Fundamentals of Secure Software Engineering

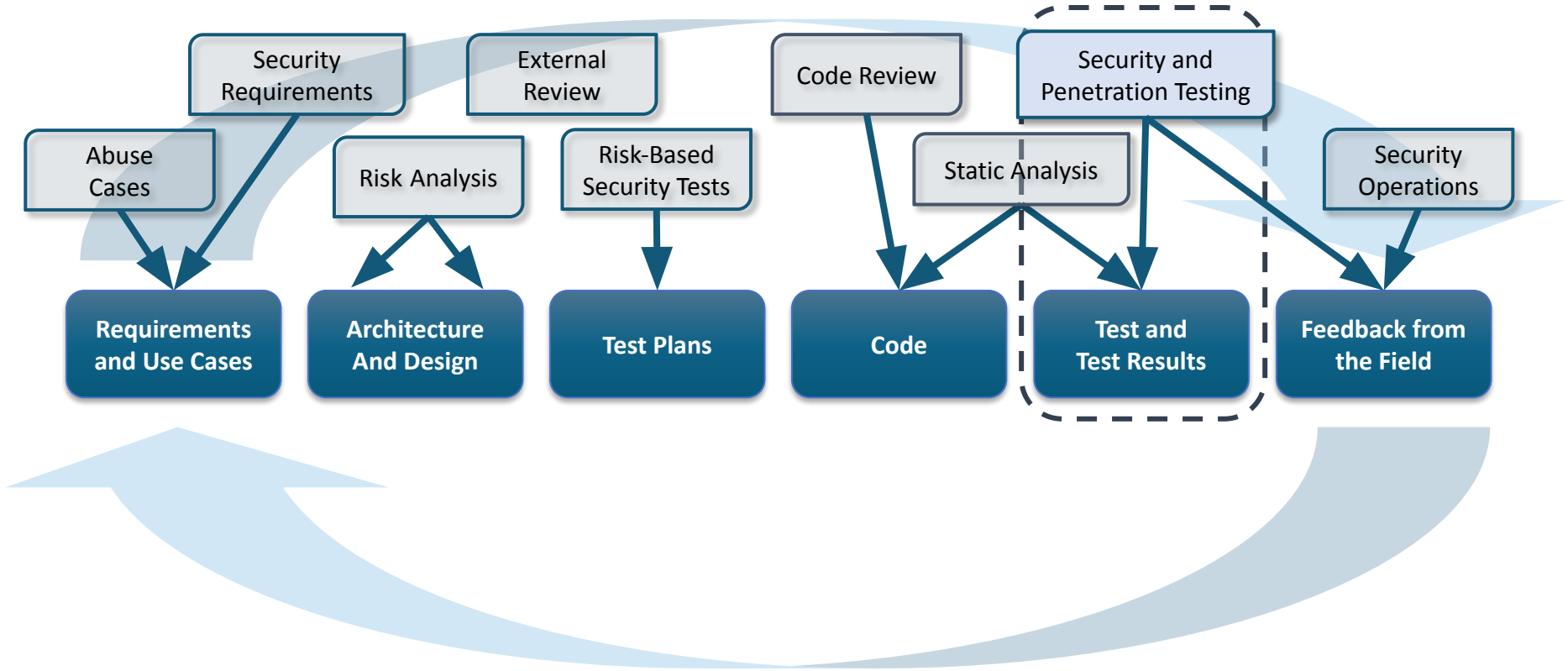
Winter term 2025/26

Code Testing

Dr. Sergej Dechand / Dr. Christian Tiefenau

Today's Lecture

Shifting further to Security Testing



Outline today

🐛 Vulnerability of the day

- Compression Bomb

⚙️ Recent Trends in Dynamic Analysis

- How to measure progress
- Instrumentation
- How to enforce progress
- How bug detection is handled

Vulnerability of the day

Compression Bomb

Compression Bomb

🦹 Attack Idea

- Compression works well for statistical redundancy
- Represent longest common sequence using as few bits as possible

aaabbb**aaa**b**aaa**b**aaa**c**aaa**

⬇

$**bbb**$b**$**b**$**c$

💾 (In)famous example is 42.zip

- Is 42 kilobyte compressed data
- Contains five layers of nested zip files in sets of 16
- Each bottom layer archive contains a 4.3 gigabyte file
- Uncompressed 4.5 Petabyte
- Factor ≈ 10^7

💥 Also known as CWE-409 Data Amplification

Compression Bomb

🧩 Compression is ubiquitous and hard to detect

- static analysis detects only obvious cases
- png, jpeg, e.g., libpng
- xml, e.g., xml bombs
- Office documents are simple zip files of XML
- HTTP response compressed by web server
- Git Bomb https://kate.io/blog/git-bomb/

