# Informationssicherheit:

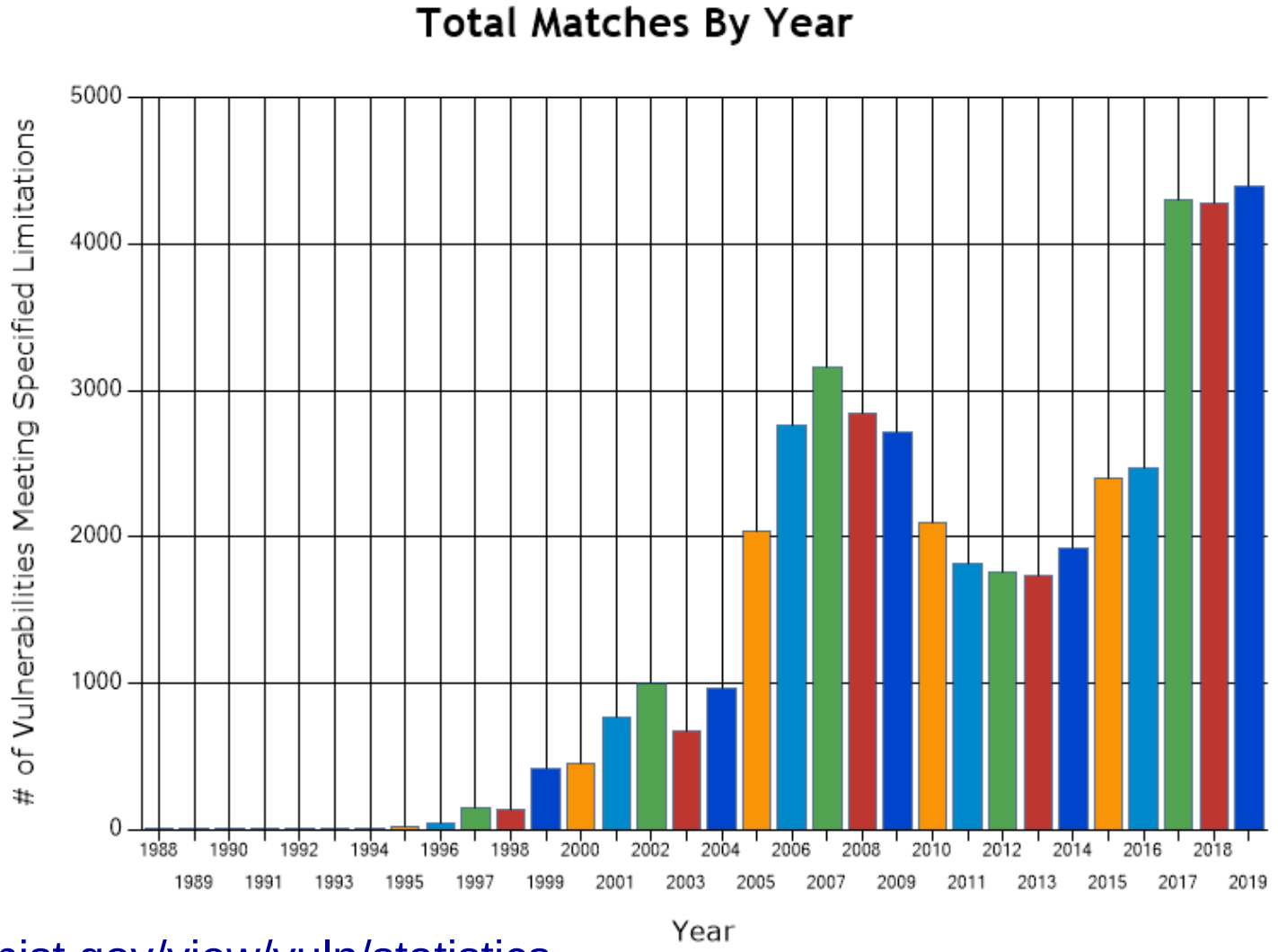
# Überblick über die Software Security

# Sicherheitslücken in Software: Heartbleed Bug

- Sicherheitslücke in OpenSSL 1.0.1 bis 1.0.1f
  - April 2014
  - ab 1.0.1g behoben
- Angreifer kann speziell fabrizierte Nachricht an Server schicken und dort unerlaubte Arbeitsspeicherzugriffe machen (Länge unter Kontrolle des Angreifers, Arraygrenzenfehler)
- Server sendet einfach Speicherinhalte an den Angreifer zurück
  - Mit privaten Schlüsseln (z.B. zum Server-Zertifikat gehörig)
  - Mit Sitzungsschlüsseln (z.B. für TLS-Kommunikation)
- OpenSSL kryptographische Bibliothek, die u.a. im Software-Stack der meisten Web-Server weltweit enthalten ist

  ☞ Hohe Angriffsfläche, Rücknahme der Schlüssel aufwendig

# Schwerwiegende Lücken




Quelle:
http://web.nvd.nist.gov/view/vuln/statistics

# Trinity of Trouble


- Gary McGraw: Software Security, Addison Wesley, 2006

- „Trinity of Trouble“
  - Steigende Komplexität (Windows 8 bis zu 80 Mio. Lines of Code?, Linux Kernel ca. 28 Mio. Lines of Code)
  - Wachsende Vernetzung (SOA, Industrie 4.0, Internet der Dinge,…)
  - Erweiterbarkeit von Systemen (Nachladen von Apps, Plugins für Browser)

