Cybersecurity Testing Fundamentals

Fundamental question

- Mistakes happen in **every** field of engineering
  - ...occasionally

- Why are they so **common** and so **pervasive** in **software**?

Reasons (in a nutshell)

1. **Intrinsic** problems of **software** and of **cybersecurity**
   - Fundamentally different from every other field
   - Cannot go away because of some magic technology

2. Lack of **incentives** for minimizing likelihood and impact
   - **Very complex issue**

Intrinsic Problems?

1. **Intrinsic** problems of **software** and of **cybersecurity**
   - Fundamentally different from every other field
   - Cannot go away because of some magic technology

- Very intuitive (but very useful, I think) analysis of **cybersecurity testing**

Our focus on Cybersecurity Testing

- What makes it **hard**
- What makes it **intrinsically different** from any other form of testing

- **Out of scope**: How to execute it in practice

Non-SW Example: Fridge

- Requirement:
  - Operation with external temperature [-15,+40] °C

Non-SW Example: Testing

- Requirement:
  - Operation with external temperature [-15,+40] °C

- You will **not** test with 200 °C / -50 °C

- Test with +25 °C
  → Behavior as expected

- Behavior with 25.43 °C / 25.81 °C / 24.63 °C / ...
  will be "**similar**"

Non-SW Testing

1. Tests for inputs representative of **known** operating conditions
2. System behavior on **untested** inputs: **not "very different"** from behavior with tested inputs

SW is "different"!

1. Tests for inputs representative of **known** operating conditions

- Adversaries may inject **carefully selected** and **unexpected inputs** in the system (**adversarial** world)

2. System behavior on **untested** inputs: not "very different" from behavior with tested inputs

- Software is **not continuous**: The output of a test with a certain input tells you **nothing** about the system behavior with a **similar but different** input

Perfect Storm

1. Adversaries may inject **carefully selected** and **unexpected inputs** in the system (**adversarial** world)

- **Totally unexpected** inputs **carefully selected** for violating guarantees are **routinely** injected

2. Software is **not continuous**: The output of a test with a certain input tells you **nothing** about the system behavior with a **similar but different** input

- **No idea** about system behavior with **untested** inputs

- Potential outcome is evident
- No other technology has these (very bad) features

Consequence on Design/Development

1. Adversaries my inject **carefully selected** and **unexpected inputs** in the system (**adversarial** world)

- A SW artifact **must** take into account **unexpected** inputs

- A Non-SW artifact need not
  - You will not design a fridge for operating at 200 °C / -50 °C

Consequence on Testing

1. Adversaries my inject **carefully selected** and **unexpected inputs** in the system (**adversarial** world)

- Testing of a SW artifact **must** (well, should) consider **unexpected** inputs

- Much more difficult than it would appear

SW Testing Example: Palo Alto vulnerability

- HTTP Request with header `Cookie: SESSID=./../../../`

- We need to **test** behavior with **unexpected** HTTP Requests

- `AAAAAA...AAAA /index.html HTTP/1.1`
- `GET ///////index.html HTTP/1.1`
- `GET %4n%n%n%n%n%n. html HTTP/1.1`
- `GET /AAAAAAAAAAAAA. html HTTP/1.1`
- `GET /index.html HTTTTTITTITTITIP/1.1`
- `GET /index.html HTTP/1.1.1.1.1.1.1.1`
- ...

Hhhmmm…(I)

- We need to **test** behavior with **unexpected** HTTP Requests
- Nearly endless ways for constructing unexpected HTTP Requests…

Hhhmmm…(II)

- Software is **not continuous**: The output of a test with a certain input tells you **nothing** about the system behavior with a **similar but different** input

- `GET /AAAAAAAAAAAAA. html HTTP/1.1`

What if the letter was different?
What if the sequence was longer/shorter?
What if there were two (or more) sequences?

