory** → an attacker can write to an address of their choosing and hijack control flow.
2. **allocated-by · freed-by · use-here.** The fix is usually **between freed↔use**: set the pointer to `NULL` after `free`, or fix the lifetime.
3. Signed overflow is **undefined behaviour**; the compiler assumes "overflow never happens" → the condition is always `false` → it **deletes** it. Do the check with `__builtin_add_overflow`.
4. `printf` is **not async-signal-safe**; if the handler re-enters it, locks/state get corrupted → deadlock/crash. Only safe calls like `write` are allowed.

4. Static Analysis, Sanitizers, Fuzzing

ASan Redzone — Diagram



ASan ayrıca hiç çalışmayan kod yolunu da göremez — fuzzing bu yüzden gerekir.

Static ↔ Dynamic — Diagram

Statik analiz ↔ dinamik analiz

biri diğerinin yerine geçmez — ikisi birden kullanılır

STATİK (çalıştırmadan)
tüm kod yollarına bakar
hiç çalışmayan yolu da görür

yanlış pozitif ÇOK
bağlamı bilmez

DİNAMİK (çalıştırarak)
yalnız yürütülen yol
gerçek hatayı KESİN görür

yanlış pozitif AZ
kapsam girdiye bağlı

Fuzzing üçüncü ayaktır: dinamik analizin göreceği YENİ yolları üretir.

What Is Static Analysis? (Reminder)

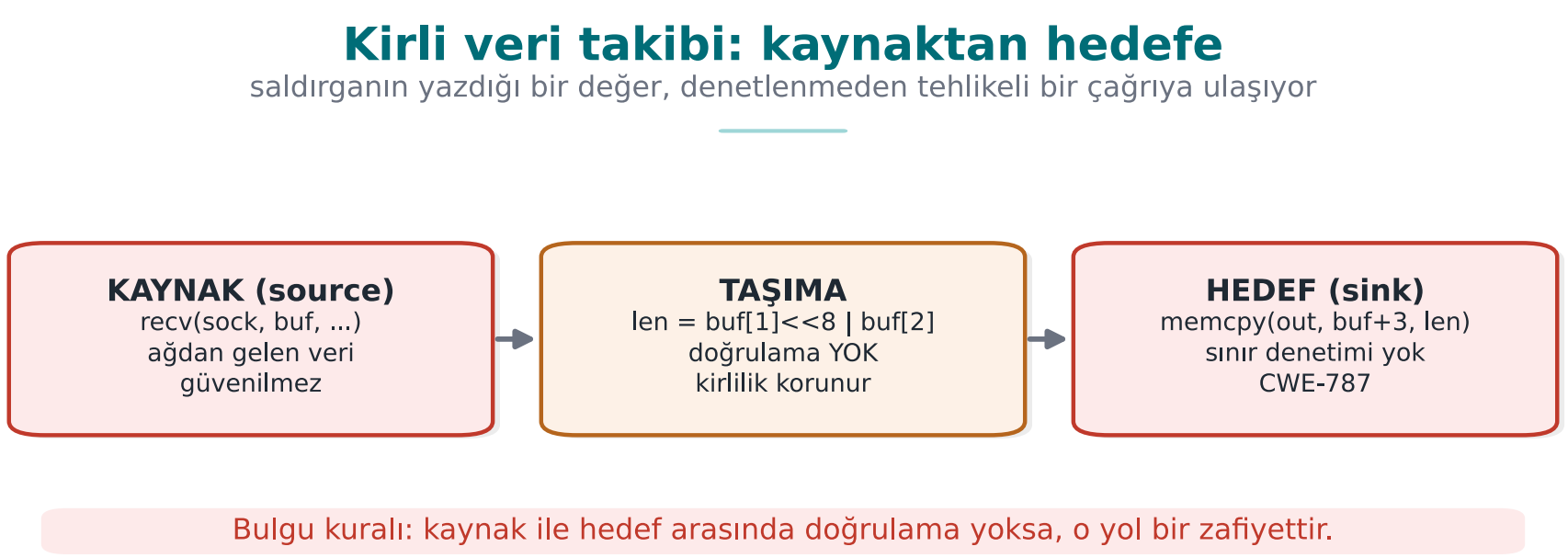
Examining code **without running** it, looking for bugs.

Advantage: it also sees paths that never execute.
Disadvantage: it produces **false alarms**.