# Software Security als eigene Disziplin

- Gängige Sicherheitsmechanismen wie z.B. Firewalls, Antiviren-Software oder Intrusion Detection-Systeme sind reaktiv
- Ursache für Sicherheitsprobleme: Lücken in Software
- Werkzeuge und systematische Vorgehensweisen zur Verbesserung der Software Security
  - Security Development Lifecycle (SDL)

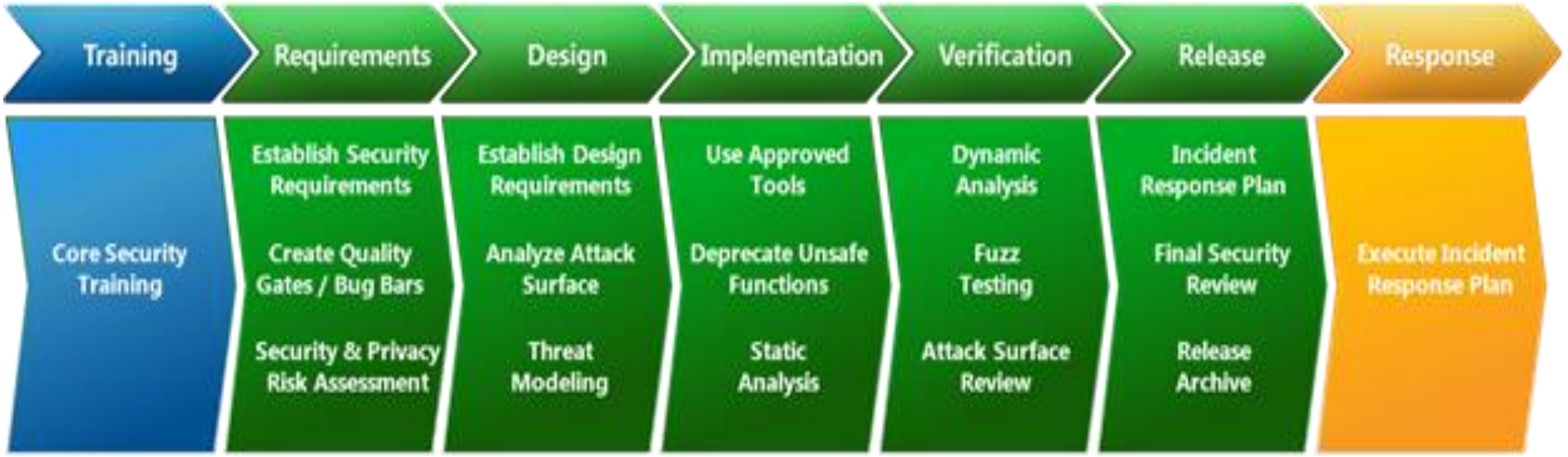
# SDL – Security Development Lifecycle

- Viele große Software-Hersteller (z.B. Microsoft, SAP, Adobe, Siemens) setzen Prozesse zur Entwicklung sicherer Software ein
- Orientierung des SDL an Software-Entwicklungsprozessen
  - Wasserfall
  - Iterative Software-Entwicklung
  - Agile Software-Entwicklung
- Umfasst alle Phasen der SW-Entwicklung und injiziert dort Security-relevante Schritte

# Beispiele für SDLs

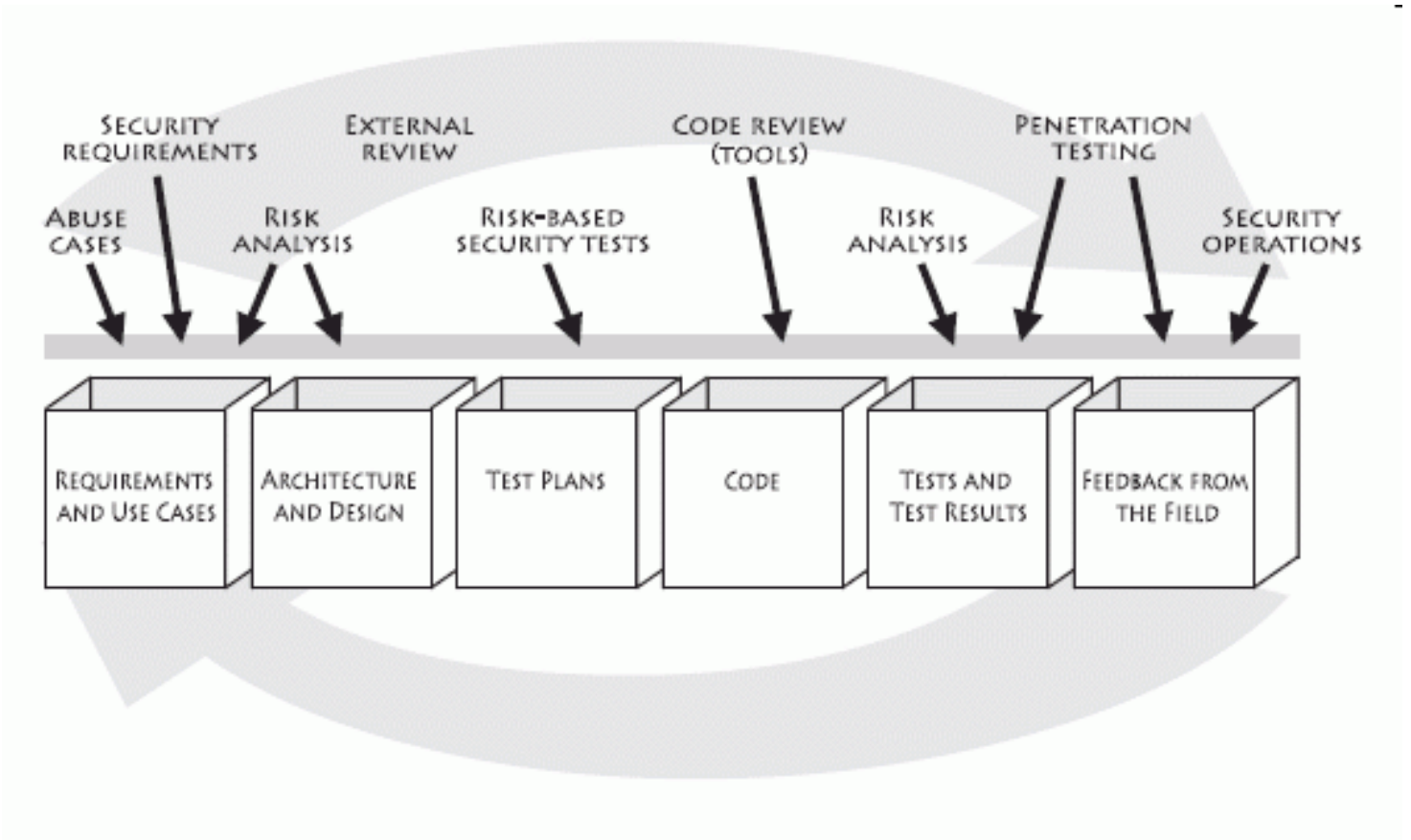
- Microsoft SDL, https://www.microsoft.com/en-us/sdl/
- Seven Touchpoints, Gary McGraw, http://www.swsec.com/resources/touchpoints/
- Beide Prozesse umfassen ähnliche Aktivitäten, die z.T. nur anders benannt sind

