DEEPSEC

Secure Design & Secure Coding
Wie schaut sicherer Code aus?

René Pfeiffer, DeepSec In-Depth Security Conference

Linuxwochen Eisenstadt 2021, 24. April 2021

DEEPSEC

- René Pfeiffer
- Senior Sysadmin, Vortragender & Security Consultant
- Theoretische Physik
- Internet User seit 1992
- Config, Code, Crypto, Chaos
- DeepSec & DeepINTEL Konferenzen

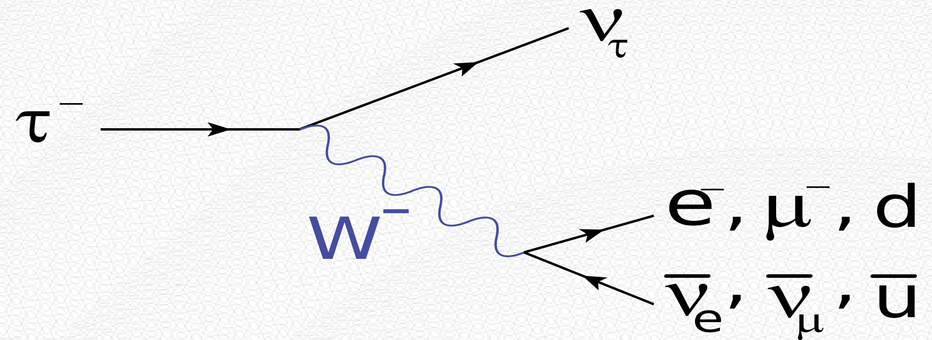


Bild: τ Lepton Zerfall, Quelle https://commons.wikimedia.org/wiki/File:Feynman_diagram_of_decay_of_tau_lepton.svg

Software - Geschichte

- Algorithmen (Abu Dschaᶜfar Muhammad ibn Musa Chwārizmī)
- Ada Lovelace – 19. Jahrhundert
- Digitalisierung – 1941 (Z3, Deutsche Luftwaffe)
- Creeper Test Virus – 1971 (DEC PDP-10, ARPANET)
- Reaper Test Cleaner (sucht & löscht Creeper Virus)
- Elk Cloner – 1982 (Macintosh, erster Computervirus)

Sicherheit

- Lange (vor-digitale) Geschichte
- Sicherheit / Security
  - Stabilität – keine Abstürze, Code verhält sich vorhersehbar
  - Verwandt mit Schutz & Sicherung (security ↔ safety)
  - Software/Code soll
    - sich „wohl verhalten“ und
    - sich selbst schützen !!

Daten?

DEEPSEC

- Code ≠ Daten, Daten ≠ Code – selten 😉
  - Von-Neumann-Architektur der Computer
  - Daten werden oft interpretiert
- Annahmen
  - Software steckt voller Vermutungen
  - Code benötigt klare interne Zustände
- Buzzwords und Trends 🦄
  - Cloud, Virtual, Hyper, Blockchain, holistisch, …
  - CI, [Dev|Sec|Ops|Ups|Aha|Tst|Out], Agile, …

Secure Coding

- Secure Coding
  - soll Sicherheit adressieren und Bugs vermeiden
  - >90% basierend auf Geisteshaltung, nicht Tools!
- Secure Coding = zusätzlicher Code → „weakest link“
- Verfügbar für Betriebssysteme und Applikationen
- Legacy Code und ausgewählte Plattformen fallen raus
  - überlieferte Bibliotheken
  - Firmware (proprietäre Blobs)
  - diverse Spezialfälle

- Trugschlüsse
  - Address Space Layout Randomisation (ASLR)
  - Buffer/Stack Overflow Protection
  - No-eXecute (NX) Technologien
- → Air Bags für Software/Code
  - Technologien wirken *nach* dem Versagen
  - Beheben keine kritischen Bugs
  - Dämmen Schaden bestenfalls ein
  - Applikation versagt trotzdem

Secure Coding Umsetzung

DEEPSEC

- Code Base mit Secure Coding Checklisten vergleichen
- Volltextsuche nach gefährlichen Schlüsselworten
- Suchen und Ersetzen nach Checklisten
- Kryptographie hinzufügen (!?)
- Authentisierung/Autorisierung hinzufügen (!!)
- Neu übersetzen / Analyzer bemühen, neues Deployment
- Im Zweifel auf „nur für internen Gebrauch“ hinweisen
- #win #marketing #release #upgrade

Secure Coding - Eigenheiten

- Anleitungen / Checklisten sind hilfreich
- Air Bags und Brandschutztüren sind nützlich
- Zusätzlich notwendig:
  - Eigenheiten der Plattform(en) kennen
  - Komponenten *gründlich* prüfen
  - Code möglichst hinterhältig testen

DEEPSEC

```php
<?php
$foo = "0";                          // ist ein String (ASCII 48)
$foo += 2;                           // ist nun ein Integer (2)
$foo = $foo + 1.3;                   // ist nun Fließkomma (3.3)
$foo = 5 + "10 Little Piggies"; // ist wieder Integer (15)
$foo = 5 + "10 Small Pigs";      // bleibt Integer (15)
?>
```

Feature ist nicht auf PHP beschränkt. “Hilfreiche” Typkonversionen findet man oft in Skriptsprachen. Jeder Code mit n>10 Zeilen muss starke Datentypen verwenden oder ganz genau prüfen.