Negative Requirements

Non-SW Testing: Examples

- Requirement: *"It must tolerate loads up to 400 Kg / m2"*
- Test:
  - Put a load of 400 Kg / m2
  - Check that it is stable

- Requirement: *"With an input of 3 mW, signal-to-noise ratio at 30 km of at least 46 dB "*
- Test:
  - Transmit 3mW
  - Measure signal-to-noise ratio at 30 Km

Non-SW Testing (I)

- **Requirement:** *"A certain action can be done"*
- **Test:** Check that it can be done

- Requirements

↓

- Select inputs to test

↓

- Check behavior against Requirements

Non-SW Testing (II)

- **Proof** that requirements are satisfied on **tested inputs**

  +

- Untested inputs:
  Behavior "not very different" from tested inputs

- When systems are delivered, we are sure they satisfy their requirements (except for occasional mistakes)

Cybersecurity Testing: NEGATIVE Requirements

- Requirement: *"A certain action **cannot** be done"*

  +

- Software is not continuous

- One has to:
  1. Identify **all** the possible ways for attempting to execute the action
  2. Check them **all**

Cybersecurity Testing: Example (I)

- Requirement: *"Lucifer cannot read file x"*

- Test (basic idea):
  - All possible **inputs** that might include an attempt to read file x
  - ...even **totally unrelated** to "file read"
    (remember Heartbleed – read overflow?)
  - All possible **sizes**, **properties**, **ACL** of file x

- How many tests?
- How to select an "optimal subset"?

Cybersecurity Testing: Example (II)

- Requirement: *"Lucifer cannot read file x"*

- Test (basic idea):
  - Bug observable in corresponding output (Heartbleed)
  - Bug **not** observable in corresponding output (PaloAlto)
  - Bug observable only in some **future** output (WingFTP)

- How to **detect** the bug?
- How to make sure it is indeed a vulnerability?

Cybersecurity Testing

- Proof of correct behavior on tested inputs
  is **not** enough because of **negative requirements**
- **Exhaustive** testing of the input space **practically unfeasible**
- **Software is not continuous:**
  No idea about system behavior with **untested** inputs

- When systems are delivered, we **cannot be** sure they satisfy their cybersecurity requirements
- **FUNDAMENTAL** problem: It will **not** disappear with some magic bullet

Remark 1

- **Non-SW** artifacts:
  When systems are delivered, we **can be** sure they satisfy their requirements

- **SW** artifacts:
  When systems are delivered, we **cannot be** sure they satisfy their cybersecurity requirements
- Fundamentally a cross your fingers technology

Remark 2

- Proof of correct behavior on tested inputs is **not** enough because of **negative requirements**
- **Exhaustive** testing of the input space **practically unfeasible**
- **Software is not continuous:** **No idea** about system behavior with **untested** inputs

- When systems are delivered, we **cannot be** sure they satisfy their **functional** requirements
- **Intrinsic** issue of **software**
- Additional **cybersecurity** problem: **adversarial** world

Remark 3

- When a software artifact is released, no one knows **how many bugs** it has

- You can count **known bugs** but you cannot tell how many bugs **remain**

- More and "more accurate" testing
  $\Rightarrow$ more **"confidence"** it has less remaining bugs

A Few Words on Security Metrics

Guarantees for SW artifacts (I)

- What we would need:
  - A (near-)proof that:
    1. The system will do what it should
    2. The system will **not** do what it should **not**

- **Not feasible:**
  1. Adversarial environment
  2. Software is not continuous
  3. Negative requirements

Guarantees for SW artifacts (II)

- What we can achieve:
  - ~~A (near-)proof~~
    some "**degree of confidence**" (**assurance**) that:
    1. The system will do what it should
    2. The system will **not** do what it should **not**

- Cannot be quantified / measured
- Somewhat subjective

Assurance

- Many complementary techniques ("shift left")
  - Programming language and methodology
  - Development process
  - Testing methodology and effort
  - ...
  - Penetration tests
  - ...
- "High assurance" systems:
  - Strong and proven usage of such techniques

How can it be that?