# SDL von Microsoft




Die Schritte werden in Abhängigkeit des SW-Entwicklungsmodells wiederholt: Zyklus!

# Seven Touchpoints nach McGraw



# Wichtige Aktivitäten innerhalb eines SDLs

1. Code Review mit Werkzeugen (vor allem statische Code-Analyse)
2. Architekturelle Risikoanalyse/Threat Modeling
3. Penetration Testing

# Wiederholung: Buffer-Overflows

- Häufigste Einbruchmethode in Server (insbesondere in Web-Server)
- Altbekanntes Problem (schon in den 60er Jahren bekannt)
- „Attack of the decade“ (Bill Gates)
- Die meisten Viren/Würmer nutzten Buffer-Overflows aus (Morris-Wurm, Code Red, Blaster)
- Ziel: Einschleusen von Code
- Ausnutzen von Programmierfehlern

# Wiederholung: Buffer Overflows

```
void trouble(){
     char line[128];
     gets(line); /* lies von stdin */
}
```

```
void trouble1(char* input){
     char line[128];
     strcpy(line, input);
}
```

# Integer Overflows

Beispiel aus *Chess, West: Secure Programming with Static Analysis, Addison-Wesley, 2007*

```
unsigned int readamt;
readamt = getstringsize();
if(readamt > 1024) return -1;
readamt--;//don‘t allocate space for ‘\n‘
buf = malloc(readamt);
```

# Probleme mit der Speicherverwaltung

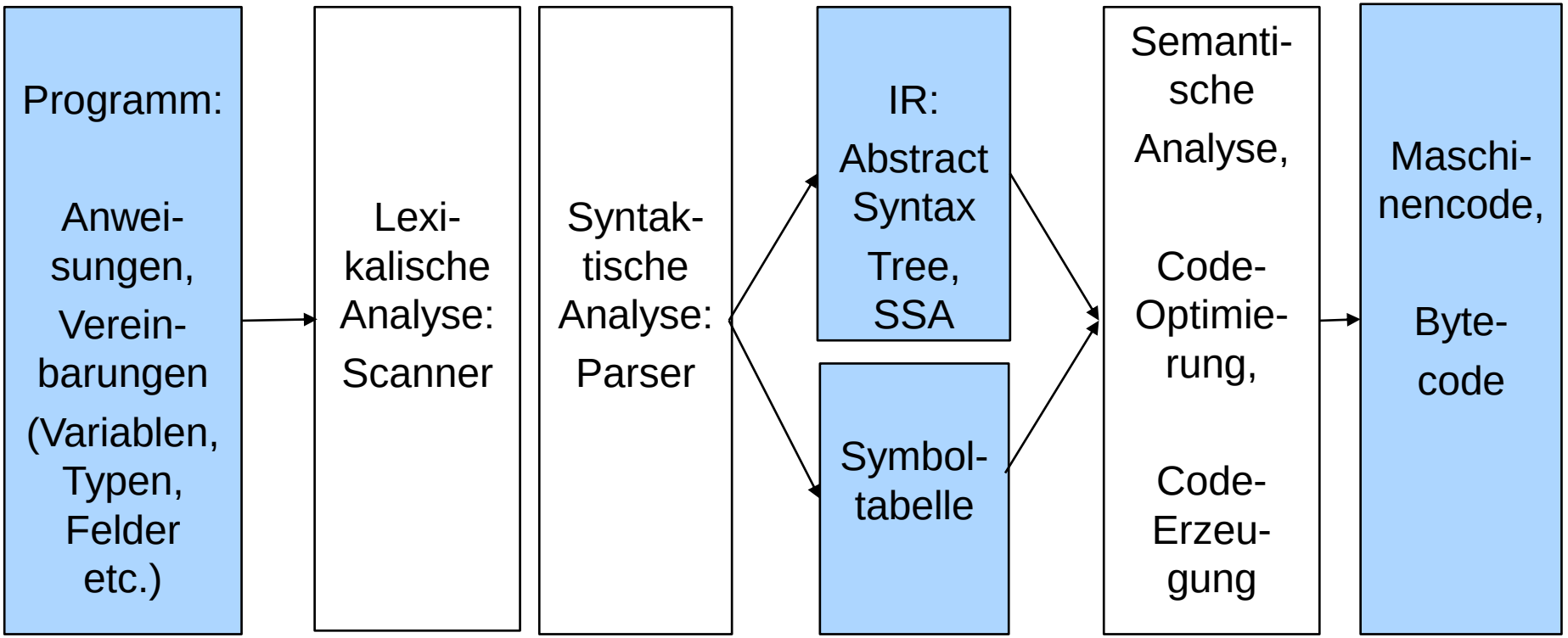
```
double_free(int i){
      int *p;
      p = (int*) malloc(42*sizeof(int));
      if(i <= 0) free(p);
      p[0] = 2;
      p[1] = 3;
}
```