Horrow Show – Out of Bounds

DEEPSEC

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    int mark[5] = { 19, 10, 8, 17, 9 }; // wie in C
    vector <int> vect { 1, 2, 3, 4, 5 }; // „neues“ C++ Feature 😈

    int n = vect.size();
    cout << "Size of the vector is :" << n << endl;
    cout << "mark[4] = " << mark[4] << endl;
    cout << "mark[5] = " << mark[5] << endl;
    cout << "vect[4] = " << vect[4] << endl;
    cout << "vect[5] = " << vect[5] << endl;
    cout << "vect[4] = " << vect.at(4) << endl;
    cout << "vect[5] = " << vect.operator[](5) << endl;
    cout << "vect[5] = " << vect.at(5) << endl;
}
```

Exkurs: Undefined Behaviour

DEEPSEC

- Konstrukte können kompletten Code „ungültig“ machen
- In C beispielsweise:

- Verwenden von nicht intialisierten Variablen oder Datenstrukturen.
- Überlauf von vorzeichenbehafteten Ganzzahlen.[3]
- Schiebeoperationen um mehr Binärstellen als der zu schiebende Typ Bits hat.
- Verwenden von Zeigern mit zufälligem oder undefinierten Inhalt (*wild pointer*).
- Verwenden von NULL Zeigern (*NULL pointer*), auch Nullzeiger-Dereferenzierungen (*null pointer exceptions*).
- Verletzen von Typregeln (beispielsweise der Cast von einem `int` Zeiger zu einem `float` Zeiger und Ansprechen des Inhalts).

Gilt auch teilweise für andere Sprachen oder Szenarien, in denen man programmiert.

- Analogie aus der Mathematik:

Satz: $2 = 1$

Beweis:

1. Es gelte : $a = b$
2. Multiplikation mit a : $a^2 = a b$
3. Addieren von $a^2 - 2 a b$: $a^2 + (a^2 - 2 a b) = a b + (a^2 - 2 a b)$
4. Zusammenfassen : $2 (a^2 - a b) = a^2 - a b$
5. Eliminieren von $a^2 - a b$: $2 = 1$

Quelle: https://de.wikipedia.org/wiki/Trugschluss_(Mathematik)

Horrow Show – Integer Overflow

- Mathematisch immer korrekt:
  - x+1 > x
- In Code immer ein potentieller Überlauf
  - Datentypen sind beschränkt
  - Analog x > x-1
  - y[i+1]
  - n++
  - m--

DEEPSEC

- Annahme Server hat folgenden Java™ Code

```
@Bean(name = "/service/toupper")
RemoteExporter exporterServiceRemote() {
  var exp = new HttpInvokerServiceExporter();
  exp.setServiceInterface(UpperCaseService.class);
  exp.setService((UpperCaseService)String::toUpperCase);
  exp.setAcceptProxyClasses(false);
  return exp;
}
```

Quelle: https://ttulka.medium.com/insecure-deserialization-explained-with-examples-in-java-b599b662599f

DEEPSEC

- Client schickt folgenden Java™ Code an Server:

```
static  {
  try {
    Runtime.getRuntime()
        .exec("touch /HACKED");
  } catch (IOException e) { }
}
```

→ serialisiert in exploit.ser Datei