💣 Leading to DoS by filling up RAM or hard disk

🛡️ Mitigation

- Your software must inspect compressed input anyway
- Keep track how many bytes have been decompressed, or limit rounds (sadly, often not supported in libs)
- **Distrustful Decomposition:** limit resources for process

Recap

Dynamic Analysis

Recap | DAST Pro's and Con's

✅ Advantages

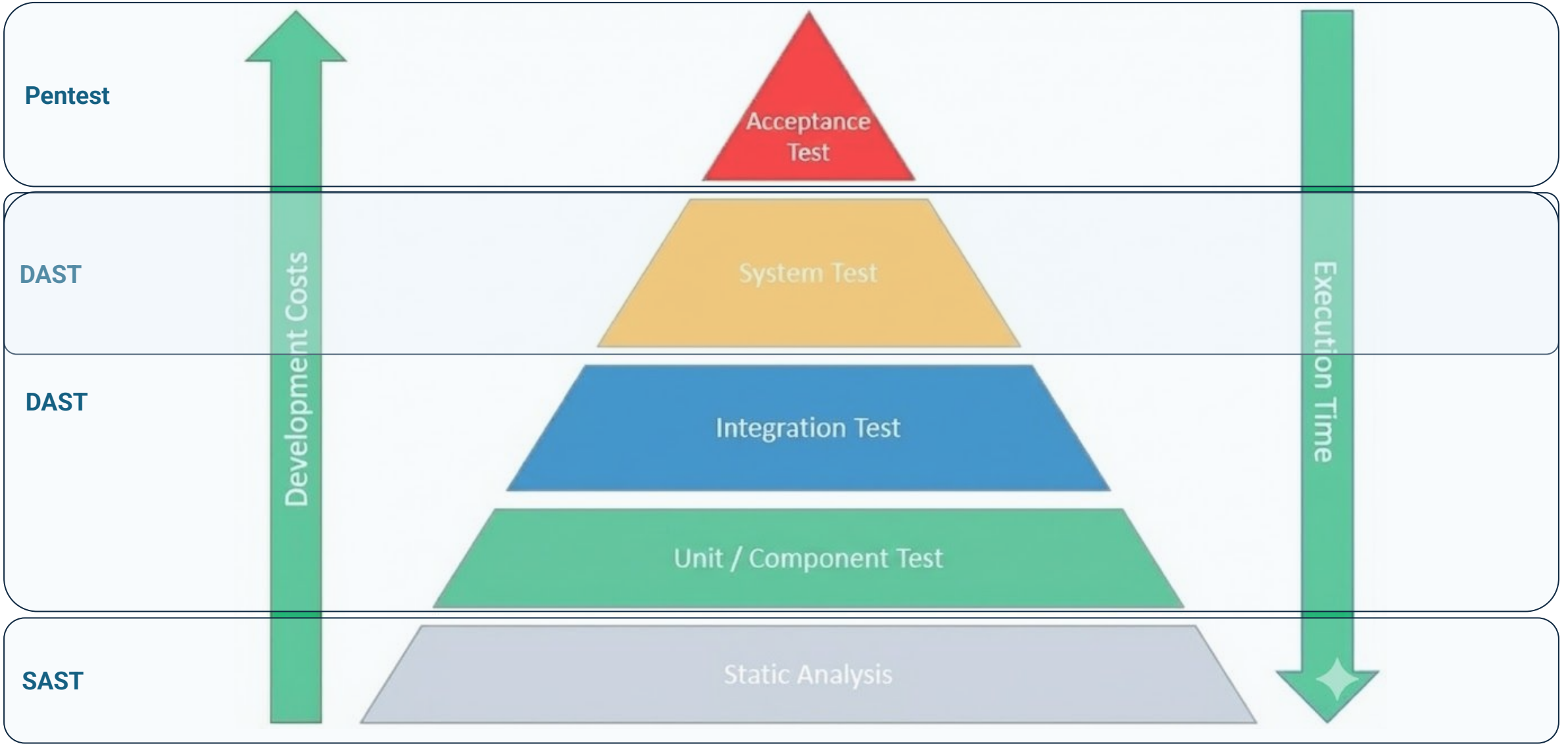
- No source code required / language-agnostic
- Finds vulnerabilities that SAST overlooks
- Provides the input that triggers the vulnerability

❌ Disadvantages

- Simple source code constructs can confuse DAST (Difficulties with complexity)
- No defined end (when to stop testing)
- Requires a deployed/running application (late-stage SDLC)
- Requires good API documentation

Recap | Remember the Test Pyramid?