Ähnliche Fehler: Arraygrenzenfehler, Speicherleaks, …

# Code Review mittels statischer Programmanalyse

- Sicherheitsanalyse des Quelltexts von Programmen
- Aufdecken von gängigen Programmierfehlern wie z.B.
  - Buffer-Overflows, Heap Overflows, Integer Overflows
  - SQL-Injection-Verwundbarkeiten
  - Cross-Site-Scripting-Verwundbarkeiten
- Automatisierte Analyse

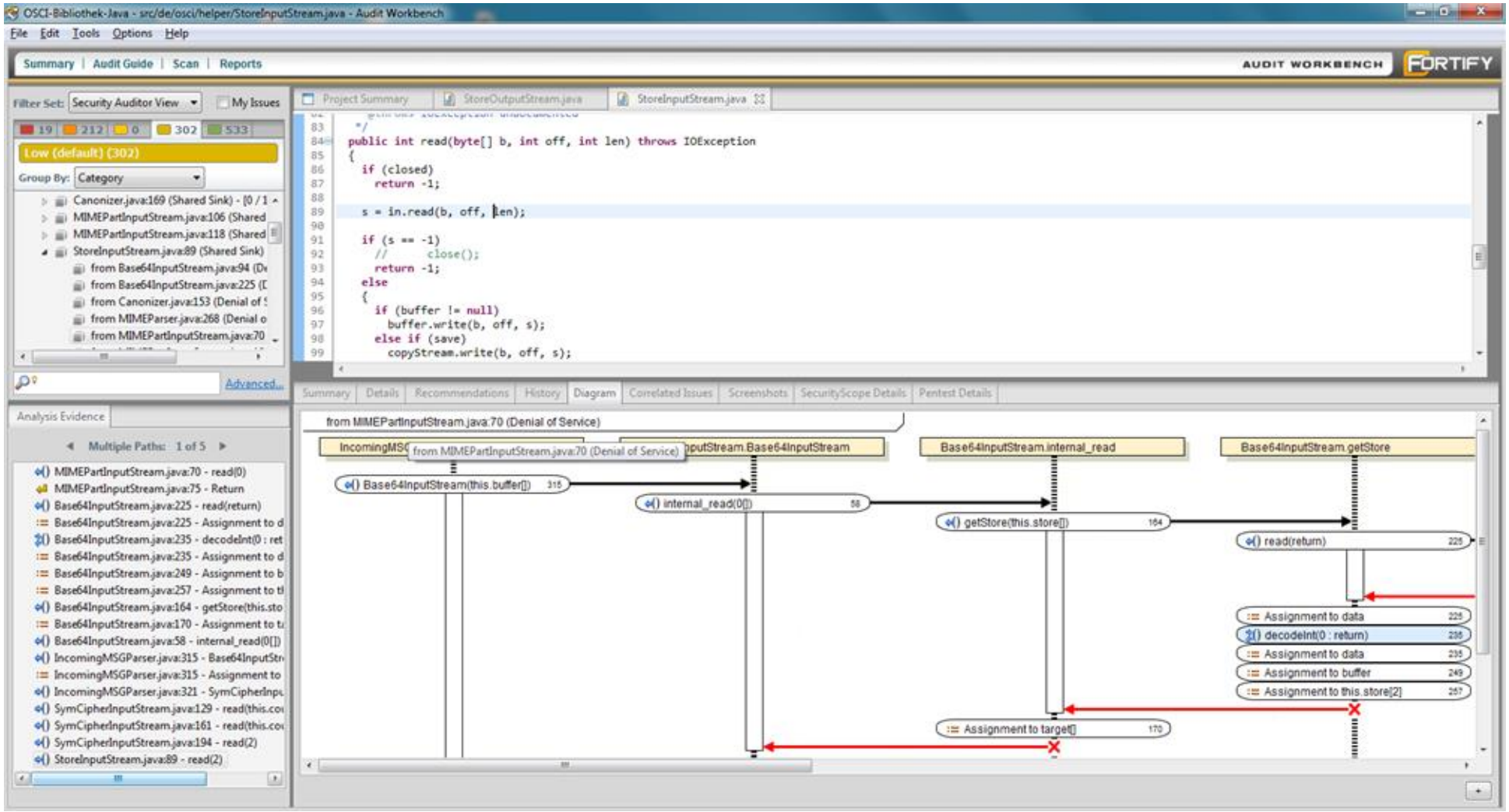
# Phasen eines Compilers



# Statische Programmanalyse

- Einsatz von Compilerbau-Techniken
  - Zwischendarstellung (IR) des Programmes z.B. durch Abstract Syntax Trees, Static Single Assignment (SSA)
  - Daten- und Kontrollflussanalysen auf Zwischendarstellungen
- False Positives (Fehlalarme), False Negatives (übersehene Fehler) aufgrund von Nicht-Entscheidbarkeit