Static Analysis · Layer by Layer

Level	Tool	What It Finds
Warnings	`-Wall -Wextra -Wconversion`	Conversion, format
Analyser	`-fanalyzer` , `clang --analyze` , `/analyze`	NULL, leaks
Rule	`clang-tidy cert-*` , `cppcheck`	CERT violation
Semantic	CodeQL, Semgrep	Data flow
Commercial	Coverity, Klocwork	Deep analysis

Tainted Data: Source → Sink



The tool sees **unvalidated** data flowing to a dangerous place.

Living with False Alarms

1. Start with **high-confidence** rules.
2. Every suppression has a **reason**: `// NOLINT(cert-err33-c): neden`.
3. **Zero warnings** on new code (enforced by CI).

> The evaluator asks for the developer's own scan report.

What Is a Sanitizer? (Reminder)

A tool that catches memory/UB bugs while the program runs.

Almost no false alarms; but it only finds bugs on the path that **actually executes**.

→ combine with tests + fuzzing.

The Sanitizer Family

Sanitizer	Finds	Flag
ASan	Overflow, UAF, double free	`-fsanitize=address`
LSan	Leaks	with ASan
UBSan	Overflow, shift, NULL	`-fsanitize=undefined`
MSan	Uninitialized reads	`-fsanitize=memory`
TSan	Data race	`-fsanitize=thread`

How Does ASan Work? · Shadow Memory

- One **shadow byte** per 8 bytes: how much of it is valid?
- A **poisoned region** (redzone) before/after arrays.
- Freed blocks wait in **quarantine**.
- A shadow check before every access.

ASan's Blind Spot

```
struct { char ad[8]; long yetki; } k;
strcpy(k.ad, uzun_girdi);   /* ad → yetki taşar */
```

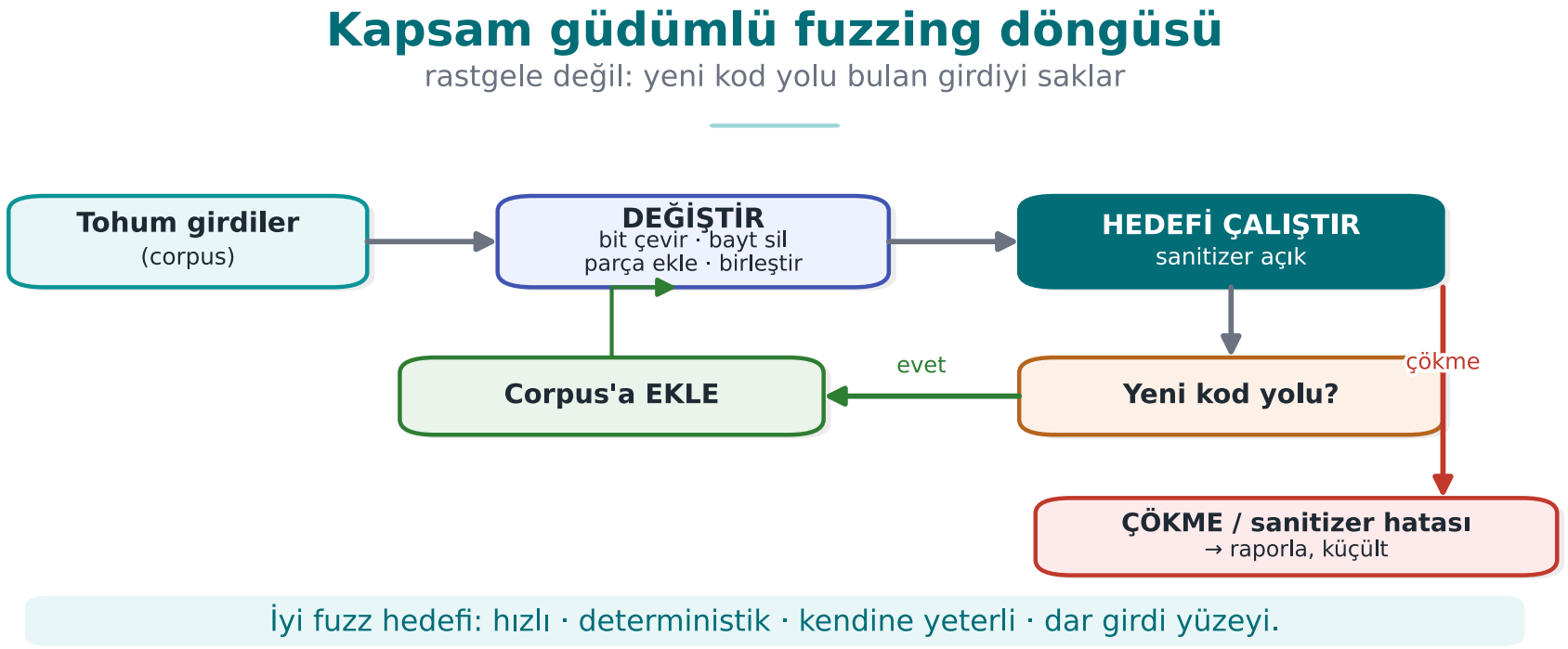
A field-to-field overflow **inside** the same structure → no poisoned region in between → ASan **cannot see it** (Week 1 Demo 3).