Same concepts apply for security testing



Recent Trends in DAST

Use Instrumentation during dynamic Analysis

Question

Any idea how to measure DAST progress?

Code Coverage (Testabdeckung)

Same metrics as in functional testing can tell us whether security testing is making progress

📊 Coverage Types

- **Line/Statement coverage**: which lines were executed?
- **Branch coverage:** which decision paths were taken?
- **Function coverage:** which functions were called?

⚔️ Security Testing

- Higher coverage: more code explored for vulns
- Will show blind spots
- How to measure code coverage?

```
public ImageFormat detectFormat(byte[] input) {
    if (input[0] == 0x89 && input[1] == 0x50) {
        return ImageFormat.PNG;
    }
    else if (input[0] == 0xFF && input[1] == 0xD8) {
        return ImageFormat.JPEG;
    }
    else if (input[0] == 0x47 && input[1] == 0x49) {
        return ImageFormat.GIF;
    }
    return ImageFormat.UNKNOWN;
}
```

Instrumentation | See Inside a Program

Modifying software so that analysis can be performed on it through marker injection

Original Code

```
public ImageFormat detectFormat(byte[] input) {
    if (input[0] == 0x89 && input[1] == 0x50) {
        return ImageFormat.PNG;
    }
    return ImageFormat.UNKNOWN;
}
```

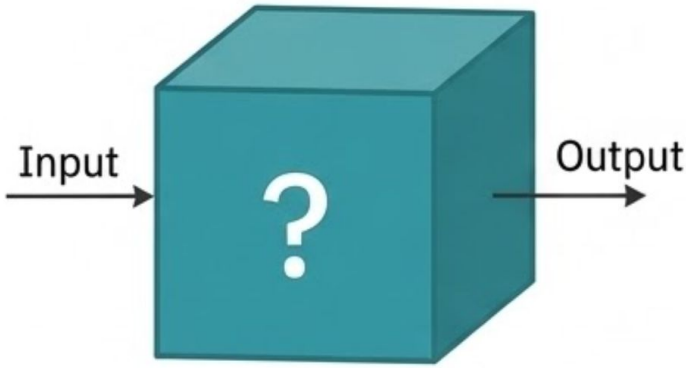
Instrumented Code

```
public ImageFormat detectFormat(byte[] input) {
    __coverage[0]++;  // ← track entry
    if (input[0] == 0x89 && input[1] == 0x50) {
        __coverage[1]++;  // ← track branch
        return ImageFormat.PNG;
    }
    __coverage[2]++;  // ← track branch
    return ImageFormat.UNKNOWN;
}
```

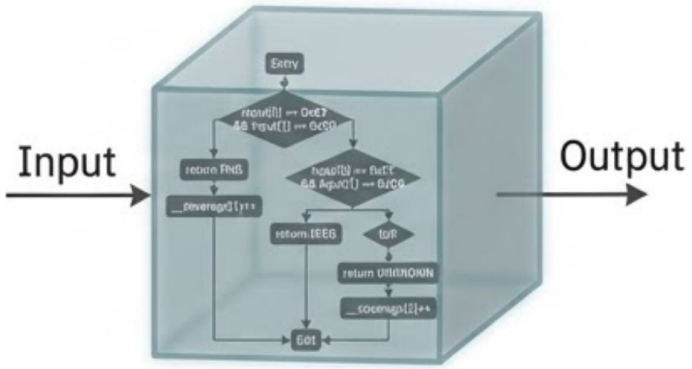
Allows new approaches

The world isn't black and white

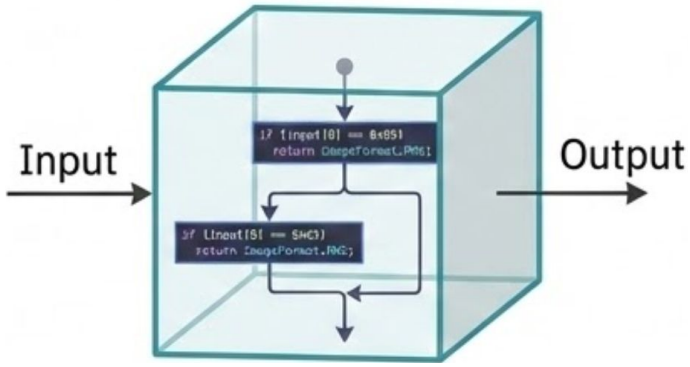
Blackbox Testing



Greybox Testing



Whitebox Analysis



Problem with DAST

fuzz | verb.
/'fəz/
1. to make or become blurred

⌨️ Input Generation / Fuzzing

Mutated Inputs

Feedback

🔎 Observe System Under Test

`JPEG Parser`

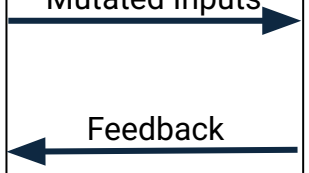
Instrumented (Greybox) Dynamic Analysis / Fuzzing