- Gängige kommerzielle Werkzeuge: Fortify SCA (Java), IBM AppScan, Veracode, Coverity Static Analysis (für C/C++-Code), Checkmarx
- Einfache statische Analysen eingebaut in Compiler wie gcc oder Clang
- Screenshots Fortify: nächste Folie


OSCI-Bibliothek-Java - src/de/osci/helper/StoreInputStream.java - Audit Workbench
File Edit Tools Options Help
Summary | Audit Guide | Scan | Reports
AUDIT WORKBENCH
FORTIFY
Filter Set: Security Auditor View
My Issues
19 212 0 302 533
Low (default) (302)
Group By: Category
Canonizer.java:169 (Shared Sink) - [0 / 1
MIMEPartInputStream.java:106 (Shared
MIMEPartInputStream.java:118 (Shared
StoreInputStream.java:89 (Shared Sink)
from Base64InputStream.java:94 (D
from Base64InputStream.java:225 (I
from Canonizer.java:153 (Denial of S
from MIMEParser.java:268 (Denial o
from MIMEPartInputStream.java:70
Advanced...
Project Summary
StoreOutputStream.java
StoreInputStream.java
public int read(byte[] b, int off, int len) throws IOException
{
if (closed)
return -1;
s = in.read(b, off, len);
if (s == -1)
// close();
return -1;
else
{
if (buffer != null)
buffer.write(b, off, s);
else if (save)
copyStream.write(b, off, s);
Summary Details Recommendations History Diagram Correlated Issues Screenshots SecurityScope Details Pentest Details
Analysis Evidence
Multiple Paths: 1 of 5
MIMEPartInputStream.java:70 - read(0)
MIMEPartInputStream.java:75 - Return
Base64InputStream.java:225 - read(return)
Base64InputStream.java:225 - Assignment to d
Base64InputStream.java:235 - decodeInt(0 : ret
Base64InputStream.java:235 - Assignment to d
Base64InputStream.java:249 - Assignment to b
Base64InputStream.java:257 - Assignment to tl
Base64InputStream.java:164 - getStore(this.sto
Base64InputStream.java:170 - Assignment to t
Base64InputStream.java:58 - internal_read(0[])
IncomingMSGParser.java:315 - Base64InputStr
IncomingMSGParser.java:315 - Assignment to
IncomingMSGParser.java:321 - SymCipherInpu
SymCipherInputStream.java:129 - read(this.co
SymCipherInputStream.java:161 - read(this.co
SymCipherInputStream.java:194 - read(2)
StoreInputStream.java:89 - read(2)
from MIMEPartInputStream.java:70 (Denial of Service)
IncomingMS
from MIMEPartInputStream.java:70 (Denial of Service)
putStream.Base64InputStream
Base64InputStream.internal_read
Base64InputStream.getStore
Base64InputStream(this.buffer[]) 315
internal_read(0[]) 58
getStore(this.store[]) 164
read(return) 225
Assignment to data 225
decodeInt(0 : return) 235
Assignment to data 235
Assignment to buffer 249
Assignment to this.store[2] 257
Assignment to target[] 170

# Statische Programmanalyse: SonarQube


- Statisches Analysewerkzeug zur Bewertung der Code-Qualität im Allgemeinen
- Auch: Unterstützung von Security Rules: z.B. SQL-Injection- und XSS-Verwundbarkeiten, OWASP Top 10 (Liste mit gängigen SW-Schwachstellen, OWASP - Open Web Application Security Project)
- Frei verfügbare Fassung, aber auch kostenpflichtige Commercial Edition
- In der Praxis weit verbreitet
- Unterstützte Sprachen: u.a. Java, Python, Groovy, JavaScript, PHP, TypeScript, HTML, C#
- https://www.sonarqube.org/features/security/

PR + Taint Analysis (Java, PHP, JS) src/main/java/devoxx/security/injection/Servlet.java

```
26 …      }
27
28        2 String name = 1 taintedRequest.getParameter(FIELD_NAME);
29          taintedString = name;
30
31          taintedString = 3 java.net.URLDecoder.decode(taintedString, "UTF-8");
32
33          try {
34 …          new SQLInjectionVulnerability(taintedString);
35        4   new CommandInjectionVulnerability(taintedString);
36            new RegexInjectionVulnerability(taintedString);
37          } catch (SQLException|IOException e) {
```

PR + Taint Analysis (Java, PHP, JS) src/main/java/devoxx/security/injection/CommandInjectionVulnerability.java

```
12 …
13          if (path != null) {
14        5   command[command.length - 1] = path;
15          }
16
17          runtime.exec(command);
```

**Refactor this code to not construct the OS command from tainted, user-controlled data.** See Rule 3 months ago L17

Vulnerability Blocker Confirmed Not assigned 30min effort Comment cwe, owasp-a1, sans-top25-insecure

```
18
19 …      runtime.exec("rm -rf /tmp/bla");
20 …    }
21    }
22
```