- What we can achieve: some "**degree of confidence**" (**assurance**)
- Cannot be quantified / measured
- Somewhat subjective

- *System X has 30 vulns, System Y has 20 vulns*
- *10 PM testing for System X, 1 PM testing for System Y*
- *...*

That's the way it is

Cyber Hard Problems: Focused Steps Toward a Resilient Digital Future (2025)

NATIONAL ACADEMIES *Sciences Engineering Medicine*

Security metrics sufficient
to **predict** or **verify** the security properties
of a cyber system are **non-existent**

Example: Vulnerabilities

- Examples for highlighting difficulties
- Not formal proofs

#known-vulns

- #known-vulns(**SW-x**, t) > #known-vulns(**SW-y**, t)

SW-x *is more secure than* ***SW-y***

Hhmmm...(I)

- #known-vulns(**SW-x**, t) > #known-vulns(**SW-y**, t)

- #known-vulns(SW-x, t) **not a good predictor** of:
  - #known-vulns(SW-x, t**+DELTA**)
  - #vulns-**unknown**-and-actually-exploited(SW-X)

Hhmmm...(II)

- #known-vulns(**SW-x**, t) > #known-vulns(**SW-y**, t)

- How do we quantify:
  - **Impact** of vulns?
  - Difficulty of **injection**?

Hhmmm...(III)

- #known-vulns(**SW-x**, t) > #known-vulns(**SW-y**, t)

- #known-vulns(SW-x, t) is not even a property **intrinsic** to SW-X
- It greatly depends on the **population of vulnerability hunters** interested in SW-X:
  - Size
  - Actual effort
  - Skills
  - Motivations

Hhmmm...(IV)

- #known-vulns(**SW-x**, t) > #known-vulns(**SW-y**, t)

- How do we quantify:
  - Existence and effectiveness of workarounds?
  - Availability of **patches**?
  - Difficulty of **installing** a patch?
  - **Time** for releasing a patch?

Vendor-related

- How a vendor **reacts** to a vulnerability is much more informative for security than the vulnerability itself (Matt Blaze)

Keep in mind

Cyber Hard Problems: Focused Steps Toward a Resilient Digital Future (2025)

NATIONAL ACADEMIES Sciences Engineering Medicine

Security metrics sufficient to **predict** or **verify** the security properties of a cyber system are **non-existent**

- A lot (really a lot) of deep consequences
  - How do you justify investments?
  - How do you define public policies?

"Shifting left"

Software Development Life Cycle (SDLC)

Requirements
Use cases

Design

Coding

Testing

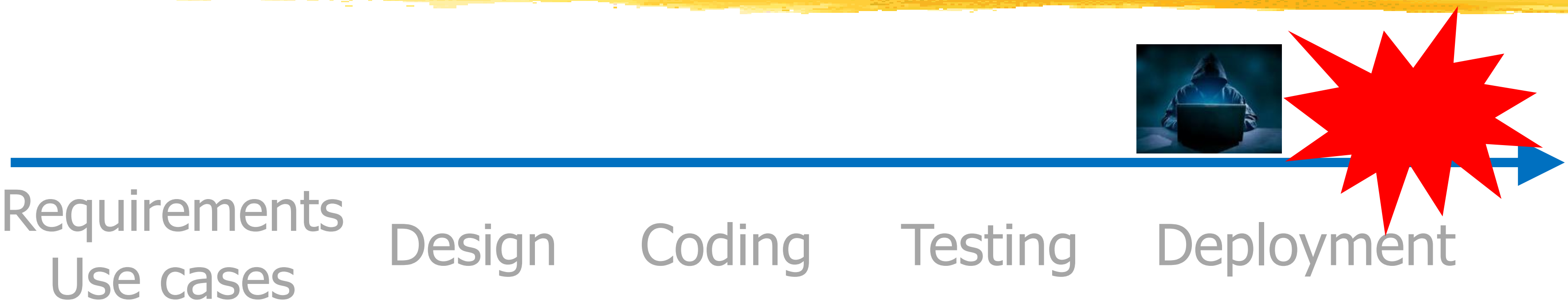
Deployment

Cybersecurity in SLDC (I)

