



Runtime Application Self-Protection (RASP)

CEN429 Secure Programming — Week 6

Asst. Prof. Dr. Uğur CORUH · 23.10.2026

Today's Plan (3 Hours)

Hour	Section	Topic
1	0–2	Basic concepts · what RASP is · RASP architecture
2	3–7	Integrity · debugger · emulator · hook/Frida · memory protection
3	8–13	Root/signature · control-flow integrity · response policy · limits · project

A Brief History — the Application Protects Itself

- **1980s** — the crack/anti-crack culture: the origin of **anti-debug** and self-checks
- **2000s** — DRM and mobile banking move protection **inside** the application
- **2012** — Gartner coins the term **RASP**
- **today** — **OWASP MASVS-RESILIENCE** turns this into an auditable requirement

RASP is not a new idea: it is the name of the corporate answer to the **MATE attacker**.

Where Does This Week Fit?

- **Weeks 4–5:** we hardened the code **statically** (compilation, obfuscation).
- **This week (6):** the program protects itself **while it runs** → RASP.
- **Weeks 9/11/14:** obfuscation and whitebox crypto work together with RASP.

Learning Outcome

This week is about **LO.3**.

By the end, you will be able to:

- Recognise RASP checks (integrity, anti-debug, environment, hooking)
- Design a response policy and device binding
- Explain RASP's **limits** and its ethical framework

Core Idea

RASP = **Runtime Application Self-Protection**: the application **monitoring** itself while it runs and **responding** to threats.

Detect → defend → deter. But never a single check — a **layer**.



0. Basic Concepts (From Scratch)

What Is Runtime?

- **Runtime:** the moment the program is **running** (not compilation).
- RASP protections kick in here: the program monitors itself **while running**.

What Is RASP?

- **RASP (Runtime Application Self-Protection)**: the application protecting itself while it runs.
- It **detects** threats such as tampering, debugging, and a fake environment, and **responds**.

Debugger

- **Debugger:** a tool that runs a program step by step, halts it, and reads memory (gdb, lldb).
- The attacker uses it to trace the flow and change values.
- RASP checks "is a debugger attached to me?"

Emulator and Virtual Machine

- **Emulator/VM:** a software environment that **imitates** a device (Android emulator, QEMU).
- The attacker does their analysis here instead of on a real device (easier).
- RASP tries to sense the fake environment.

Hook and Instrumentation

- **Hook:** intercepting a function call and **changing** it.
- **Instrumentation:** injecting code into a running program to observe/change its behaviour.
- Tool: **Frida** (a very common dynamic instrumentation tool).

LD_PRELOAD

- **LD_PRELOAD:** a way on Linux to load a library **before** the program and replace functions.
- The attacker can use it to **hook** critical functions.
- RASP tries to detect this.

Integrity and Self-Hashing

- **Integrity:** the code/file being **unchanged**.
- **Self-hashing:** the program computing a digest of **its own code** and comparing it against the expected value.
- If it's been tampered with, the digest won't match.

Self-Hashing — Diagram

Bütünlük denetimi (self-hashing) uygulama kendi baytlarını doğrular



Tek başına yetmez: denetimin kendisi de yamalanabilir → örtüşen denetim gerekir.

Digest (Checksum/Hash)

- **Digest:** a fixed-size **fingerprint** computed from data (SHA-256).
- If the data changes, the digest changes.
- The foundation of integrity checking.

Root / Jailbreak

- **Root:** full privilege on the device (normally restricted).
- On a rooted device, protections weaken; the attacker can access everything.
- RASP checks "is the device rooted?"

Signature Verification

- Application packages are signed with a **digital signature**.
- **Signature verification:** checking whether the caller's/loaded component's signature is the expected one.
- Catches a fake/modified component.

Control-Flow Integrity (Counter)

- If critical checks are done with **a single `if`**, one patch at that single point skips them.
- **Control-flow counter**: verifying that checks passed in the correct **order**, by counting them.
- A single patch is no longer enough.

Response Policy

- **Response policy:** what will RASP **do** when it sees a threat?
- Shut down silently, restrict the feature, notify the server, respond with a delay...
- "Crash immediately" is not always the best choice.

Device Binding and Deterrence

- **Device binding:** data/keys being meaningful only on a **specific device**.
- **Deterrence:** making the attack costly/risky enough that the attacker gives up.
- RASP's ultimate goal: raising the cost.

Now We're Ready

Terms:

runtime · RASP · debugger · emulator/VM · hook/Frida · LD_PRELOAD · integrity/self-hashing · digest · root · signature verification · control-flow counter · response policy · device binding · deterrence

Now: what is RASP, and what does it do?



1. What Is RASP?

WAF ↔ RASP — Diagram

WAF ile RASP farkı

'kurcalandım mı, beni kim izliyor, hangi cihazdayım?'

WAF

uygulamanın ÖNÜNDE
HTTP isteklerini süzer

uygulamanın içini bilmez
sunucu tarafı

RASP

uygulamanın İÇİNDE
kendi kodunu, belleğini,
ortamını izler

MATE saldırganına cevap

RASP, cihazın güvenilmez olduğu durumlar için tasarlanmıştır.

WAF ↔ RASP Comparison

Protection	Where It Stops	What It Sees
Network firewall	At the network edge	Packets
WAF (Web App Firewall)	In front of the application	HTTP requests
RASP	Inside the application	Its own code, memory, environment

A WAF asks "is the incoming request malicious?"; RASP asks "have **I** been tampered with?"

The Attacker Model: MATE

- **MATE (Man-At-The-End):** an attacker who **owns the endpoint the application runs on**. They read memory, modify code, and trace it with a debugger. Identical to the **white-box** model from Week 11.
- **The hard truth:** if the device's owner is the attacker, **no client-side protection is ultimately unbreakable**.
- So why does RASP exist? The goal is not **unbreakability**; it's making the attack **unscalable and expensive**.

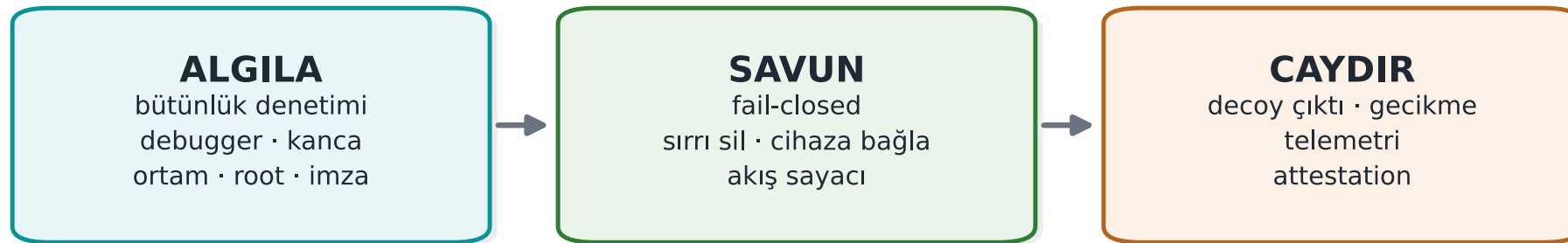
Three Jobs: Detect, Defend, Deter

- **Detect:** sense the abnormal environment/behaviour (debugger, emulator, tampering).
- **Defend:** if there's a threat, restrict/halt the feature.
- **Deter:** make the attack costly → the attacker gives up.