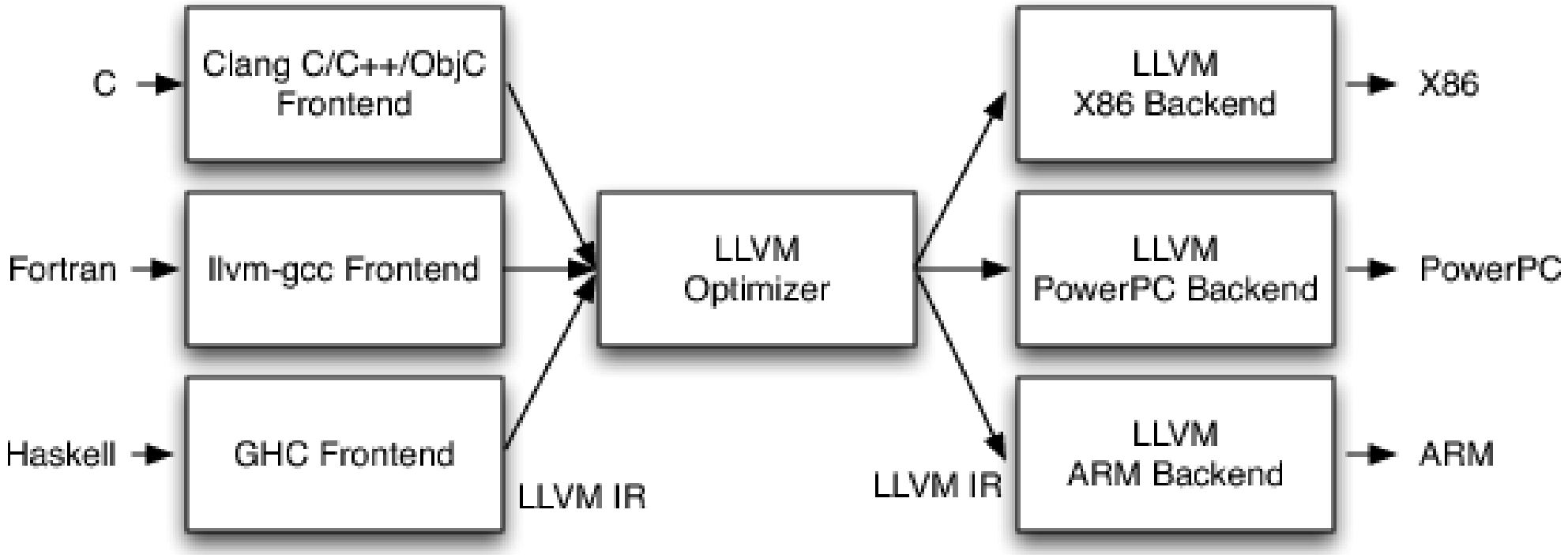
# Statische Programmanalyse mit einem frei verfügbaren Werkzeug

- Statische Sicherheitsanalysewerkzeuge oft teuer ($ € ¥ £)
- Einsatz von frei verfügbarem statischen Analysewerkzeug

- LLVM: Compiler Infrastructure
- Unterstützt verschiedene Frontends: C/C++, Objective-C, Haskell, Fortran, (Java)
- Clang Static Analyzer von LLVM:

http://clang-analyzer.llvm.org/

# Design von LLVM



# Clang Static Analyzer

- Clang: C/C++ Frontend von LLVM
- Clang Static Analyzer: Statisches Analysewerkzeug für Clang, http://clang-analyzer.llvm.org/
- Unterstützte Analysen (http://clang-analyzer.llvm.org/available_checks.html), u.a.:
  - Nullpointer-Dereferenzierung
  - Unsichere Verwendung von C/C++-Funktionen wie z.B. gets(), strcpy()
  - Arraygrenzenfehler
- Manche Analysen noch experimentell, recht eingeschränkter Regelsatz

# Beispiel für eine zu prüfende Sicherheitsregel

**Regel:** alpha.security.ArrayBoundV2 (C):
Warn about buffer overflows (newer checker).

```
void test() {
      char *s = "";
      char c = s[1]; // warn
}
```

```
void test() {
      int buf[100];
      int *p = buf;
      p = p + 99;
      p[1] = 1; // warn
}
```

# Statische versus dynamische Programmanalyse

- **Statische Analyse:** Durchführung der Programmanalyse auf Quelltext oder Zwischendarstellung (IR) ohne Ausführung des Programms, z.B. zur Compilezeit

- **Dynamische Analyse:** Durchführung der Analyse im laufenden Betrieb der Software. (Beispiel mittels Code-Instrumentierung, s. nachfolgende Folien)

# Dynamische Analyse mittels Code-Instrumentierung

- Idee:

  Compiler setzt zusätzliche Sicherheitsüberprüfungen beim Verwenden von kritischen Funktionen wie strcpy(), sprintf(), strcat(), malloc(), free() in den Code ein
- Fachbegriff hierfür: Code-Instrumentierung
- Vorteil: Keine oder wenige False Positives, Analyseergebnis genauer
- Nachteil: Code ist größer und weniger effizient
- Anderes Beispiel für Code-Instrumentierung: Canaries
  - Einfügen von Canaries in den Unterprogrammstack durch Compiler

# Beispielwerkzeug: AddressSanitizer

- Auf Basis von Code-Instrumentierung: Google AddressSanitizer (oder ASan); *to sanitize* - reinigen
  https://github.com/google/sanitizers/wiki/AddressSanitizer
- Memory Error Detector für C/C++
- Durchschnittliche Verlangsamung des instrumentierten Programmes: Faktor 2
- Run-time library ersetzt die malloc() und free() Funktionen.
- Integration in LLVM/Clang (ab Version 3.1) und auch gcc (ab Version 4.8)
- Aufruf: -fsanitize=address

# Beispielwerkzeug: AddressSanitizer

- Unterstützte Checks, z.B.:
  - Use after free (dangling pointer dereference)
  - Heap buffer overflow
  - Stack buffer overflow
  - Use after return
  - Use after scope
  - Initialization order bugs
  - Memory leaks