Requirements
Use cases | Design | Coding | Testing | Deployment

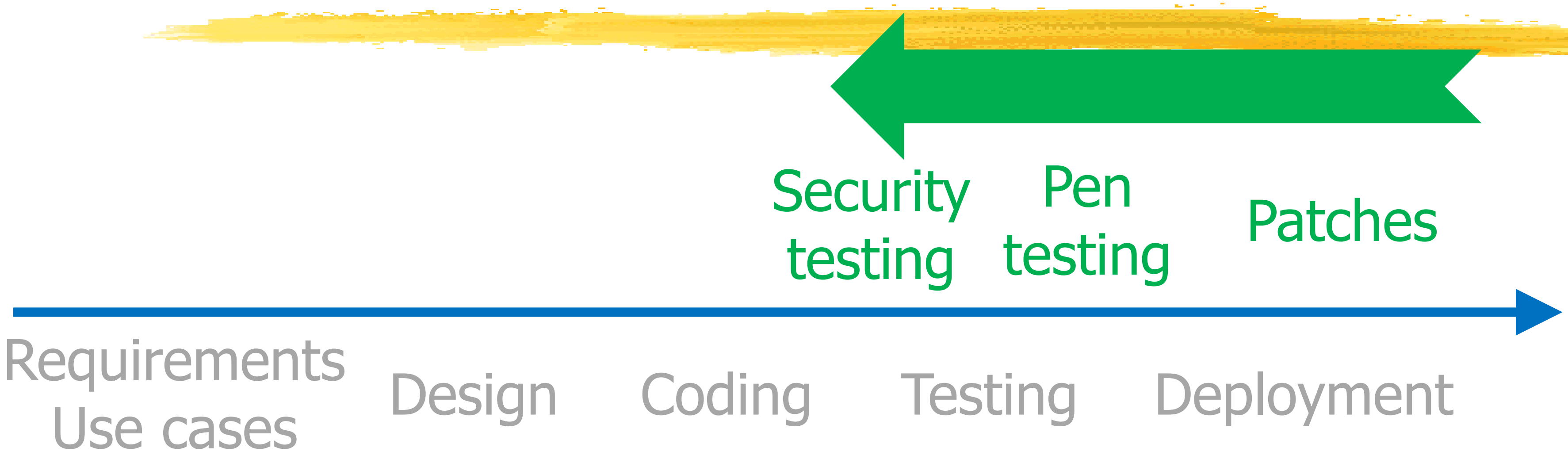
- Typical manufacturer behavior:
  1. Do nothing
  2. ...

Cybersecurity in SLDC (II)



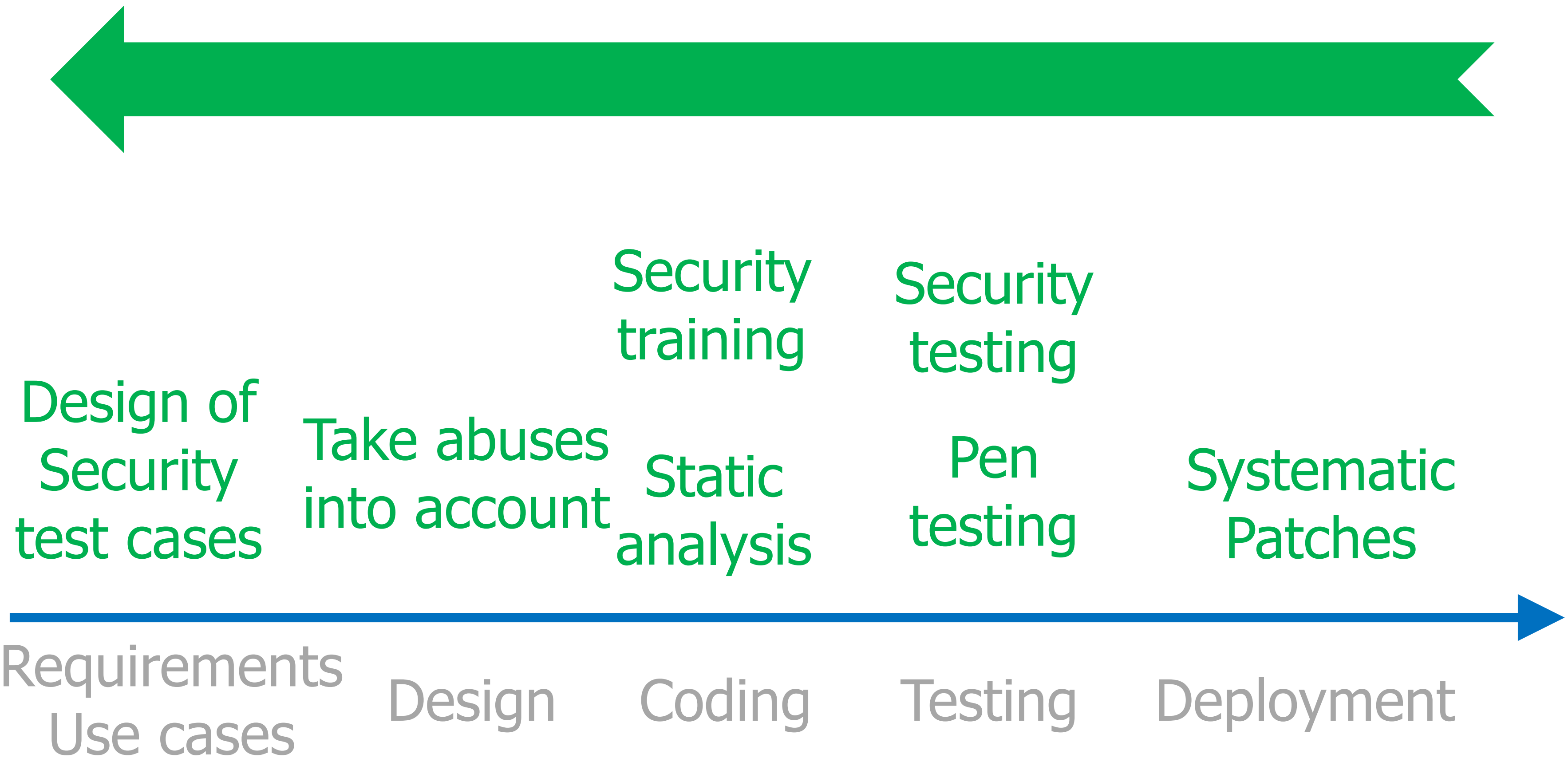
- Typical manufacturer behavior:
  1. Do nothing
  2. Tackle security issues **after deployment** (e.g., occasional patches)
  3. ...