The RASP Loop — Detect → Defend → Deter

RASP döngüsü: algıla → savun → caydır

koruma uygulamanın İÇİNDE çalışır



Tek denetim değil, birbirini kontrol eden ÖRTÜŞEN AĞ → tek yama yetmez.

What Does RASP Complete?

- Static protections (Weeks 4–5) apply **without running**.
- Obfuscation (Weeks 9/11/14) makes **understanding** harder.
- RASP adds active defence **while the program runs**.

Obfuscation hides RASP; RASP protects obfuscation while it runs.

Why Is RASP Necessary?

- A white-box attacker **runs** the program and observes it (debugger, Frida).
- Static obfuscation alone does not stop this.
- RASP checks "am I being watched?" and responds.

The Main Rule (Up Front)

RASP is not unbreakable on its own; it gains strength from **layered** and **varied** checks.

A single "is it rooted?" check is easily bypassed; dozens of different checks together are hard.



2. RASP Architecture

When Does a Check Run?

- **At startup:** environment check on launch.
- **Right before a critical operation:** immediately before payment/key use.
- **Periodically:** regularly, in the background.
- **Randomly:** at unpredictable moments (so the attacker can't time it).

Where Does a Check Run?

- **On the native side** (C/C++): harder to reverse-engineer, preferred.
- **On the managed side** (Java): easier to read, but still necessary.
- **Cross-checking**: the two sides check each other.

The Cross-Check Idea

- The Java side checks the native library's **digest**.
- The native side checks the Java side's **integrity**.
- The attacker has to bypass **both at once**.

Multi-Point Placement and Decoy Parameters

- **Multi-point placement:** the same check done in **many places** in the code, each time in a slightly different form; when one point is patched, the others still run.
- **Inside the protected function itself:** a valuable function (e.g., key derivation) performs **several checks of its own** at its entry point — in the field this can reach **close to fifteen** checks.
- **Decoy parameters:** at an entry point, there can be extra parameters, hidden for readability, that trigger an integrity check (Week 4).

How Does a Check Run? (Hidden)

- The check code is **obfuscated** (Week 9): so the attacker can't easily find it.
- The result is **opaque** (not plain true/false).
- The response can be **delayed** (more on this shortly).

Weak Design — a Single Decision Point

```
if (hata_ayiklayici_var() || root_var() || butunluk_bozuk())  
    return HATA;          /* bu tek satırı değiştiren  
                           her şeyi atlatır */  
anahtari_coz();
```

An attacker who flips a single comparison bypasses **the entire protection**.

Strong Design — Data Dependency

The check result feeds into **the data of the next computation**, not into an `if`:

- Check results feed into **key derivation** (the correct key if clean, a different but plausible-looking key if not).
- Check results are carried with **opaque values** (Week 4).
- The response happens **far in time and code** from the trigger.

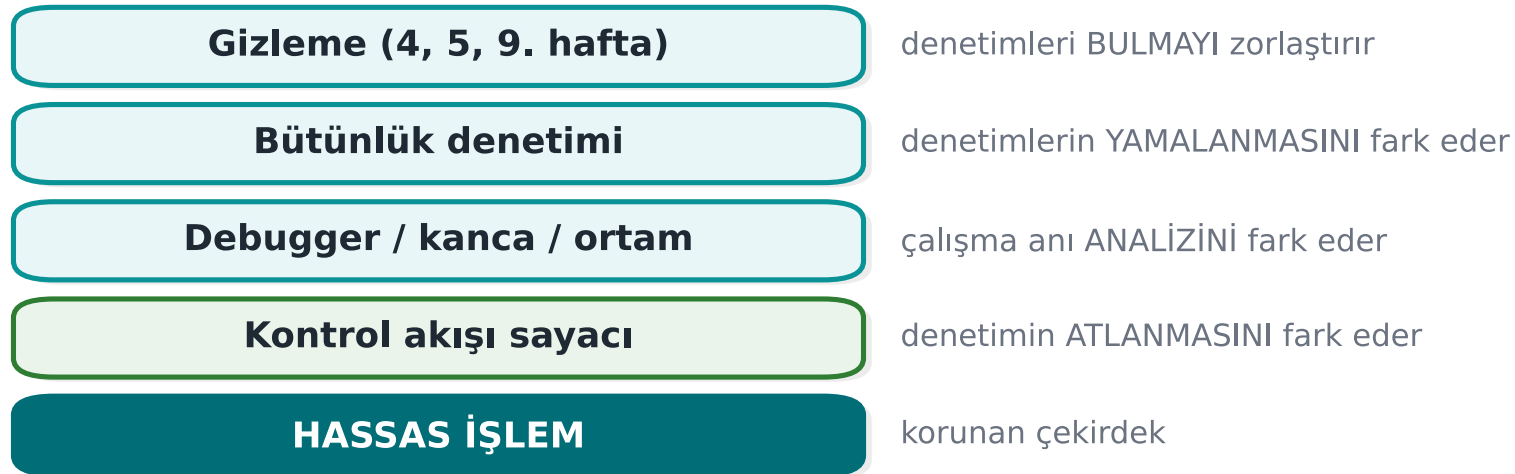
A Layered Network of Checks

- Not a single check, but **dozens** of different checks.
- A **network** that checks itself (a checker network).
- Bypassing one triggers another.

Layers Protect Each Other — Diagram

Katmanlar birbirini korur

aynı hassas işlemi saran katmanlar

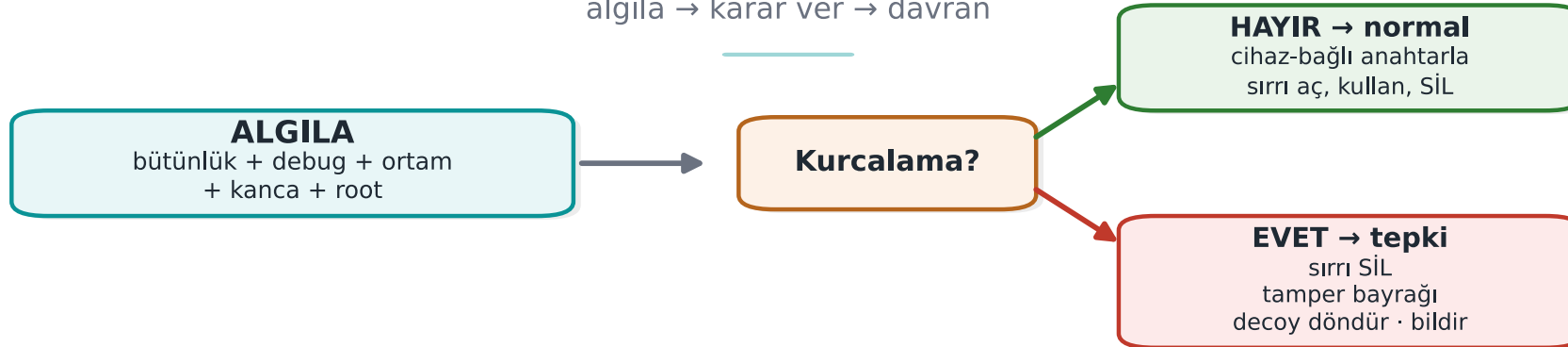


Saldırganın hepsini aynı anda aşması gerekir; bu da saldırı potansiyelini yükseltir.