# Fehlerhaftes Programm: Zugriff auf Variable nach Speicherfreigabe

```
#include <stdlib.h>
int main() {
  char *x = (char*)malloc(10 * sizeof(char*));
  free(x);
  return x[5];
}
```

% ../clang_build_Linux/Release+Asserts/bin/clang -fsanitize=address -O1 -fno-omit-frame-pointer -g tests/use-after-free.c

# Ausgabe von AddressSanitizer

```
% ./a.out
==9901==ERROR: AddressSanitizer: heap-use-after-free on address 0x60700000dfb5 at pc 0x45917b bp
0x7fff4490c700 sp 0x7fff4490c6f8
READ of size 1 at 0x60700000dfb5 thread T0
    #0 0x45917a in main use-after-free.c:5
    #1 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
    #2 0x459074 in _start (a.out+0x459074)
0x60700000dfb5 is located 5 bytes inside of 80-byte region [0x60700000dfb0,0x60700000e000)
freed by thread T0 here:
    #0 0x4441ee in __interceptor_free projects/compiler-rt/lib/asan/asan_malloc_linux.cc:64
    #1 0x45914a in main use-after-free.c:4
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
previously allocated by thread T0 here:
    #0 0x44436e in __interceptor_malloc projects/compiler-rt/lib/asan/asan_malloc_linux.cc:74
    #1 0x45913f in main use-after-free.c:3
    #2 0x7fce9f25e76c in __libc_start_main /build/buildd/eglibc-2.15/csu/libc-start.c:226
SUMMARY: AddressSanitizer: heap-use-after-free use-after-free.c:5 main
```

# Nicht nur Bugs, sondern auch Flaws

- Designfehler (Flaws) 50% aller Sicherheitsprobleme von Software
- Beispiele
  - Rollenbasierte  Zugriffskontrolle inkonsistent über mehrere Schichten einer Multi-Tier-Anwendung hinweg
  - Falscher Einsatz von Krypto (Schlüssellänge zu klein)
  - Schutz der Integrität von Daten, wenn eigentlich Vertraulichkeit benötigt
- Lösung: Führe Architekturelle Risikoanalyse durch

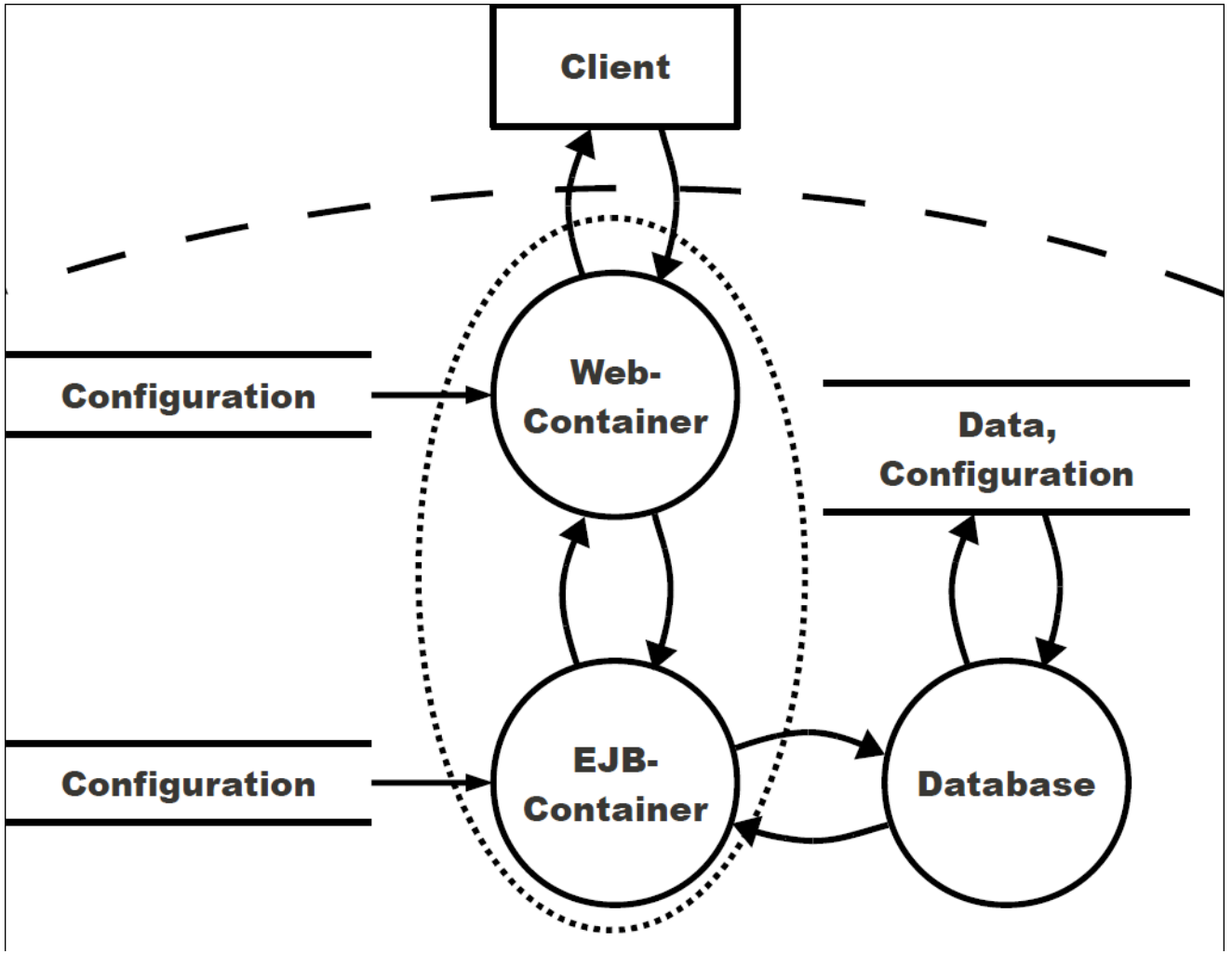
# Architekturelle Risikoanalyse

- Aktivität in der Designphase von Software
  - Später gefundene Lücken teuer zu beheben
- MS-SDL: Threat Modeling, G. McGraw: Architekturelle Risikoanalyse (ARA)
- Grobe Idee:
  1. Erstelle Diagramme der Software
     - Forest-Level-Übersicht, nicht Tree Level
  2. Diskutiere anhand des Bildes systematisch Design-Schwächen (manueller Prozess!)