Sanitizers Do Not Ship in the Release Build

```
gcc -g -O1 -fno-omit-frame-pointer \
    -fsanitize=address,undefined p.c -o p_test
```

- Slows things down (~2×), uses memory.
- Only in **test/CI**; disabled in the release build.

What Is Fuzzing? · Coverage-Guided



An input that opens a new code path is **saved** and worked on further.

What Does Fuzzing Find?

- It finds magic bytes and the right length field **on its own**.
- Inputs a human would never think of.
- Tools: libFuzzer, AFL++, honggfuzz, OSS-Fuzz (tens of thousands of bugs).

Writing a Fuzz Target

```
int LLVMFuzzerTestOneInput(const uint8_t *veri, size_t boyut) {
    ayristir(veri, boyut);   /* sınanacak fonksiyon */
    return 0;
}
```

```
clang -g -O1 -fsanitize=fuzzer,address fuzz.c ayristir.c -o f
./f corpus/ -max_total_time=10
```

A Good Fuzz Target

Four properties:

- **Fast** (no network/disk)
- **Deterministic** (same input → same result)
- **Stateless** (not affected by the previous call)
- **Single job** (tests one function)

Demo 4 — Finding Bugs with Fuzzing

`code/week-04/04-fuzzing` · CWE-125/787

Step	What Happens?
1	`tohum/normal.bin` is fine
2	libFuzzer ≤ 10 s → a crashing input
3	Replay with ASan → full report
4	Fixed version → no crash
5	`cokerten.bin` is safely rejected

> "This function was fuzzed for this long" = strong evidence (S16).

Section 4 — Quick Check

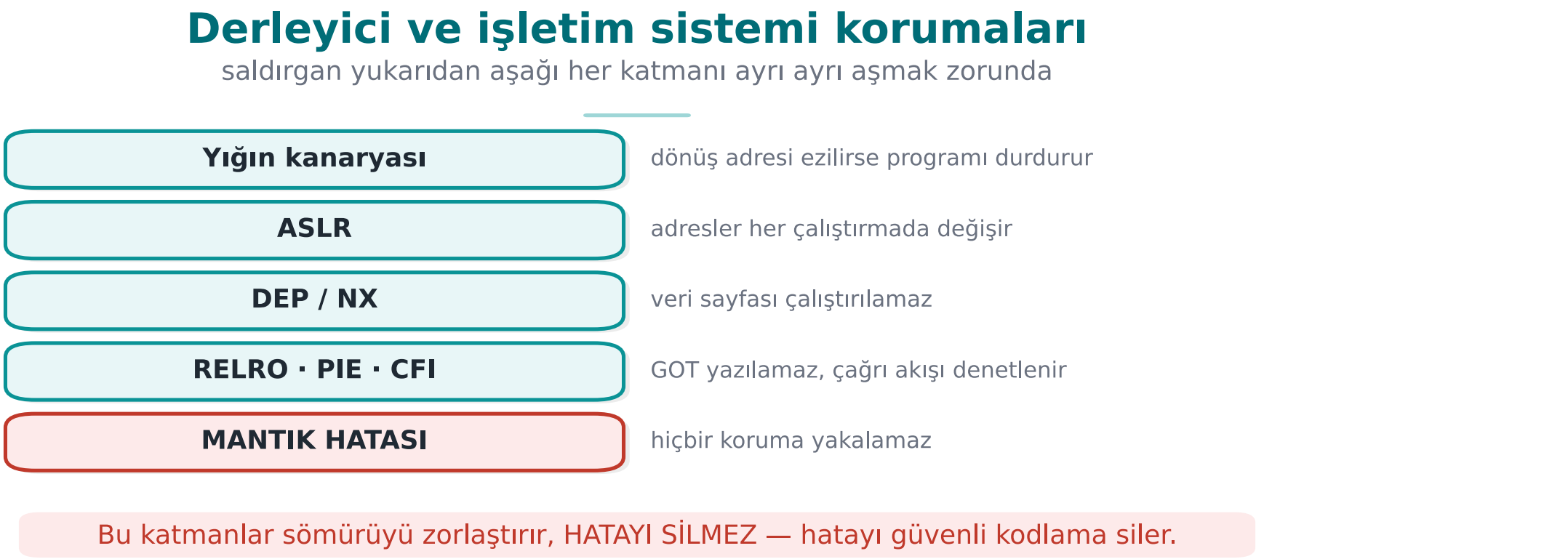
1. The difference between static and dynamic analysis?
2. Which overflow can ASan not see?
3. The four properties of a good fuzz target?