Cybersecurity in SLDC (III)



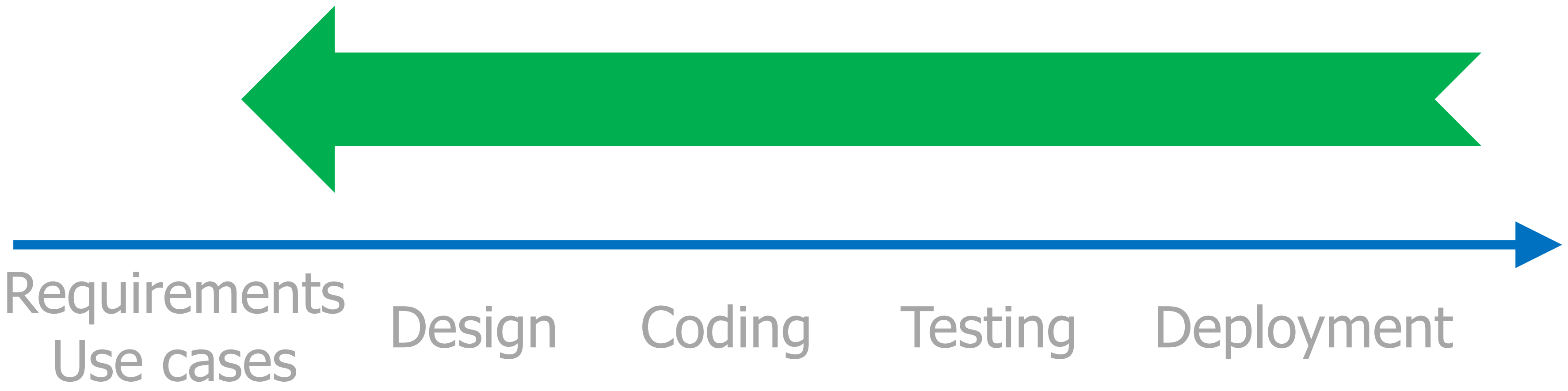
- Typical manufacturer behavior:
  1. Do nothing
  2. Tackle security issues **after deployment**
  3. **Evolve** to tackle them **earlier** ("**shift left**")

"Shifting left"



Keep in mind (I)

- We do **not** know how to make software secure
- ...but we **do** know how to make software **more secure**



Keep in mind (II)

- **Slow** evolution
- **Necessary** condition: strong **incentives**