```
curl -XPOST -H "Content-Type: application/x-java-serialized-object" \
 http://127.0.0.1:8080/service/toupper --data-binary @exploit.ser
```

Quelle: https://ttulka.medium.com/insecure-deserialization-explained-with-examples-in-java-b599b662599f

Horrow Show – unbedarfte Parser

DEEPSEC

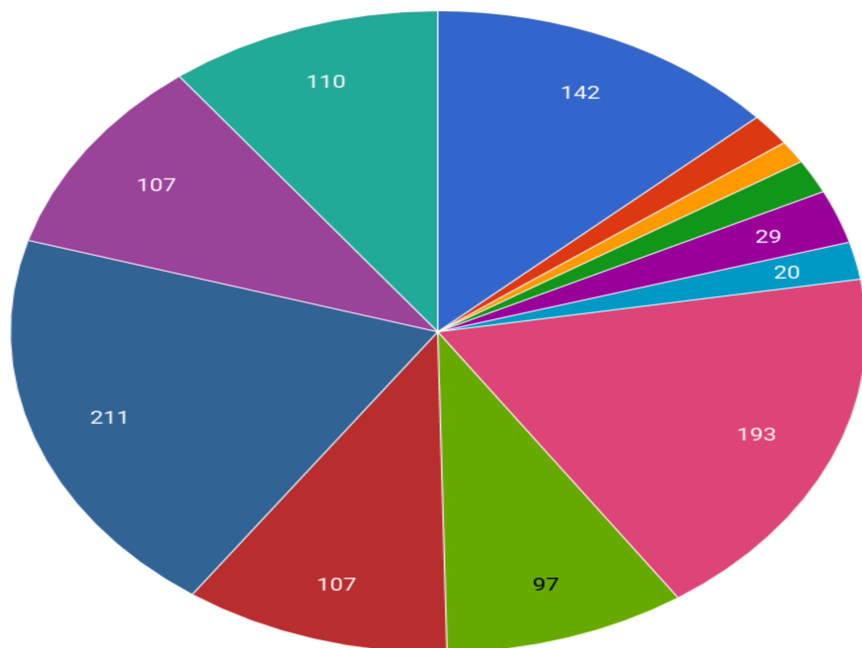
XML Billion Laughs Attack

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ELEMENT lolz (#PCDATA)>
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

Quelle: https://en.wikipedia.org/wiki/Billion_laughs_attack

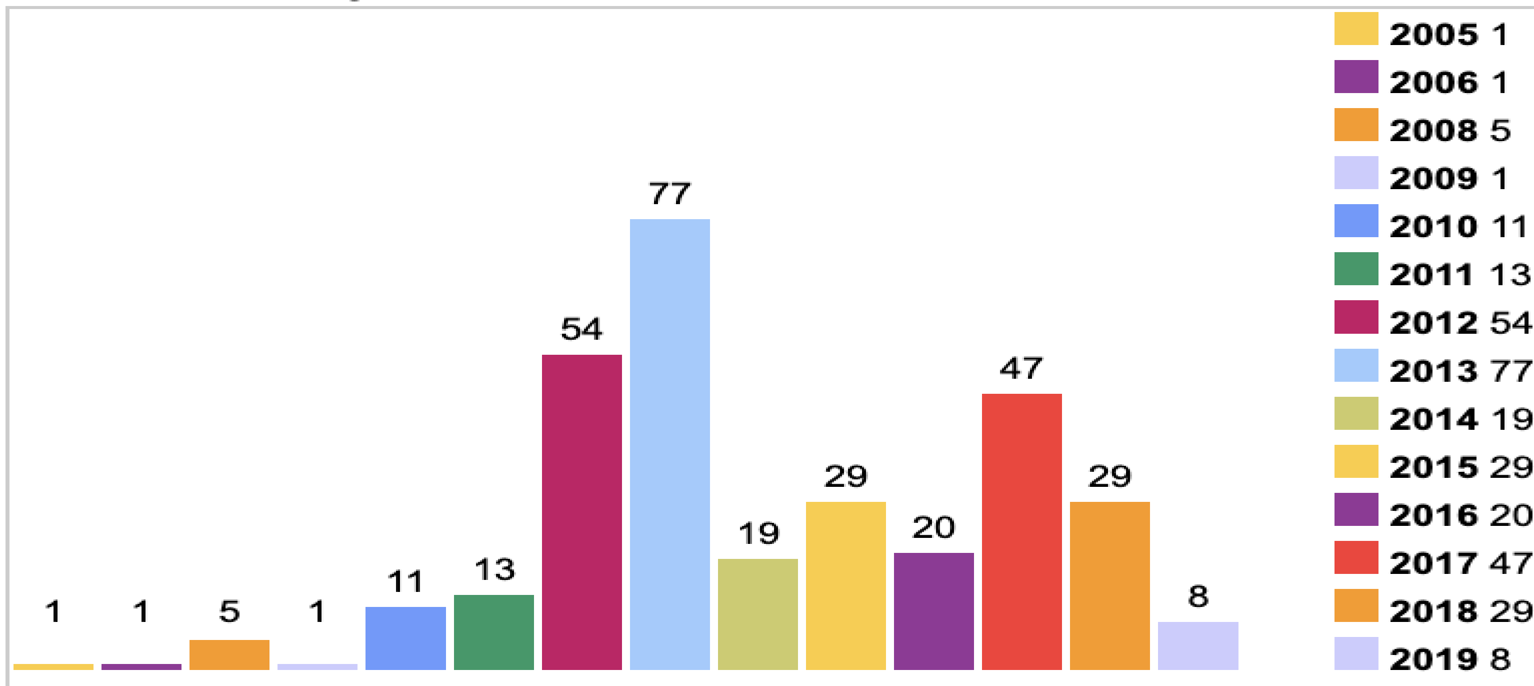
- “Ja, aber meine Programmiersprache ist von Haus aus sicher!”
- Nein,weil Code
  - mit Komponenten interagiert,
  - Netzwerke verwendet und
  - Daten speichern/lesen muss.

- 2020 CWE Top 25 Most Dangerous Software Weaknesses
- OWASP Top 10 Web Application Security Risks
- Google Project OSS Fuzz:



Komponenten

DEEPSEC



Quelle: https://signal.org/blog/cellebrite-vulnerabilities/