Section 4 — Answers

1. **Static:** examines all paths without running the code (can have false positives). **Dynamic:** while running, observes **real** bugs only on the path that actually executes.
2. ASan sees the places where it placed a **redzone**; it cannot see a **field-to-field** overflow inside an object, or paths that **never execute**.
3. **Fast**, **deterministic**, **self-sufficient** (not dependent on external state), a target that handles a **narrow input surface** and fails cleanly on a crash.

5. Compiler and OS Protections

Compiler/OS Protections — Diagram



Layer 3 · the Idea

Say a bug slipped through.

- There are automatic protections that make exploitation **expensive**.
- Most are turned on with a **compile flag**.
- Not free, but cheap.

Stack Canary

- A **sentinel value** is placed just before the return address.
- If an overflow corrupts this value, it is noticed before the return and the program **halts**.
- `-fstack-protector-strong` / MSVC `/GS`.

Stack Overflow and Canary — Diagram

Durdurmaz: hedefli yazma · öbek (heap) taşması · bilgi sızıntısı.

_FORTIFY_SOURCE

- Catches **library calls** that write more than a buffer's known size.
- `-D_FORTIFY_SOURCE=2` (+ at least `-O1`).
- Does not protect your **own loops**.

PIE + ASLR

- **ASLR:** randomizes addresses.
- **PIE:** makes the program itself load at a random address too.
- `-fPIE -pie` / `/DYNAMICBASE /HIGHENTROPYVA`.

DEP/NX and RELRO

- **DEP/NX:** makes the data region non-executable (default).
- **RELRO:** makes tables like the GOT read-only → `-Wl,-z,relro,-z,now`.

CFI / CFG and the Shadow Stack

- **CFI/CFG:** checks that indirect calls only go to **valid** targets.
- `-fcf-protection`, `-fsanitize=cfi` / `/guard:cf`.
- **Shadow stack:** a second copy of the return addresses (CET).

Protections · One Table

Protection	GCC/Clang	MSVC
Canary	`-fstack-protector-strong`	`/GS`
FORTIFY	`-D_FORTIFY_SOURCE=2`	Secure CRT
PIE/ASLR	`-fPIE -pie`	`/DYNAMICBASE`
NX	default	`/NXCOMPAT`
RELRO	`-Wl,-z,relro,-z,now`	—
CFI	`-fcf-protection`	`/guard:cf`

⚠ Every Protection Has Limits

Protection	Stops	Does Not Stop
Canary	Overflow reaching the return address	Heap overflow, a leaked canary
FORTIFY	Library-call overflow	Your own loops
ASLR	Assuming a fixed address	**Address leak**
NX	Code in the data region	Code reuse (ROP)

The Main Rule

> Every protection makes a **technique** harder; it does not remove a **class of bug**.

None of them is enough for a logic error → secure coding comes first.

Demo 5 — Turning Protections On and Off

`code/week-04/05-derleyici-korumalari` · CWE-121

- The same overflow: `giris_zayif` (off) vs `giris_sert` (on).
- Hardened: `*** stack smashing detected ***` / Windows `0xC0000409`.
- Weak: silent corruption or a crash.

```
readelf -h p | grep Type          # DYN = PIE
readelf -d p | grep BIND_NOW      # tam RELRO
readelf -s p | grep __stack_chk   # kanarya
```

Recommended Release Flags

GCC/Clang:

```
-O2 -D_FORTIFY_SOURCE=2 -fstack-protector-strong -fPIE -pie -Wl,-z,relro,-z,now -fcf-protection -Wall -Wextra -Wformat=2 -Werror=format-security
```

MSVC:

```
/O2 /GS /sdl /guard:cf /DYNAMICBASE /HIGHENTROPYVA /NXCOMPAT /CETCOMPAT /W4
```

Protection Table = Evidence

- The evaluator extracts a **protection table** for every binary and `.so`.
- A disabled protection is a **finding**; its justification must be in the trade-off record.
- Scheme: the application can also **verify itself** that the protections are enabled.

Secure Build Pipeline (CI)

Ucuzdan pahalıya: erken adımlar kolay hataları eler, pahalı adım yalnız kalanlara harcanır.