Instrumentation feedback improves both issue detection and input evolution

fuzz | verb.
/'fəz/
1. to make or become blurred

Fuzzing Engine

- Scripted test cases
- Fuzzing
  - Mutated inputs
  - Payload variations
  - Boundary value cases
- Authentication bypass attempts
- Use instrumentation feedback
  - to know whether new code was reached
  - Produce better inputs



🔎 Bug Detection

- Observe
  - Crashes
  - Timeouts
  - Error messages
  - Data flow anomalies
- Bug detectors
  - Use instrumentation
  - Taint analysis
- Return Instrumentation Feedback

Instrumented (Greybox) Dynamic Analysis / Fuzzing

Instrumentation feedback improves both issue detection and input evolution

fuzz | verb.
/'fəz/

1. to make or become blurred

Fuzzing Engine

Mutated Inputs

🔎 Bug Detection

```
static final int[] MAGIC_NUMS_JPEG = {0xff, 0xd8};
//...
if (compareBytePair(MAGIC_NUMS_GIF, input)) {
   return ImageFormats.GIF;
}
else if (compareBytePair(MAGIC_NUMS_JPEG, input)) {
   return ImageFormats.JPEG;
}
return ImageFormats.UNKNOWN
//...
```

1. Jazzer | Coverage-guided, in-process fuzzing for the JVM (by Code Intelligence)

Instrumented (Greybox) Dynamic Analysis / Fuzzing

Instrumentation feedback improves both issue detection and input evolution

fuzz | verb.
/'fəz/

1. to make or become blurred

Fuzzing Engine

Mutated Inputs

Feedback

🔎 Bug Detection

```
static final int[] MAGIC_NUMS_JPEG = {0xff, 0xd8};
//...
if (compareBytePair(MAGIC_NUMS_GIF, input)) {
   return ImageFormats.GIF;
}
else if (compareBytePair(MAGIC_NUMS_JPEG, input)) {
   return ImageFormats.JPEG;
}
return ImageFormats.UNKNOWN
//...
```

1. Jazzer | Coverage-guided, in-process fuzzing for the JVM (by Code Intelligence)

Instrumented (Greybox) Dynamic Analysis / Fuzzing

Instrumentation feedback improves both issue detection and input evolution

fuzz | verb.
/'fəz/

1. to make or become blurred

Fuzzing Engine

0xFF0xD8...

Mutated Inputs

Feedback

🔎 Bug Detection

```
static final int[] MAGIC_NUMS_JPEG = {0xff, 0xd8};
//...
if (compareBytePair(MAGIC_NUMS_GIF, input)) {
   return ImageFormats.GIF;
}
else if (compareBytePair(MAGIC_NUMS_JPEG, input)) {
   return ImageFormats.JPEG;
}
return ImageFormats.UNKNOWN
//...
```

Instrumented (Greybox) Dynamic Analysis / Fuzzing

Instrumentation feedback improves both issue detection and input evolution

fuzz | verb.
/'fəz/

1. to make or become blurred

Fuzzing Engine

Mutated Inputs →

← Feedback

🔎 Bug Detection

```
static final int[] MAGIC_NUMS_JPEG = {0xff, 0xd8};
//...
if (compareBytePair(MAGIC_NUMS_GIF, input)) {
   return ImageFormats.GIF;
}
else if (compareBytePair(MAGIC_NUMS_JPEG, input)) {
   return ImageFormats.JPEG;
}
return ImageFormats.UNKNOWN
//...
```

What is Required for Smart Fuzzing?

Definition for a system under test

📦 Unit Testing

Test Cases written with traditional frameworks as jUnit and Mockito that developers already know and use

```
@Test
public void myUnitTest() {
    // Call your API using concrete inputs
    // and check for fixed results
    int result = callMyInternalAPI("fixed data");
    assertEquals(expectedResult, result);
}
```

- Explicit test cases
- Deterministic results
- Limited by developer's imagination

🔀 Fuzz Testing

Fuzz Targets / Test Harnesses integrated into jUnit, extending the familiar with security testing

```
@FuzzTest
public void myFuzzTest(AnyType generatedData) {
  // Generated data to iteratively call the API
  // Genetic algs use feedback maximizing coverage
  // trigger deep behavior w/out previous knowledge
  callMyInternalAPI(generatedData);
}
```

- Automated input generation
- Discovers edge cases humans miss
- Finds vulnerabilities in unexplored paths
- Self-improving test coverage

What is Required for Smart Fuzzing?