# Weitere Schritte der architekturellen Risikoanalyse nach McGraw

1. Durchsuchen der Literatur nach bekannten architekturellen Sicherheitsproblemen (z.B. CWE – Common Weakness Enumeration, CAPEC - Common Attack Pattern Enumeration and Classification )
2. Identifizierung von Sicherheitsproblemen verwendeter Frameworks
3. Zusammensetzen in einer Gruppe, um unbekannte Risiken zu identifizieren
   - Parallelisierung
   - Forest-Level-Abbildung hilfreich

# Threat Modeling/ARA  mit Datenflussdiagrammen



# Penetrationstests

- Integration in SW-Entwicklungsprozess
  - Ergebnisse der architekturellen Risikoanalyse nutzen – zielgerichtetere Pen-Tests
  - Ergebnisse der Pen-Tests in den SW-Entwicklungszyklus einbringen;
  - Nicht nur das offensichtliche Sicherheitsproblem beheben:

    *Declare victory and go home*

- Nicht als erste Sicherheitsüberprüfung im SW-Entwicklungsprozess, letzter Test vor dem Deployment!

- Werkzeuge nutzen

# Software Penetrationstests – Werkzeuge nutzen

- Reverse Engineering mittels Disassembler (z.B. IDA - www.hex-rays.com/idapro/, Ghidra - https://ghidra-sre.org/) und Debugger
- Dynamic Instrumentation Toolkit: Frida (https://frida.re/)
- Testen von Web-Anwendungen
  - OWASP Zed Attack Proxy Project; jetzt von Checkmarx weiterentwickelt: https://github.com/zaproxy/zaproxy
  - Burp-Suite (https://portswigger.net/burp/); dynamisches Testen von Web-Anwendungen inkl. MITM-Angriffen
  - MicroFocus (früher HP) WebInspect – dynamisches Testen von Web-Anwendungen
  - Fuzzer (nächste Folie)

# Fuzzing

- Wichtiges Hilfsmittel für Penetrationstests
- **Idee:** Zufällig fehlerhafte Daten erzeugen und dann (Web-) Applikation automatisiert damit testen
- Man kann Fuzzer auch für spezielle Anwendungen und Protokolle schreiben
  - Sogar für Automobilelektronik
  - (in Masterarbeit an der Uni Bremen) für App-gesteuerte Alarmanlagen
- Liste von Fuzzern auf den OWASP-Webseiten:

  https://www.owasp.org/index.php/Fuzzing

# Beispiel: american fuzzy lop

- **Problem:** Wie Codeabdeckung eines Fuzzers erhöhen?
- Lösung für C/C++-Programme: american fuzzy lop, Open Source
  https://lcamtuf.coredump.cx/afl/
- Entwickler: Michal Zalewski (Google)
- Ansatz:
  - Erzeugt aus einer kleinen Menge von Testsamples automatisiert viele Fälle
  - Testfälle **passen zur erwarteten Eingabe** des Analysekandidaten
  - Hierzu: Instrumentierung des Analysekandidaten (Quellcode, Binaries), Genetische Algorithmen zur Testfallerzeugung (Kontrollfluss des Analysekandidaten berücksichtigen)
- Generierte Testfälle können auch als Eingabemenge für andere Fuzzer verwendet werden

# Animation: american fuzzy lop

- https://lcamtuf.coredump.cx/afl/rabbit.gif
- Zum Ausprobieren
- Hintergrund:
  - Ziel: Testen des Programmes ImageMagick
  - Startpunkt das afl-Logo (als gif)
  - Darstellung aller hieraus generierten Testfälle als Animation
  - Testfälle werden von ImageMagick akzeptiert

# Forschung am TZI: ML-SAST

- Forschungsauftrag vom BSI an die Universität Bremen (TZI, AG Softwaretechnik) und neusta mobile solutions GmbH

Bundesamt für Sicherheit in der Informationstechnik

- ML-SAST– Machine Learning im Kontext von Static Application Security Testing: Statische Code-Analyse mit Maschinellem Lernen verknüpfen
- Laufzeit: ca. zwei Jahre: 12/2020 – 11/2022
- Auftragssumme: knapp 500.000 €

neusta mobile solutions

# Ziele von ML-SAST

- Stand der Technik systematisch erheben durch:
  - Experteninterviews
  - Analyse vom am Markt erhältlicher SAST-Werkzeuge
  - Literature Mapping-Studie über den Stand der Forschung
- Studie zugreifbar unter:

  https://www.bsi.bund.de/DE/Service-Navi/Publikationen/Studien/ML-SAST/ml-sast_node.html

- Entwicklung eines eigenen Prototyps:
  - Mit statischen Programmanalysemethoden Programmpfade ermitteln
  - Dann den Abstand zu gutem oder schlechtem Verhalten bestimmen (aus Trainingsdaten zuvor ermittelt)

# ML-SAST-Prototyp