Every failing step **stops** the merge.

CI · Failure Criteria

Stage	Failure
Build	A single warning
Sanitizer-enabled test	Any report
Fuzzing	A new crash (old corpus = regression)
Binary audit	Log strings, symbols, a disabled protection

The Course's CMake Modes

Mode	Flags
`korumasiz`	`-O0`
`optimize`	`-O2`
`asan`	`-O1 -fsanitize=address`
`denetimli`	`-O2 -D_FORTIFY_SOURCE=2`
`guvenli`	`-O2` + fixed source

Most of the protection flags: Demo 5's `giris_sert`.

The Build Environment Is a Target Too

- Access logging; pinning dependencies (Week 5's SBOM).
- Signed releases; **reproducible** builds.
- If the build machine is compromised, the entire product is affected.

Section 5 — Quick Check

1. What does the canary stop, and what does it not stop?
2. Why is ASLR weakened by an information leak?
3. Why is "no protection is enough for a logic error"?

Section 5 — Answers

1. **Stops:** a **sequential** stack overflow that overwrites the return address (the canary gets corrupted). **Does not stop:** a targeted write, a heap overflow, an information leak.
2. A single **leaked address** reveals the module's base address (the offset is fixed) → the entire randomisation **collapses**.
3. Protections catch **memory** violations; incorrect authorization/business logic is a **valid** memory access → no protection is triggered. Correct design + review is required.

6. Introduction to Code Obfuscation

Layer 4 · Why?

- White-box attacker: reads strings, finds functions by name, bypasses checks.
- Obfuscation: same behaviour, **harder to understand**.

The Purpose of Obfuscation

Not making it impossible, but making it **expensive** (Cookbook 12.1):

- cost > the value being protected
- make automation harder (diversification, Week 14)
- buy **time** for the other layers

> A few simple techniques + data hiding, rather than one advanced technique.

Obfuscation Map

Family	What It Hides	Week
Layout	Names, structure	4
Data	Constants, strings, variables	4, 9
Control flow	Algorithm structure	4, 9
Preventive	Analysis tool	9
Virtualisation	Machine code	9, 14

From the Field: 18 Measures on the Native Side

Symbol visibility off · name mangling · constant arithmetic hiding · string encryption (decrypt on use, **wipe immediately**) · opaque booleans · your own library functions · fake operation/dead branch · flattening + randomised exit · **no** logging in the release build.

➡️ The strength comes from **combining them**, and from RASP.

Three Cheap Measures

Measure	How?
Hide symbols	`static`, `-fvisibility=hidden`, `strip -s`
Remove logging	Empty the macro at compile time
Hide strings	Encrypt at compile time, decrypt on use, wipe immediately

Logging Macro

```
#ifdef GUNLUK_ACIK
#  define GUNLUK(...) fprintf(stderr, __VA_ARGS__)
#else
#  define GUNLUK(...) ((void)0)   /* dizge de çağrı da yok */
#endif
```

`GUNLUK_ACIK` is not defined in the release build → the logging is **never compiled in**.

⚠️ a Runtime Flag Is Not Enough

```
if (hata_ayikla) printf("...");  /* KÖTÜ */
```

- The string **remains** in the binary.
- The attacker can flip the flag and turn logging back on.
- The right way: remove it **at compile time**.