Definition for a system under test

📦 Unit Testing

Test Cases written with traditional GTest or GMock developers already know and use

```
// Google Test
TEST(MyApiTest, FixedInputTest) {
    // Call your API using concrete inputs
    // and check for fixed results
    int result = callMyInternalAPI("fixed data");
    EXPECT_EQ(expectedResult, result);
}
```

- Explicit test cases
- Deterministic results
- Limited by developer's imagination

🔀 Fuzz Testing

Fuzz Targets / Test Harnesses by defining a function receiving a buffer and its size

```
// Compile with: afl-clang-fast++ fuzz_target.cpp
extern "C" int LLVMFuzzerTestOneInput(uint8_t *data,
size_t size) {
    // Generated data to iteratively call the API
    // Genetic algs use feedback maximizing coverage
    // trigger deep behavior w/out previous knowledge
    callMyInternalAPI(data, size);
    return 0;
}
```

- Automated input generation
- Discovers edge cases humans miss
- Finds vulnerabilities in unexplored paths
- Self-improving test coverage

Example | Commons Imaging

Find lots of parsing issues

📦 Unit Testing

With jUnit and Mockito

```java
@Test
public void testGetBufferedImg10() throws Exception {
  File file = TestResources.resourceToFile("t1.jpg");
  JpegImageParser prsr = new JpegImageParser();
  BufferedImage img = prsr.getBufferedImage(file);

  assertEquals(680, img.getWidth());
  assertEquals(241, img.getHeight());
  assertEquals(-16777216, img.getRGB(0, 0));
  assertEquals(-12177367, img.getRGB(198, 13));
}
```

🔀 Fuzz Testing

Testing libraries for potential security issues

```java
@FuzzTest
public void buffImgFuzzTest(ByteSourceArray input) {
  try {
    JpegImageParser prsr = new JpegImageParser();
    prsr.getBufferedImage(input, new HashMap<>());
  } catch (IOException | ImageReadException ignored) {
  }
}

// https://issues.apache.org/jira/browse/IMAGING-277 and
// https://issues.apache.org/jira/browse/IMAGING-278
```

Example | Commons Imaging

Find lots of parsing issues

📦 Unit Testing

With Googletest

```
TEST(PngParserTest, testReadPngImage) {
    FILE* fp = fopen("test_images/t1.png", "rb");
    png_structp png =
png_create_read_struct(LIBPNG_VER, NULL, NULL, NULL);
    png_infop info = png_create_info_struct(png);
    png_init_io(png, fp);
    png_read_info(png, info);

    EXPECT_EQ(680, png_get_image_width(png, info));
    EXPECT_EQ(241, png_get_image_height(png, info));
    EXPECT_EQ(PNG_COLOR_TYPE_RGBA,
png_get_color_type(png, info));
    EXPECT_EQ(8, png_get_bit_depth(png, info));
}
```

🔀 Fuzz Testing

Testing libraries for potential security issues

```
extern "C" int LLVMFuzzerTestOneInput(uint8_t *data,
size_t size) {
    png_structp png =
png_create_read_struct(LIBPNG_VER, NULL, NULL, NULL);
    if (!png) return 0;
    png_infop info = png_create_info_struct(png);
    if (!info) { png_destroy_read_struct(&png, NULL,
NULL); return 0; }

    if (setjmp(png_jmpbuf(png))) {
        png_destroy_read_struct(&png, &info, NULL);
        return 0;  // libpng error handling
    }
    png_set_read_fn(png, /* custom read from buffer */);
    png_read_info(png, info);
    png_destroy_read_struct(&png, &info, NULL);
    return 0;
}

// CVE-2015-8126, CVE-2015-8472, CVE-2016-10087...
```

Coverage-Guided / Feedback-Based Fuzzing

🎯 History

- ~1990s – Classical fuzzing: Random inputs without feedback (Miller et al., 1990).
- Early 2000s – Instrumentation arrives: Researchers start adding program instrumentation to observe code coverage
- **~2013 – AFL**: American Fuzzy Lop introduces compile-time instrumentation, genetic mutations, and edge coverage guidance.
- 2016–2020 Improved feedback: libFuzzer, honggfuzz, AFL++ and others give more than coverage and allow collision free counting
- >2020 – Further programming languages: Jazzer, Artheris, etc