Architecture · Summary Diagram

RASP motoru: her kritik işlemde

algıla → karar ver → davran



Tepki gecikmeli/sessiz olmalı: hemen çökmek saldırgana denetimin yerini gösterir.

This Section's Rule (1–2)

Do not do the check **in one place, at one moment, with one if** .

Multi-point, cross-checked, hidden, and data-dependent checks **together** build a RASP architecture.

Section 1–2 — Quick Check

1. RASP's three jobs?
2. Why is cross-checking strong?
3. Why a network instead of a single check?

Section 1–2 — Answers

1. **Detect** (tampering/debugging/emulator), **Defend** (protect the key/data, restrict the feature), **Deter** (response: shut down/break/notify).
2. Checks **watch each other**; when the attacker disables one, another notices → silencing a single point isn't enough.
3. A single check is bypassed with **a single patch**; an overlapping network requires breaking every node separately → it raises **the cost of the attack**.



3. Integrity Checking (Self-Hashing)

The Idea

- The program computes a digest of **its own code sections**.
- It compares it against the expected digest.
- If it's been tampered with, the digest **won't match** → response.

How Does It Work? · Step by Step

1. After compilation, the digest of the critical code blocks is **recorded**.
2. While running, the program **recomputes** the digest of the same blocks.
3. It compares it against the recorded digest.
4. If there's a difference → tampering → response.

Step 1 · Record the Digest

- **After** compilation finishes, the critical code/data region is identified.
- This region's digest is computed and stored as the **golden value**.
- This value is the signature of the "untampered" state.

Step 2 · Recompute While Running

- **While running**, the program **recomputes** the digest of the same recorded code/data region.
- This computation happens not at compile time, but at **runtime**.
- It can be done not just once, but **periodically**.

Step 3 · Compare

- The newly computed digest is compared against the recorded **golden value**.
- The comparison must be done in **constant time** (not `memcmp`): a plain comparison stops at the first differing byte and leaks timing (Week 3).
- If equal: integrity is **fine**.

Step 4 · Respond

- If the digest **doesn't match** → tampering (a patch) has been detected.
- The result is tied not directly to an `if`, but to a **response** (Section 10).
- A smarter response is preferred over "crash immediately."

What Does It Catch?

- A **patch** applied to the binary file.
- Bytes changed to skip a check.
- Injected code.

Demo 1 · Runtime Integrity Checking

`code/week-06/01-butunluk-hmac` — self-hashing, HMAC-SHA-256.

- The textbook (Viega & Messier, Recipe 12.2) does this with **CRC32**; **CRC32 is not cryptographic**, an attacker can easily adjust bytes to make the CRC match. We use **HMAC-SHA-256** (needs a secret key).
- The program computes its own binary file's digest and records it as the **golden value**.
- A **copy** of the binary file is made and one byte in it is changed (a patch simulation); the patched copy's digest is compared against the golden value.

Demo 1 · The Core of the Code

```
/* Kendi yolunu bul, dosyayı oku, HMAC hesapla */
oz_yol(yol, sizeof(yol));
dosya_oku(yol, &veri, &boy);
kripto_hmac_sha256(RASP_ANAHTAR, 32, veri, boy, ozet);

/* sabit zamanli karsilastirma */
unsigned char fark = 0;
for (int i = 0; i < 32; i++)
    fark |= beklenen[i] ^ ozet[i];
if (fark == 0) { /* TAMAM */ } else { /* YAMA */ }
```

Demo 1 · Actual Output

```
ADIM 2 - Normal calisma: butunluk dogrulanir (yama yok)
Beklenen   : bae498a6a982279a8...ee4d01e6
Hesaplanan : bae498a6a982279a8...ee4d01e6
SONUC: BUTUNLUK TAMAM - hedef degismemis.
```

```
ADIM 3 - Saldiri: kopyanin 1 bayti degistiriliyor
ADIM 4 - Yamali kopyanin HMAC'i karsilastiriliyor
Beklenen   : bae498a6a982279a8...ee4d01e6
Hesaplanan : 05fb42748cb5b573...131c6705
SONUC: YAMA ALGILANDI - hedef degistirilmis!
```

Changing a single byte completely changed the digest (the avalanche effect).

Self-Hashing Alone Is Not Enough

- The attacker can find the **digest function** and force it to always say "match."
- Or bypass the check with hardware breakpoints (van Oorschot et al.).
- So: hide it, diversify it, cross-check it.

Strengthening It

- The digest-checking code is **hidden** (Week 9).
- **Multiple** blocks, overlapping regions.
- The result is not a plain comparison, it **feeds into a computation** (the check result affects the program's behaviour).

Overlapping Checks

- One check also covers another check's code.
- When the attacker disables one, the other **breaks**.
- Bypassing from a single point becomes hard.

This Section's Rule (3)

Verify the integrity of critical code/data regions at runtime with a **cryptographic** digest (HMAC/signature), not CRC.

Don't do the check at a single point and only at startup; make it **periodic, multiple, and overlapping**.



4. Debugger Detection

Anti-Debug Approaches — Diagram

Hata ayıklayıcı algılama yolları

hiçbiri tek başına kesin değil — sinyaller birlikte tartılır

İşletim sistemi bayrağı

Linux /proc/self/status TracerPid · Windows IsDebuggerPresent

PEB / NtGlobalFlag

Windows süreç yapısındaki izler

Zamanlama

iki nokta arası süre eşiği aşıyor mu?

Kendi kendini izleme

ptrace(PTRACE_TRACEME) başarısızsa biri izliyor

Her sinyal atlatılabilir; güç, birden çok bağımsız sinyalin birlikteliğinden gelir.

Why Is a Debugger Dangerous?

- The attacker **halts** the program, reads memory, changes values.
- They can bypass checks one by one.
- RASP checks "is a debugger attached to me?"

Detection Approaches (Concept)

- Asking the operating system for "am I being traced?" information.
- The state of specific debugging interfaces.
- Timing: if an operation took **much longer** than expected, someone may be single-stepping it.

Indicators by Platform · What's Outdated

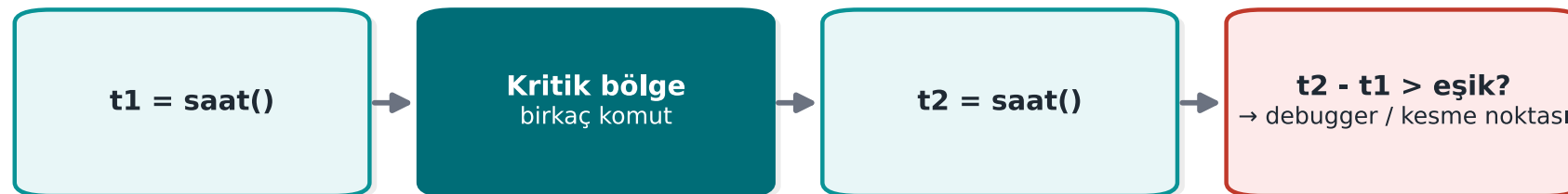
Environment	Indicator
Linux / WSL	<code>/proc/self/status</code> → <code>TracerPid</code> (if > 0, we're being traced); is the parent process <code>gdb/strace</code>
Windows	<code>IsDebuggerPresent</code> (PEB <code>BeingDebugged</code>); <code>CheckRemoteDebuggerPresent</code> ; PEB <code>NtGlobalFlag</code>

From 2003 to today: SoftICE is dead, replaced by `x64dbg/WinDbg/gdb/lldb/Frida`; `ptrace` + **TracerPid** is still valid.