⚠️ String Hiding ≠ Key Storage

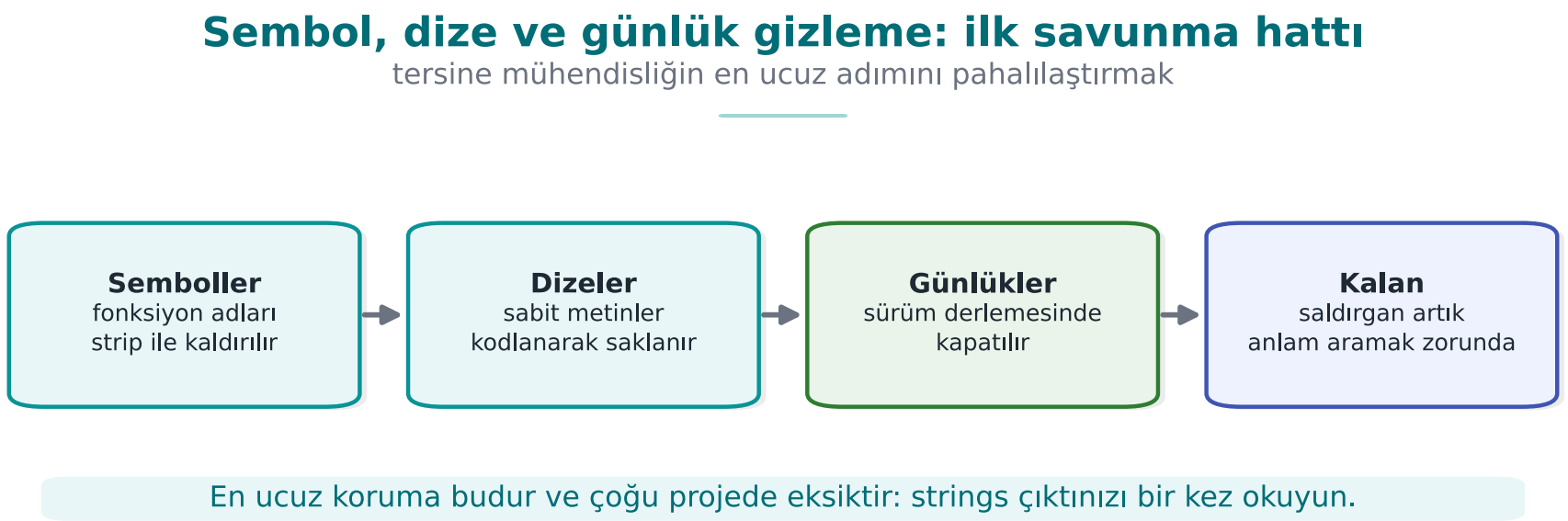
- The decrypted string sits **in memory, in the clear**, while it's in use.
- The decryption key is **inside** the program.
- What it stops: static scans like `strings`.
- Against someone inspecting it while running → RASP (6). A real key → whitebox (11).

Demo 6 — Symbol and String Leakage

`code/week-04/06-sembol-dize` · CWE-200/215

Step	What Happens?
1	`gizli_acik` and `gizli_kapali` give the same result
2 `strings`	The license string and `[LOG]` are visible in the open one; not in the closed one
3 `nm`	`lisans_dogrula` is visible in the open one; no symbol in the closed one
4	Windows: names are in the PDB; the closed build produces no PDB