Popular Fuzzing Tools

Coverage-guided fuzzers for different ecosystems

🐰 AFL++

- Community-driven fork of AFL with advanced features
- Targets: C/C++ (native binaries)
- Features: Custom mutators, QEMU mode for binary-only fuzzing

☕ Jazzer

- Coverage-guided fuzzer for the JVM
- Targets: Java, Kotlin, Scala, and other JVM languages
- Features: JUnit 5 integration, autofuzz, native code support via JNI

🐍 Atheris

- Coverage-guided Python fuzzing engine
- Targets: Python code and native extensions
- Features: Built on libFuzzer, integrates with Python's coverage module

🌐 OSS-Fuzz

- Continuous fuzzing infrastructure for open source projects
- Supports: AFL++, libFuzzer, Honggfuzz, Jazzer, Atheris
- Features: Free for open source, 1000+ projects, found 10,000+ vulnerabilities

Question

What happens when we make a code change and restart fuzzing?

Corpus Management

⚠️ The Cold Start Problem

- Random inputs rarely reach deep code paths
- Most inputs rejected by early parsing stages
- Starting from zero = weeks of wasted compute

💾 Corpus: Collective Memory

- Set of inputs each triggering unique code paths
- Persisted between fuzzing sessions
- Grows organically as fuzzer discovers new paths

🔄 Automated Corpus Management

- Remove redundant inputs (same coverage)
- Keep smallest input per unique path
- Reduces storage + speeds up mutation cycles

Question

What kind of issues does fuzzing find?

Bug Detectors

How the fuzzer knows when something goes wrong

🎯 Detection Mechanism

1. **Crash Detection**
   Segfaults, exceptions, timeouts

✂️ Simplified Code Example

```
@FuzzTest
public void buffImgFuzzTest(ByteSourceArray input) {
  try {
    JpegImageParser prsr = new JpegImageParser();
    prsr.getBufferedImage(input, new HashMap<>());
  } catch (IOException | ImageReadException ignored) {
  }
}
```

Bug Detectors

How the fuzzer knows when something goes wrong

🎯 Detection Mechanism

1. **Crash Detection**
   Segfaults, exceptions, timeouts
2. **Assertion Violations**
   assert(), custom invariants

✂️ Simplified Code Example

```
@FuzzTest
public void buffImgFuzzTest(ByteSourceArray input) {
  try {
    JpegImageParser prsr = new JpegImageParser();
    prsr.getBufferedImage(input, new HashMap<>());
    if (img != null) {
      assertTrue(
        img.getWidth()  > 0 &&
        img.getHeight() > 0 &&
        img.getWidth()  < 100_000 &&
        img.getHeight() < 100_000,
        "Image dimensions must be reasonable"
      );
    }
  } catch (IOException | ImageReadException ignored) {
  }
}
```

Bug Detectors

How the fuzzer knows when something goes wrong

🎯 Detection Mechanism

1. **Crash Detection**
   Segfaults, exceptions, timeouts
2. **Assertion Violations**
   assert(), custom invariants
3. **Differential Testing**
   Compare outputs across implementations

✂️ Code Example

```java
@FuzzTest
public void jsonParserConsistency(String in) {
  Map jacksonResult = null;
  Map gsonResult = null;

  try {
    jackResult = new ObjectMapper().readValue(in, Map.class);
  } catch (Exception ignored) {}

  try {
    gsonResult = new Gson().fromJson(input, Map.class);
  } catch (Exception ignored) {}

  assertTrue((jackResult == null) == (gsonResult == null),
    "Parsers disagree on validity: " + in);

  if (jackResult != null) {
    assertEquals(jackResult, gsonResult,
      "Parsers produced different results for: " + in);
  }
}
```

Bug Detectors

How the fuzzer knows when something goes wrong

🎯 Detection Mechanism

1. **Crash Detection**
   Segfaults, exceptions, timeouts
2. **Assertion Violations**
   assert(), custom invariants
3. **Differential Testing**
   Compare outputs across implementations
4. **Round-Trip Testing**
   Test entire round trips