Timing-Based Detection

Zamanlama tabanlı sezme

debugger kodu yavaşlatır



Yanlış pozitif riski: yavaş cihaz, arka plan yükü. Eşiği ölçerek seçin.

Debugger single-stepping **slows down** the work.

Demo 2 · Debugger Detection

`code/week-06/02-hata-ayiklayici` — anti-debug, read-only.

- The program looks at several independent signals and **changes nothing**.
- It doesn't even self-`ptrace`; it only **reads** `/proc` and PEB fields.
- `demo.sh` runs the program first normally, then under `gdb`.

```
/* /proc/self/status icinden TracerPid oku */  
long tp = tracer_pid();  
if (tp > 0) supheli++; /* bir surec bizi izliyor */
```

Demo 2 · Actual Output (WSL)

```
ADIM 1 - Normal calisma (hata ayiklayici YOK)
1) /proc/self/status TracerPid : 0 (izleyen yok)
2) Ana surec (parent) adi      : sh
SONUC: Temiz - izleyen bir arac gorunmuyor.

ADIM 2 - gdb altinda calistir
1) /proc/self/status TracerPid : 2909 (IZLENIYOR)
SONUC: Hata ayiklayici/izleme ARACI algilandi (2 sinyal).
```

Without a debugger, TracerPid is **0**; under gdb it came out **2909** (>0).



The Detection–Counter-Detection Race

Anti-debug is a classic **arms race**: the attacker can turn `ptrace` into a no-op with `LD_PRELOAD`, patch `IsDebuggerPresent`'s return value, or clear the PEB flag.

- Every single method is known and **can be bypassed**.
- Strength: **many** different methods + obfuscation + diversification.
- The goal is still **deterrence** and **delay**.

What About the Response?

- Crashing immediately is not always good (it tells the attacker "there's a check here").
- Better: a **delayed, indirect** response (Section 10, shortly).

This Section's Rule (4)

Don't write anti-debug as **a single** `if (IsDebuggerPresent()) exit();` .

Collect multiple independent signals, tie the result to a behaviour, **delay** the response.

Section 3–4 — Quick Check

1. What does self-hashing catch?
2. Why isn't it enough alone?
3. How does timing give away a debugger?

Section 3–4 — Answers

1. **Code/binary bytes being changed** (a patch, breakpoint/code injection) — it computes its own digest at runtime and compares it against the expected one.
2. The attacker can find/bypass the hash routine or the expected value, or NOP out the comparison; it also only catches a **static** change. Overlapping checks are needed.
3. A debugger/breakpoint **slows down** the code; if the elapsed time between two points crosses a threshold, debugging is sensed.



5. Environment Detection: Emulator/VM

Emulator False Positive — Diagram

Emülatör algılamada yanlış pozitif

suçsuz kullanıcıyı engellemek en pahalı hatadır

Zayıf sinyaller

pil yok · sensör yok
hipervizör biti

WSL2 / Hyper-V / VBS yüzünden
MEŞRU makinede de görülür

Doğru yaklaşım

birden çok sinyali TART
sertliği derecelendir

sunucu tarafı attestation
ikili 'çalış/çalışma' değil

Bir skor üretin; eşiği ürünün risk iştahına göre ayarlayın.

Why an Emulator?

- The attacker does their analysis **in an emulator** instead of a real device.
- Halting, tracing, and resetting is easy in an emulator.
- RASP checks "am I on a real device?"

Detection Clues (Concept)

- Hardware/sensor gaps specific to emulators.
- Typical fake identifiers.
- The absence of features expected on a real device.

Detection Techniques (Concrete)

Technique	How	Note
CPUID hypervisor bit	<code>CPUID.1:ECX[31]</code> — "hypervisor present"	0 on bare metal
Hypervisor vendor signature	CPUID leaf <code>0x40000000</code> → 12 characters	Some hypervisors hide it
Timing	Measure an operation; heavy emulation slows it down a lot	Threshold varies by machine

Demo 3 · VM/Emulator and Timing

`code/week-06/03-ortam-zamanlama` — only reads a CPU instruction and the clock.

```
(A) CPUID.1:ECX[31] hypervisor biti : VAR
    Hypervisor satıcı imzası       : (gizli/bos)
(B) 2.000.000 işlem süresi       : 0.806 ms
SONUC: Hypervisor GORULDU.
```

This output was captured on a **real** WSL2 machine.

The Most Important Lesson: False Positives

- The output above said the hypervisor **"IS PRESENT."**
- But this is **not** an analysis environment — it's an ordinary developer machine!
- On modern Windows, most machines report a hypervisor because of **Hyper-V, WSL2, VBS**.

The Cost of a False Positive

- This bit alone does **not** mean "an attacker is analysing."
- If an application **refused** to run just because it saw a hypervisor, it would block millions of **legitimate** users.
- Legitimate users: developers, enterprise VM users, ordinary Windows users with WSL2/Hyper-V/VBS.
- RASP always **weighs multiple indicators** and accepts that any single one **can be bypassed**.

Rule: a Risk Score, Not a Binary Decision

- Use environment detection as a **risk score**.
- **Combine** multiple indicators.
- Make the final decision together with a **server-side risk engine**.

How Is This Applied in the Field?

- In the mobile world the equivalent is **emulator detection**: `ro.kernel.qemu`, fake sensors, typical IMEI/serial values.
- The same principle applies: a signal, **not proof**.
- Combination + server-side verification.

This Section's Rule (5)

In emulator/VM detection, **a single bit never means "attack."**

Use a risk score instead of a harsh response; the result is evaluated together with server-side verification.

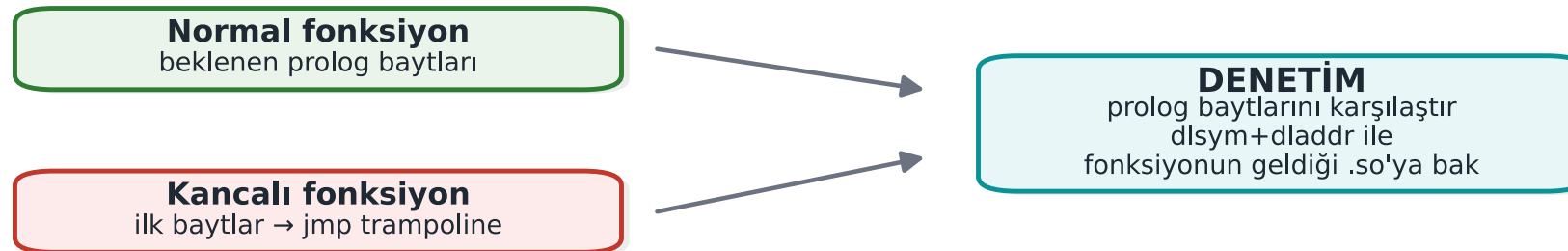


6. Hook and Instrumentation Detection

Hook Detection — Diagram

Kanca (hook) nasıl algılanır?

kanca, fonksiyonun ilk baytlarını değiştirir



libc/vDSO dışında bir modülden geliyorsa kanca vardır (LD_PRELOAD).