Symbol and String Hiding — Diagram

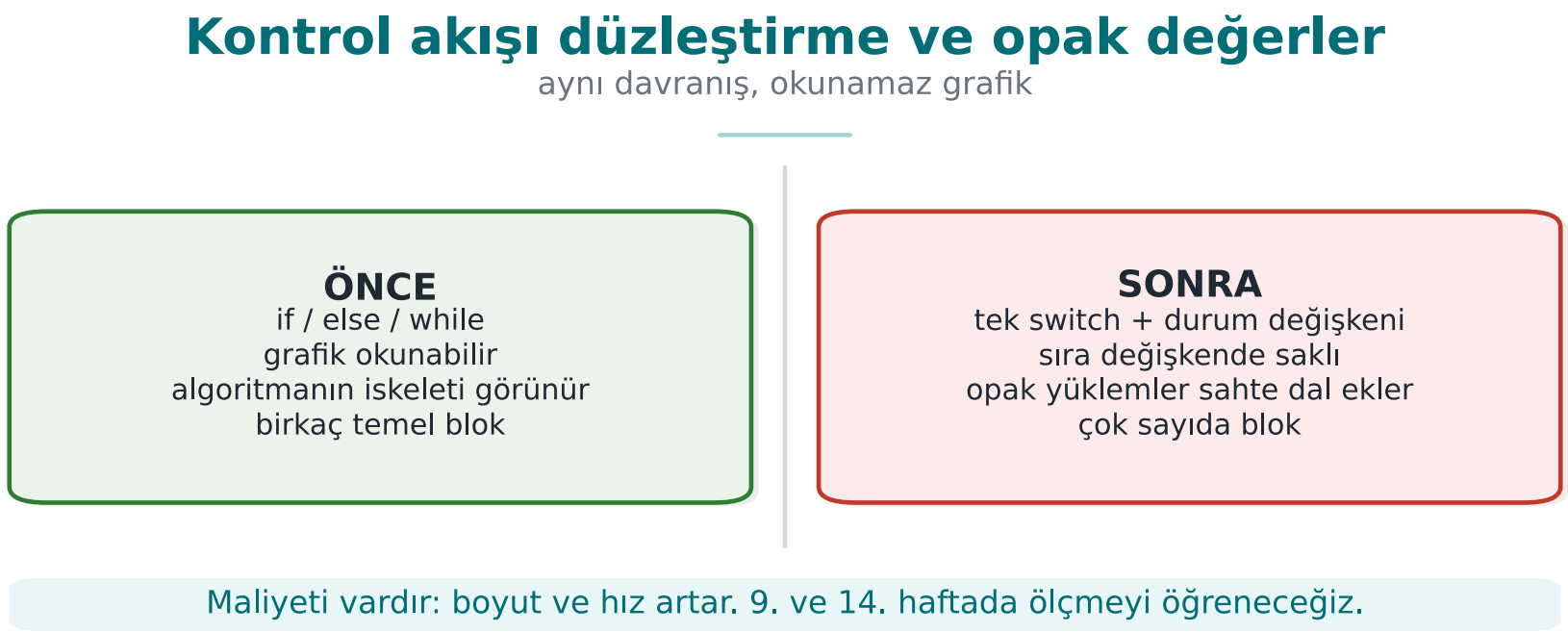


Control-Flow Flattening

```
int durum = 1;
for (;;) switch (durum) {
  case 1: durum = 2; break;
  case 2: durum = (kosul ? 3 : 4); break;
  case 3: durum = 5; break;
  case 4: return BASARISIZ;
  case 5: return BASARILI;
}
```

The blocks' natural adjacency is lost; the order lives only in the **state variable**.

Flattening: Before/After — Diagram



What Strengthens Flattening

- **Hide the state values** (arithmetic transformation)
- **Fake/dead blocks** (which one is real?)
- **Randomised exit** (where the error was caught becomes unreadable)
- **Opaque values** (a single byte cannot flip it)

The template alone is solved quickly on its own → Week 9 depth.

Demo 7 — Flattening

`code/week-04/07-akis-duzlestirme`

Step	What Happens?
1	`pin.c` and `pin_duz.c` give the same result (the PIN is synthetic)
2	`objdump` : the flattened version has far more branches
3	Timing: the cost of flattening

➡️ Automated with Tigress's `Flatten` in Week 14; measuring effectiveness in Week 9.

Concept: Three Techniques Leading into Week 9

- **Function name hiding:** exported names are meaningless; readability handled via a macro.
- **Allocation-size hiding:** blur the "32 bytes = a key" tell.
- **Dynamic decryption:** sensitive code sits encrypted in the binary; conflicts with DEP/NX, used selectively.

Section 6 — Quick Check

1. Does obfuscation fix the bug?
2. Why isn't silencing logging with a runtime flag enough?
3. Three techniques that strengthen flattening?

Section 6 — Answers

1. **No.** Obfuscation only makes reverse engineering harder; the bug **stays where it is**. **Fix first, then obfuscate.**
2. The string/code **remains** in the binary; the attacker can flip the flag or read the strings. Remove logging in the release build **at compile time** (macro/dead-code elimination).
3. **Opaque predicates** + **fake blocks/dead branches** + **state-variable encoding & randomised exit** (also name/string hiding).

7. Project and Closing

Project · S9 "Code Hardening" — 1

- [ ] Build with the recommended release flags; attach the protection table (`checksec` / `dumpbin`).
- [ ] Run a CERT scan with `clang-tidy` / `cppcheck`; fix ≥ 5 findings (with rule ids).

Project · S9 — 2

- [ ] Run the tests with ASan + UBSan.
- [ ] At least one fuzz target, ≥ 10 min.
- [ ] No logging in the release build; no sensitive string in `strings` output.

Measure card: Description → Implementation → Verification → Residual risk.

Solved Self-Check

Question 1

Which of the three layers actually removes the bug?

Answer: Only **secure coding** (layer 2). Compiler/OS protections make exploitation harder; obfuscation slows down reading — neither one **removes** the bug.

Question 2

What is the mildest outcome of `printf(girdi)`?

Answer: An **information leak** (reading the stack via `%x` / `%p`). The worst is writing to memory via `%n`.

Question 3

The three stack traces in a UAF report? Which one gets the fix?

Answer: Use, free, allocation. The fix is usually at the **free** point (an ownership decision).

Question 4

Why can `if (x + 100 < x)` be deleted?

Answer: Signed integer overflow is **undefined behaviour**; the compiler assumes "overflow never happens" and can optimise the check away, **deleting** it.

Question 5

Why is there no `printf` in a signal handler?

Answer: `printf` / `malloc` are **not async-signal-safe**; if the handler calls them while interrupted mid-state, corruption follows. Only a flag is set (SIG30-C).

Question 6

Which overflow can ASan not see?

Answer: A **field-to-field** overflow inside the same structure (there is no poisoned region between them).

Question 7

The four properties of a good fuzz target?

Answer: Fast, deterministic, stateless, single job.