✂️ Code Example

```
@FuzzTest
public void gzipDecompressionStability(byte[] in) {
  // Round-trip comp(decomp(in))==data fail because zips store
  // timestamps, compression level, etc. in the header.
  // But decompressed content must be stable:
  // decomp(comp(decomp(in)))==decomp(in)

  byte[] decomp1;
  try (z = new GZIPInputStream(new ByteArrayInputStream(in))) {
    decomp1 = z.readAllBytes();
  }

  var o = new ByteArrayOutputStream();
  try (var gzip = new GZIPOutputStream(out)) {
    gzip.write(decomp1);
  }

  byte[] decomp2;
  try (y = new GZIPInputStream(new ByteArrayInputStream(o))) {
    decomp2 = y.readAllBytes();
  }

  assertArrayEquals(decomp1, decomp2);
}
```

Bug Detectors

How the fuzzer knows when something goes wrong

🎯 Detection Mechanism

1. **Crash Detection**
   Segfaults, exceptions, timeouts
2. **Assertion Violations**
   assert(), custom invariants
3. **Differential Testing**
   Compare outputs across implementations
4. **Round-Trip Testing**
   Test entire round trips
5. **Bug Oracles**
   ????

✂️ Code Example

```java
@FuzzTest
public void sqlInjectionTest(String inp) {
  String query = "SELECT * FROM users WHERE name = '"+inp+"'";

  try (Statement stmt = connection.createStatement()) {
    stmt.execute(query);  // Check whether is tainted?
  }
}
```

Bug Oracles

How do these oracles work

Bug Oracles

How do we detect vulnerabilities during dynamic analysis?

🧩 Classic: Dynamic Taint Analysis

- Direct application of graph based (dynamic) taint analysis at runtime
- Simple rule: tainted data reaches sink → alert

🚀 Modern Oracles

- Memory Sanitizers
- Grammar-Based Detection
- Honeypot-Based Detection

Classic Bug Oracles Based on Taint Analysis

How do we detect vulnerabilities during dynamic analysis?

⛔ Limitations

- **Unknown Sanitizers**: Custom or new sanitization functions not in allowlist
- **Sanitizer Allowlist Maintenance**: Must constantly update for new frameworks/libraries
- **Context-Dependent Sanitization**: `htmlEscape()` safe for HTML, not for SQL
- **Partial Sanitization**: Some functions sanitize incompletely

🤦 Modern Oracles

- False positives when sanitizers aren't recognized
- False negatives when sanitizers are incorrectly trusted

✂️ Code Example

```java
@FuzzTest
void sanitizerProblemDemo(String userInput) {

  // ❌ False Positive: alert, but code is safe
  String safe = myCompanyEscapeSQL(userInput);
  String q1 = "SELECT * FROM users WHERE name = '" + safe + "'";
  stmt.execute(q1);  // 🚨 ALERT! (wrong - it's safe)

  // ❌ False Negative: trust, but code is vulnerable
  String unsafe = legacyEscape(userInput);
  String q2 = "SELECT * FROM users WHERE name = '"+unsafe+"'";
  stmt.execute(q2);  // ✅ OK (wrong - it's vulnerable!)
}
```

Modern Oracles | Philosophy Shift

Three approaches to detect vulnerabilities more reliably

🚀 Modern Oracle Types

1. **Memory Sanitizers:** Detect memory corruption at runtime
2. **Grammar-Based Oracles:** Detect semantic structure changes
3. **Honeypot Oracles:** Detect successful exploitation attempts