The Hook Threat

- The attacker **hooks** a critical function: intercepts and changes the call.
- Example: making the "is the signature valid?" function always return "yes."
- Tool: **Frida**, Xposed.

LD_PRELOAD Hooking

- On Linux, loading a library before the program to **replace** functions.
- The critical function is replaced with the attacker's version.
- RASP checks the loaded libraries.

Hooking Techniques — Table

Technique	How	Platform
LD_PRELOAD	Preloading a <code>.so</code> and replacing a function	Linux
PLT/GOT hooking	Redirecting a call-table entry	Linux/ELF
Inline hooking	Overwriting the first bytes with a <code>jmp</code>	Everywhere
Frida / Xposed	Dynamic binary instrumentation	Mobile/desktop

Detection Approaches (Concept)

- Unexpected loaded libraries/modules.
- Traces of known instrumentation tools (in memory, on ports).
- Function addresses **deviating** from the expected.

A Weak Approach: getenv

```
if (getenv("LD_PRELOAD") == NULL) printf("temiz\n");
```

`getenv` itself can also be **hooked** to return `NULL`. That's why you need to **verify the source**:

1. **Resolve** the critical function (`dlsym`) → 2. find **which module it comes from** (`dladdr`) → 3. check whether it's **legitimate** → 4. **decide**.

Step 1 · Resolve the Function

```
void *p = dlsym(RTLD_DEFAULT, "time");
```

- `dlsym` finds, at runtime, **which function** would actually be called.
- If a preloaded hook exists, `dlsym` returns **that one** (a manipulated result).

Step 2 · Find the Source

```
Dl_info info;  
dladdr(p, &info); /* fonksiyonu saglayan .so */
```

- `dladdr` tells you **which shared object (.so)** an address comes from.
- It does **not rely on** `getenv` -style spoofing — it looks directly at the memory address.

Step 3 · Check Whether It's Legitimate

```
int kanca = !mesru_mi(info.dli_fname);
```

- The expected source is **libc**, the **vDSO**, or the dynamic loader.
- If it comes from a foreign **.so** → **there's a hook**.

Step 4 · Decide

- If it's **not** in the list of legitimate sources → the hook flag is raised.
- The result isn't left alone; it's tied to the **response policy** (Section 10).
- It's **repeated** for multiple functions (a single function isn't enough).

Function Integrity

- Are the critical function's first bytes what's expected?
- A hook usually places a **jump** at the start → the bytes change.
- If they've changed → there's a hook.

Demo 4 · LD_PRELOAD Hooking — Actual Output

```
ADIM 1 - Normal calisma (LD_PRELOAD yok)
time      -> linux-vdso.so.1
getenv    -> /lib/x86_64-linux-gnu/libc.so.6
SONUC: Temiz - preload/kanca gorunmuyor.

ADIM 2 - Saldiri: sahte kanca YALNIZ bu surece yukleniyor
time      -> bin/linux/libsahtekanca.so (KANCA!)
time(NULL) dondurdu : 1234567890 (sabit sahte deger)
SONUC: Fonksiyon kancasi / preload ALGILANDI (2 sinyal).
```

In the clean run, `time` resolved from the kernel's **vDSO** — this is **legitimate**, not a hook.

Hook Detection Can Also Be Bypassed

- Advanced tools hide their traces.
- Strength: multiple methods + obfuscation + server-side verification.

This Section's Rule (6)

In hook detection, don't trust **manipulable** sources like `getenv(LD_PRELOAD)` .

Verify the **source** of critical functions (`dlsym` + `dlsym`); check multiple functions.



7. Dynamic Memory Protection

Memory Protection Layers — Diagram

Dinamik bellek koruması: hangi katman neyi kapatır?

bellek saldırganının en kolay ulaştığı yerdir

Döküm kapatma	PR_SET_DUMPABLE=0 — çökme dökümünde sır kalmaz
Bellek kilidi	mlock — sır takas alanına yazılmaz
Kısa ömür + silme	explicit_bzero — pencere daralır
Bütünlük sayacı	korunan değer yanında sağlama tutulur; değişirse yakalanır
İzleme tespiti	ptrace / okuma denemesi algılanır ve puana katılır

Hiçbiri tek başına yetmez; her biri saldırganın bir yolunu pahalılaştırır.

The Problem

- Sensitive values (keys, counters) sit in memory.
- The attacker can read/change memory and bypass a check.

What Does the Attacker Do With Memory?

Attack	Example	Target
Reading	Dumping process memory, searching for a known value	Key, PIN, decrypted data
Changing	Changing a counter/flag in memory	Logic and checks
Watching	Tools that alert when a value changes	Discovering where variables live

All of these require the attacker to **access** process memory (debugger, OS interface, injected library).

The Protected Counter Idea

- Instead of keeping a critical counter in the plain:
 - keep it together with a **shadow copy** + a **digest**.
 - check consistency on every read/write.

Protected Value · Concept

```
typedef struct { uint32_t deger; uint32_t golge; uint32_t ozet; } Korunan;  
/* okuma: deger == ~golge ve ozet doğru mu? */
```

If the attacker changes only `deger` (value), the inconsistency is caught.

Memory-Scanning Detection

- Is there unexpected access to sensitive regions being watched?
- Traces of a memory dump/scan.
- Goal: raise the cost of live memory analysis.



Memory Protection Is Limited

- A determined attacker can still get access.
- Goal: **delay**, make the key short-lived (Week 10).

RASP's Limits — Diagram

RASP'in sınırları: ne vaat eder, ne etmez?

cihazın sahibi saldırgansa yeterli zamanla her katman aşılır

ETMEZ

kırılmazlık
MATE'ye karşı kalıcı engel
kötü mimariyi kurtarma
meşru kullanıcıyı ayırt etme garantisi

EDER

saldıryı geciktirir
toplu kırılmayı zorlaştırır
telemetriyle görünürlük
riskli ortamda kapsamı daraltır

Etik sınır: kullanıcının cihazına zarar vermeyin, veri toplamayı şeffaf ve ölçülü tutun.

This Section's Rule (7)

Keep things in memory **briefly, minimally, and scattered**; protect a critical value with a **shadow copy + digest**.

An inconsistency must be tied to a response — silently ignoring it defeats the protection.

Section 5–7 — Quick Check

1. Why is there a false-positive risk in emulator detection?
2. Why does a hook change function bytes?
3. What does a protected counter catch?