Question 8

What does the canary stop, and what does it not stop?

Answer: Stops: a stack overflow that reaches the return address. **Does not stop:** heap overflow, corruption of earlier variables, a leaked canary.

Question 9

Why is ASLR weakened by an information leak?

Answer: If one address leaks, the attacker can **calculate** the other addresses relative to it; the advantage of randomisation is lost.

Question 10

Why isn't silencing logging with a runtime flag enough?

Answer: The string **remains** in the binary and the attacker can flip the flag to turn logging back on. It must be removed **at compile time**.

Question 11

Three techniques that strengthen flattening?

Answer: Hiding state values, fake/dead blocks, a randomised exit point (and opaque values).

Question 12

Why is string hiding not the same as key storage?

Answer: The decrypted string is in memory, in the clear, while it's in use, and the decryption key sits inside the program; it only stops **static** scanning.

Summary: This Week in One Sentence

> First **bug-free code** (CERT + sanitizer + fuzzing), then **limit the damage** (compiler/OS protections), and finally **make it harder to read** (obfuscation) — and no protection substitutes for secure coding.

Next Week

Week 5 — Code Hardening: Java and Interpreted Languages

Injection (SQL, command, path), obfuscation with ProGuard/R8, dependency security and SBOM.

Preparation: JDK 17+, Maven · `code/week-05`.

Appendix A · A Vulnerability, Start to Finish

Buggy Code

```
void selamla(const char *ad) {
    char tampon[16];
    strcpy(tampon, ad);          /* sınır yok */
    printf("Merhaba %s\n", tampon);
}
```

What happens if `ad` is longer than 16 bytes?

What Happens? · Step by Step

- `tampon` is 16 bytes on the stack.
- `strcpy` copies `ad` all the way to its end, paying no attention to bounds.
- An `ad` longer than 16 → neighbouring memory (including the return address) is corrupted.

Attack · Outcome

- Short overflow: a neighbouring variable gets corrupted (a logic bug).
- Long overflow: the **return address** gets corrupted → crash or control hijacking.
- Without protection: a serious vulnerability (CWE-121).

Fix 1 · Bounded Copying

```c
int n = snprintf(tampon, sizeof tampon, "%s", ad);
if (n < 0 || (size_t)n >= sizeof tampon) {
    /* ad kesildi: reddet ya da işaretle */
}
```

`snprintf` knows the size and **reports truncation**.

Fix 2 · Layers

- **Coding:** `snprintf` (STR31-C).
- **Compiler:** `-fstack-protector-strong` (canary).
- **FORTIFY:** `-D_FORTIFY_SOURCE=2` checks the library call.
- **Testing:** ASan catches the overflow with a long input.

Lesson: One Line, Many Layers

- One bug: `strcpy`.
- Defence: the right function + compiler protection + sanitizer testing.
- But the **real** fix is in the first line: bounded copying.

Appendix B · Common Mistakes

Mistake · Ignoring Warnings

- Something dismissed as "just a warning" is tomorrow's vulnerability.
- Turn the warning into an error with `-Werror`.

Mistake · Not Checking the Return Value

- `malloc`, `fread`, `RAND_bytes` can return NULL/an error.
- If not checked: a silent disaster (ERR33-C).

Mistake · Shipping the Sanitizer in the Release Build

- Sanitizers are for test/CI; they are slow and leak information.
- Disabled in the release build.

Mistake · Hiding UB with `-fwrapv`

- Making the overflow "defined" **hides** the logic error.
- Use UBSan to diagnose it, then fix it.

Mistake · Covering Up a Bug with Obfuscation

- Obfuscation does not fix the bug, it only **hides** it.
- Fix first, then obfuscate.

Mistake · "Cleaning Up" Input and Using It Anyway

- Reject dangerous input instead of trying to **fix** it.
- Allow-list + rejection is the safest approach.

Appendix C · Quick Reference

Glossary

Term	Meaning
UB	Undefined behaviour
UAF	Use-after-free
CERT/CWE	Rulebook / weakness catalogue
Sanitizer	Runtime bug detector
Fuzzing	Finding crashes with random input
Canary/ASLR/NX	Compiler/OS protections

Hardening Checklist

- [ ] `-Werror` and the CERT scan are clean
- [ ] Every return value is checked
- [ ] Tests pass with ASan + UBSan
- [ ] At least one fuzz target has run
- [ ] Release flags are on (protection table)
- [ ] No logging/sensitive strings in the release build

Final Word (Week 4)

> Secure code is not written once; it is **checked with tools on every build**.

First correct code, then protection, finally obfuscation. The order matters.