Memory Sanitizers

Catching memory corruption through instrumentation with memory sanitizers

🎯 Problem with native languages

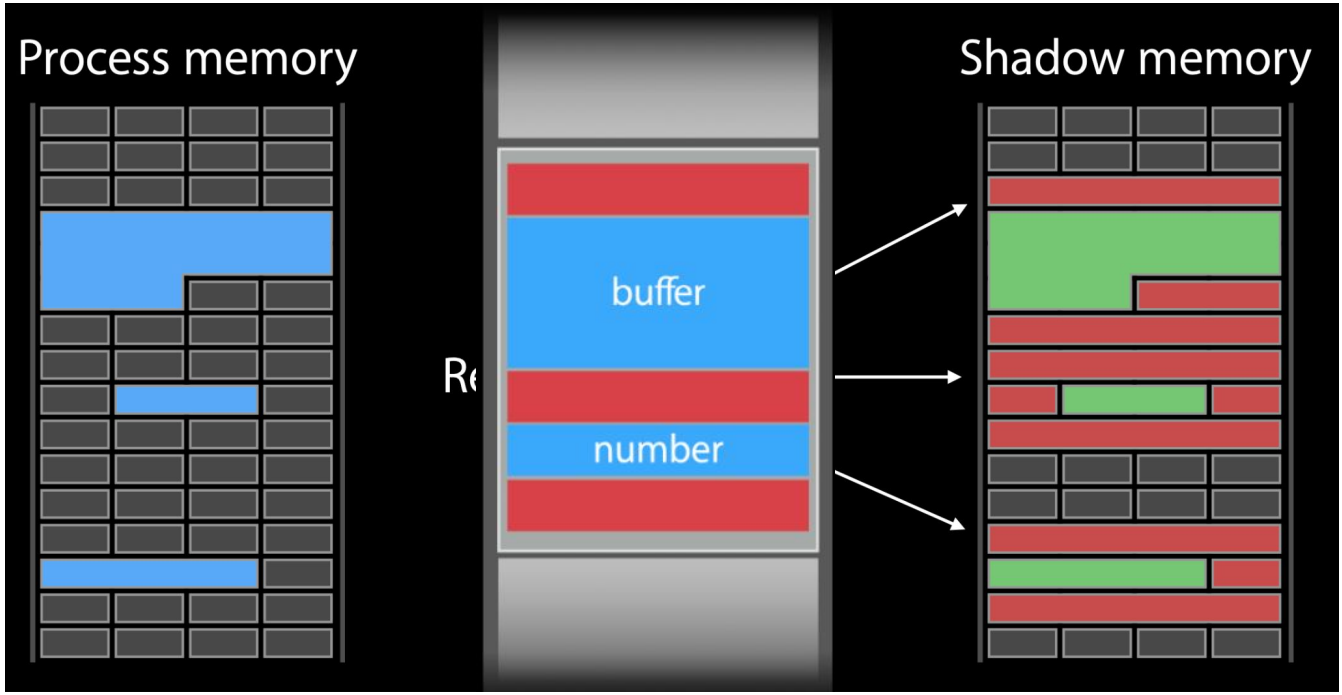
- Buffer overflows, use-after-free, uninitialized reads
- Memory-unsafe languages (C/C++) don't crash reliably
- Exploitable bugs may silently corrupt data

🎯 C/C++

- Adds "red zones" around allocated memory
- Shadow memory tracks what's accessible
- Every memory access is checked

✂️ Code Example

```
char number[8];
number[8] = 'x'; // Off-by-one: silent corruption
```



Grammar-Based Oracles

Detecting injection through semantic analysis

✂️ Code Example

```
@FuzzTest
void sanitizerProblemDemo(String userInput) {

  // ❌ False Positive
  String safe = myCompanyEscapeSQL(userInput);
  String q1 = "SELECT * FROM users WHERE name = '"+safe+ "'";
  stmt.execute(q1);  // 🚨 ALERT! (wrong - it's safe)

  // ❌ False Negative
  String unsafe = legacyEscape(userInput);
  String q2 = "SELECT * FROM users WHERE name = '"+unsafe+"'";
  stmt.execute(q2);  // ✅ OK but error in legacyEscape
}
```

🪝 Hook the Sink

- Every database query is intercepted
- Statement parsed before forwarding the original sink
- Wrong syntax means someone injected data
- If statement is correct give hints how to break it

Takeaways

Using source code and instrumentation can optimize both test generation and fault identification

✅ Improvement of Fault Detection

- Instrumentation / Interactive Application Security Testing
- Dynamic Taint Analysis

❌ Disadvantages

- Simple source code constructs can confuse DAST
- Difficulties with complex applications
- Requires good API documentation
- There is no defined end (when to stop testing)

Recap | DAST vs. SAST

SAST and DAST are complementary approaches with different strengths and weaknesses

Aspect	SAST	Classic DAST	Modern DAST
Development Phase	Early in development (IDE)	After deployment	Starting from unit tests
Access to Code	Needs the code	No code needed	Instrumentation needed
Scope	Checks patterns	Errors from outside	Errors from inside
Typical Errors	Injections, insecure practices	Injections, misconfigurations, real-world errors	Injections, misconfigurations, real-world errors
False Positives	Frequent (theoretical paths)	Low, real behavior	Lowest, real behaviour
Code Coverage	100% (even dead code)	Depends on the configuration	Automated maximization
Integration	CI/CD pipelines, IDEs	Test/Staging environments	CI/CD Pipelines, Testing from IDE
Speed	Fast, especially with small codebase	Slow and unclear when to stop	Stop when coverage converges

Key Takeaways | Securing APIs Through Testing

✅ Security testing critical for application defense

- Defense in depth utilizes multiple security testing approaches
  - Static analysis enables early detection of errors even before testing
  - Dynamic analysis provides runtime proof of real-world errors.
- Security testing should always be a proactive process

Evaluation of this lecture

Give anonymous and valuable feedback here:

https://fha8.de/eX4kP