Section 5–7 — Answers

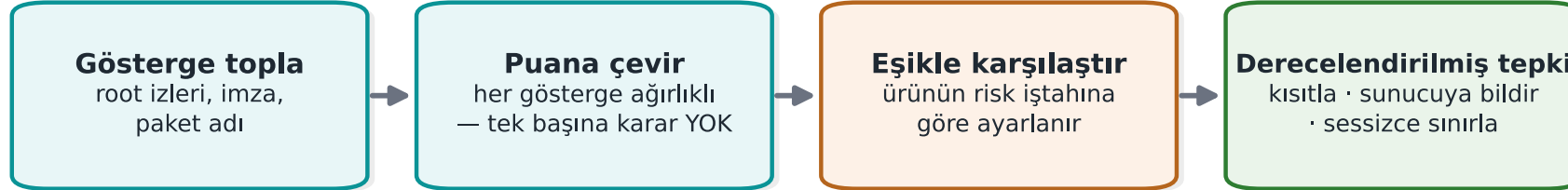
1. A legitimate user may also use a **VM/emulator/CI**; weak signals also show up on real devices → the risk of **blocking the innocent**. Grade the severity of the response.
2. A hook changes the target's **first bytes** with a jump (jmp/trampoline) so the call diverts into the attacker's code → **prologue bytes** differ from expected.
3. **A check in the control flow being skipped**: the counter increases as critical checks run; if it doesn't reach the expected value, a check has been **bypassed**.

8. Root Environment and Signature Verification

From Root Indicator to Decision — Diagram

Kök/ayrıcılık ortam: göstergedan karara

tek göstergeyle engellemek meşru kullanıcıyı vurur



Root'lu cihaz suçlu değildir; yalnızca güvence düzeyi düşüktür. Tepkiyi ona göre ölçekleyin.

Why Is a Rooted Device a Risk?

- Root = full privilege.
- Protections weaken; the attacker gains access to memory, files, the process.
- RASP checks "is the device rooted?"

Root Detection (Concept)

- The presence of known root tools/files.
- Being able to perform operations that are normally restricted.
- System-integrity indicators.

Root Detection Can Be Bypassed

- Root hiding tools exist.
- A single indicator isn't enough; **many** indicators + server-side verification.
- Response: restrict the critical feature, not block everything (don't hurt the user).

Component Signature Verification

- Is the caller's/loaded component's **digital signature** the expected one?
- Catch it if a fake/modified application is calling the library.
- Example: only the main application with the expected signature can use the SDK.

Signature + Digest Together

- Signature: **who** published the component?
- Digest: **has** the component changed?
- Together they make forgery harder.

Demo 6 and 7 · Actual Outputs

```
# Demo 7 (kök/ayrıcalık göstergesi)
Ayrıcalık seviyesi : normal kullanıcı
[BULUNDU] çıktı/sahte_su (ek işaret)
SONUC: Ayrıcalıklı/riskli ortam GOSTERGESİ var.

# Demo 6 (bileşen imzası)
Modül imzası TUTMADI -> YENİDEN PAKETLENMİŞ.
Yükleme REDDEDİLDİ.
```

Both demos work by **only reading/verifying**; they don't change the system.

This Section's Rule (8)

Root/privilege indicators are **a signal, not proof** — they're easily spoofed.

Cryptographically verify every dynamically loaded component **before** loading it; set up mutual verification where possible.



9. Control-Flow Integrity

Why Isn't a Single `if` Enough?

```
if (imza_gecerli()) devam(); /* tek nokta */
```

- The attacker **patches** this single branch → the check is skipped.
- Week 9's opaque boolean + randomised exit is critical here.

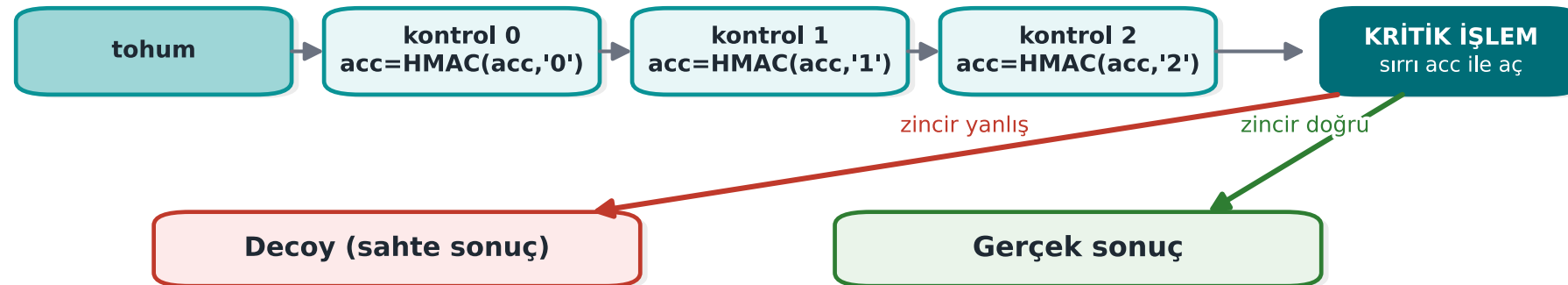
Control-Flow Counter

- Critical checks must pass in the correct **order**.
- Every check updates a **counter**.
- If the counter isn't the expected value at the end → one was skipped.

Control-Flow Counter — Diagram

Kontrol akışı sayacı: denetim atlandı mı?

her denetim akümülatörü ilerletir



Tek `if` bir NOP'la atlanır; sayaç, denetimin GERÇEKTEN çalıştığını sonradan kanıtlar.

Dual Counters

- Two independent counters (one increasing, one decreasing).
- Their sum/relationship must stay fixed.
- The attacker has to change **both, consistently**.

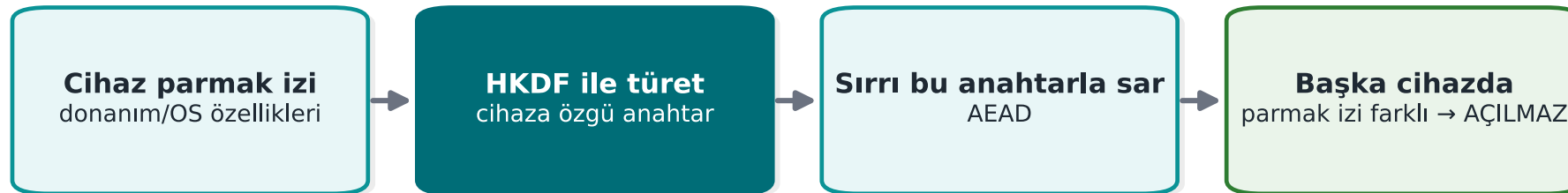
Tie the Result to a Behaviour

- The check result is not a plain `if`.
- The result feeds into **the computation** of the next step (e.g., a key derivation).
- If skipped, the program behaves **incorrectly** (breaks silently).

Device Binding — Diagram

Cihaz bağlama: kopyalama işe yaramasın

çalınan dosya başka cihazda anlamsızdır



Code lifting ve kopyalama saldırılarının ölçeklenmesini bu engeller (11. hafta).

Demo 5 · Skip Attack — Actual Output

```
SENARYO 1 - Normal: kontroller sirayla calisir
SONUC: ODEME ONAYLANDI -> "ODEME-ONAYI-TOKEN-4242"

SENARYO 2 - Saldiri: kontroller tamamen ATLANIR
      (uyari: sayac=0 iz=0x0 beklenen=3/0x7)
SONUC: ODEME REDDEDILDI (zincir anahtari yanlis). decoy doner

SENARYO 4 - Saldiri: kontroller YANLIS SIRADA calisir
SONUC: ODEME REDDEDILDI (zincir anahtari yanlis). decoy doner
```

When checks are skipped or their order is broken, the chain comes out different; the **real result couldn't be produced**.

This Section's Rule (9)

Don't tie critical decisions to **a single boolean** (it gets patched).

Tie security checks to a **control-flow counter** and, where possible, to a **data dependency**.



10. Response Policy and Deterrence

Why Is "Crash Immediately" Bad?

- It tells the attacker "the check is exactly here."
- It makes finding and bypassing the check easier.
- Smarter responses are needed.

Examples of Good Responses

- **Delayed:** trigger the response later, somewhere else (hide the source).
- **Indirect:** silently break the feature (produce a wrong result).
- **Notify the server:** raise the risk score, reject the operation server-side.

Response Strategies — Table

Strategy	What It Does
Fail-closed	Reject the operation when in doubt, don't fall back to a trusting default
Erase the secret	Securely erase the valuable data the instant tampering is found
Decoy output	Return a random/fake result instead of crashing
Delayed/implicit response	Separate the response from the trigger, in time/distance
Device/version binding	Bind the secret to the device/version
Telemetry/attestation	Report the event to the server; let the server's risk engine decide

Designing the Response

- Don't break the user experience (false-positive risk).
- A harsh response only on a **high-confidence** indicator.
- In most cases: restrict + server-side verification.

The Detect → Decide → Respond Loop

Demo 8 (`code/week-06/08-tamper-yanit`) combines this loop into a single **self-protection engine**:

1. A series of checks **run** (integrity, anti-debug, environment...).
2. The secret is **wrapped** with a device-bound key (HKDF + AES-GCM).
3. When tampering is detected, the engine **erases** the secret, raises a flag, and returns a **decoy**.

Step A · Detect (Scenario 1 — Normal)

```
SENARYO 1 - Normal: kontroller gecer, cihaz dogru -> sir acilir  
SONUC: TEMIZ. Sir acildi ve islem yapiliyor  
-> "ODEME-ANAHTARI-7C4A"  
(sir kullanildiktan sonra bellekten guvenle silindi)
```

Checks **passed**, the device was correct → the secret was opened, used, and immediately **erased**.

Step B · Decide (Scenario 2 — Tamper)

```
SENARYO 2 - Tamper: bir kontrol basarisiz -> sil + decoy + bayrak
SONUC: TAMPER ALGILANDI -> algilama kontrolu basarisiz
Politika: sir silindi, tamper bayragi kaldırıldı,
olay kaydedildi.
Cokmek yerine SAHTE (decoy) sonuc dondu: 60797327...
```

One check **failed** → the engine **decided**: erase + log + decoy.

Step C · Respond (Scenario 3 — a Different Device)

```
SENARYO 3 - Baska cihaz: kontroller gecer ama  
             cihaz anahtari tutmaz  
SONUC: TAMPER ALGILANDI -> cihaz/surum baglama tutmadi  
      (klonlama?) ... decoy doner
```

Even though checks passed, the device-bound key didn't match → the secret **couldn't be opened**. This is the innermost layer of Week 3's security shell.

Device Binding

- The key/data is meaningful only on a **specific device**.
- If copied (to another device), it's useless.
- A defence against code lifting (Weeks 9/11).

The Device-Binding Flow · Step by Step

Step 1 — read the fingerprint: the device's hardware/OS fingerprint is read (Week 5).

Step 2 — derive a key: a key is derived from the fingerprint + version string with **HKDF**; the secret is wrapped with this key using **AES-GCM**.

Step 3 — if moved, try to open: even if the file is copied to another device, that device's fingerprint is **different**, so the derived key won't match → the secret **cannot be opened**.

Deterrence · the Ultimate Goal

RASP does not make the attack **impossible**; it makes it **costly and risky**.

Enough layers + server-side verification → the attacker gives up or gets caught.

This Section's Rule (10)

Separate the response from the trigger: an immediate `exit()` hands the attacker a map.

The instant tampering is found, **erase** the secret, return a **decoy** instead of crashing, report the event to the server, bind the secret to the device.

Section 8–10 — Quick Check

1. Why is a rooted device a risk?
2. Why does a counter strengthen a single `if`?
3. Why can "crash immediately" be a bad response?

Section 8–10 — Answers

1. **Root/jailbreak** removes the barriers against reading/writing memory, hooking, and certificate injection → RASP's assumptions and isolation collapse.
2. A single `if` is bypassed with a NOP/patch; the **counter** verifies after the fact that the check actually ran (distributed proof) → a single patch isn't enough.
3. "Crash immediately" shows the attacker **exactly where** the check is, and on a false positive it hits a legitimate user. A **delayed/silent/deceptive** response is better.



11. Limits and Ethics

RASP's Limits

- Every single check **can be bypassed**.
- Its strength decreases on a rooted device.
- A false positive → a real user can be hurt.
- Maintenance cost is high (tools evolve).

That's Why RASP Is...

- **Layered** (many checks), **diversified**, **hidden**.
- Combined with **server-side** verification (the real guarantee).
- Claiming it's "unbreakable" on its own is **wrong**.

Ethics and the User

- RASP runs on the user's device; **excessive data collection** is not ethical.
- A false positive can deny the user service → respond carefully.
- A balance between transparency and privacy.

The Safe-Example Rule

- Course demos **do not harm** the student's device.
- No real root/system change; synthetic indicators.
- The goal is teaching the concept, not building an attack tool.



12. Project: This Week (S10)

Project · S10 (RASP + Response)

- [] At least two different RASP checks (e.g., integrity + anti-debug).
- [] Document **when/where** the checks run.
- [] Response policy: what happens when a threat is seen?
- [] The device-binding decision and its rationale.
- [] Limits and residual risk (what it doesn't protect).

Through the Evaluator's Eyes

- "RASP exists" isn't enough; **how many layers, how hidden, what response?**
- How does it combine with server-side verification?
- How is the false-positive risk managed?



Solved Self-Check

Question 1

What are RASP's three jobs?

Answer: Detect (abnormal environment/tampering), defend (restrict/halt the feature), deter (raise the cost).

Question 2

What does self-hashing catch, and why isn't it enough alone?

Answer: It catches a patch/tampering applied to the binary. It isn't enough alone because the attacker can find and bypass the digest function; obfuscation + cross-checking + overlapping checks are needed.

Question 3

Why is cross-checking strong?

Answer: Java checks native, native checks Java; the attacker has to bypass **both at once**.

Question 4

How does timing-based anti-debug work?

Answer: If a small piece of work took much longer than expected, someone may be single-stepping the program (a debugger).

Question 5

Why does a control-flow counter strengthen a single `if` check?

Answer: It counts that checks passed in the correct order; patching a single branch breaks the counter, and the skip is caught.

Question 6

Why isn't "crash immediately" always a good response?

Answer: It shows the attacker exactly where the check is. A delayed/indirect response + server notification is better.

Question 7

Why does the false-positive risk matter in emulator detection?

Answer: A real device/developer environment can look like an emulator; a harsh response hurts a real user.

Question 8

What is RASP's fundamental limit?

Answer: Every single check can be bypassed; it weakens on a rooted device. Strength comes from layering, diversity, and server-side verification.



Quiz-1-Style Sample Questions

Example · Multiple Choice

What is the goal of RASP cross-checking?

- A) Shrinking the code
- B) Forcing the attacker to bypass both sides ✓
- C) Speeding up encryption
- D) Turning off logging

Example · Short Answer

Name three methods that strengthen an integrity check.

Obfuscation · multiple/overlapping blocks · cross-checking · tying the result to a behaviour.

Glossary

Term	Meaning
RASP	Runtime self-protection
MATE	The attacker at the endpoint (the device's owner)
Self-hashing	Checking a digest of one's own code
Hook/Frida	Intercepting and changing a function call
CFI counter	Verifying check order by counting
Device binding	Binding data to a specific device (with HKDF)
Decoy	A fake result returned instead of crashing
Fail-closed	Rejecting an operation when in doubt
Attestation	Server-side proof of integrity

Summary: This Week in One Sentence

RASP monitors itself while the program runs and responds to threats; but it gains its strength not from a single check, but from **layered, diversified, hidden** checks and **server-side** verification.



Next Week

Week 7 — Interim Project Demo (RAP1) and Week 8 — Quiz-1

Midterm period: demo the first half of your project and prepare for Quiz-1.



Appendix A · Building a RASP Layer

Goal

Let's protect a synthetic "license check" with RASP:

- integrity checking
- debugger checking
- a smart response

Step 1 · The Point to Protect

```
int lisans_gecerli(void) {  
    /* ... kontrol ... */  
    return sonuc;    /* saldirganın hedefi */  
}
```

This function and the flow that calls it are critical.

Step 2 · Add an Integrity Check

- After compilation, record this region's digest.
- Recompute and compare while running.
- A difference → raise the tampering flag.

Step 3 · Add Anti-Debug

- "Am I being watched?" via timing + OS indicators.
- Don't leave the result alone; **combine** it with another check.

Step 4 · Tie the Result to a Behaviour

- `lisans_gecerli` 's result is not plain true/false.
- The integrity + anti-debug indicators feed into the result's **computation**.
- If there's tampering, the function behaves **incorrectly**.

Step 5 · Response

- Instead of crashing immediately: report the flag to the server, restrict the feature.
- Delayed trigger: hide the source.

Step 6 · Obfuscate and Diversify

- Obfuscate the check code (Week 9).
- Place it differently in every version (diversification).
- Add a cross-check.

Step 7 · Measure and Document

- How many checks, where, and what response?
- Performance cost (checks slow things down).
- Write it up in S10.



Appendix B · Mini Case Study

Scenario · A Payment Application

- A local key + payment flow.
- Goal: make tampering and live analysis hard.

Case · Check Placement

- **At startup:** root/emulator/signature check.
- **Before payment:** integrity + anti-debug + hook check.
- **Periodic/random:** repeat.

Case · Response

- A high-confidence indicator → reject the operation + notify the server.
- A weak indicator → raise the server-side risk score.
- Protect the user experience (false positives).

Case · Layers

- RASP + obfuscation (Week 9) + whitebox crypto (Week 11) + short-lived keys (Week 10).
- Server side: rate limiting, anomaly detection.
- Even if one layer is breached, the others hold.

Case · Residual Risk

- A determined attacker can still make progress on a rooted device.
- But: many layers + server-side verification → the damage is limited, the attack is expensive.
- This is written down explicitly.

Appendix C · RASP Check Catalogue

Catalogue (1)

Check	What It Catches	Limit
Integrity (self-hash)	A binary patch	The digest fn. can be bypassed
Anti-debug	A debugger attached	Known methods get bypassed
Emulator	A fake environment	False positives

Catalogue (2)

Check	What It Catches	Limit
Hook/Frida	A function hook	Hidden tools
Root	Privilege escalation	Root hiding tools
Signature	A fake component	If the key leaks
CFI counter	A skipped check	A sophisticated patch

The Lesson From the Catalogue

- Every check has a **blind spot**.
- None of them is enough alone.
- Choose **several** from the catalogue, combine them, diversify them.

Choosing Checks

- How many layers, based on the asset's value?
- Which checks make sense for the platform?
- What's the performance budget?

Write the decision and its rationale in S10.



Appendix D · Attacker ↔ Defender

Why This Dialogue?

Security is a **move-countermove** game.

Every defence gets an attack, every attack gets a new defence.

This is why layered defence is necessary.

Integrity · Dialogue

- **Defence:** add self-hashing.
- **Attacker:** find the digest function, force it to always say "match."
- **Defence:** obfuscate the digest check, overlap it, tie the result to a behaviour.

Anti-Debug · Dialogue

- **Defence:** add a debugger check.
- **Attacker:** find the check, bypass it.
- **Defence:** many methods + timing + obfuscation; make the response delayed.

Hook · Dialogue

- **Defence:** check the function's bytes.
- **Attacker:** hide the hook, restore the bytes.
- **Defence:** cross-checking + server-side verification.

Root · Dialogue

- **Defence:** check root indicators.
- **Attacker:** use a root-hiding tool.
- **Defence:** many indicators + a server-side risk score; restriction instead of a harsh response.

The Lesson From the Dialogue

- A single defence is always eventually bypassed.
- Strength: **multiplicity + secrecy + diversity + the server.**
- Goal: make the attack **no longer economical.**



Appendix E · Response Decision Flow

"I Saw a Threat — What Do I Do?" — 1

Question 1: How reliable is the indicator?

- **Highly reliable** → reject the operation + notify the server.
- **Uncertain** → continue, but raise the risk score.

"What Do I Do?" — 2

Question 2: Will it affect the user experience?

- If the false-positive risk is high → **don't respond harshly.**
- Prefer silent restriction + server-side verification.

"What Do I Do?" — 3

Question 3: Does the response give away the check?

- Crashing immediately → reveals the source.
- A **delayed/indirect** response → hides the source.

Response Flow · At a Glance

Tepki politikası: ne yapmalı?

tepki, algılamadan daha çok düşünülmesi gereken kısımdır

Hemen çok

saldırgana denetimin YERİNİ gösterir — kötü

Sessiz/gecikmeli tepki

sonraki işlemde sırrı sil, bağlantıyı kes

Decoy (sahte sonuç)

saldırgan yanlış yolda zaman kaybeder

Sunucuya bildir

telemetri + hesap/cihaz kısıtlama

Yanlış pozitifte meşru kullanıcıyı vurmayan, saldırganı yormayan tepki seçilir.



Appendix F · Quick Reference

Common Mistakes

- Relying on a single check.
- Not obfuscating the check (easily found).
- A "crash immediately" response (gives away the source).
- A harsh response + a high false-positive rate (hurts the user).
- Skipping server-side verification (the real guarantee is there).

RASP Checklist

- [] ≥ 2 different types of checks
- [] Checks are hidden + diversified
- [] Cross-checking (Java $\leftrightarrow$ native)
- [] A smart response (delayed/indirect)
- [] Device binding
- [] Server-side verification
- [] A false-positive plan

Final Word (Week 6)

RASP is a **time** game played against the attacker: every layer buys a little more time.

The real guarantee: layered RASP + obfuscation + short-lived keys + **server-side verification**.