ffmpg Bibliothek

Secure Design

Secure Design Vorbereitungen

DEEPSEC

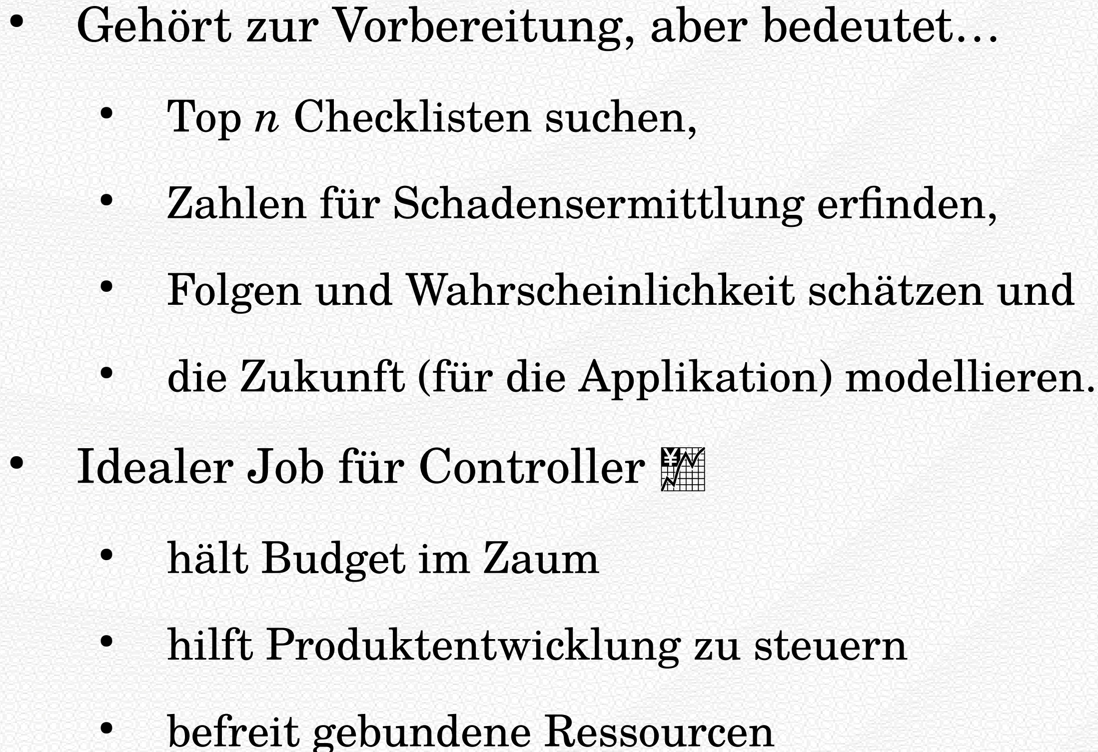
- Klassifizierung ✍
  - Welche Daten werden verarbeitet? Wie schauen sie *genau* aus?
  - Wer produziert/konsumiert Daten?
- Übergänge identifizieren
  - Welche Grenzen überschreiten die Daten?
  - Welche Teile kommunizieren womit? Wie? Warum?
  - Wiederholung für interne Übergänge
- Interne Zustände verfolgen
  - Kryptographische Algorithmen müssen Zustände beschreiben

Kenne die Plattform!

- Hardware 🔌
  - Alle Prozessoren enthalten Fehler
  - x86/x86_64 CPUs untereinander inkompatibel (-march=native)
  - Features ändern Verhalten von Code
- Compiler / Interpreter / JIT Konstrukte
- Betriebssysteme 💻
  - Was passiert wirklich mit Daten im Speicher?
- Bibliotheken 📜

Bedrohungsanalyse

DEEPSEC



- Gehört zur Vorbereitung, aber bedeutet…
  - Top n Checklisten suchen,
  - Zahlen für Schadensermittlung erfinden,
  - Folgen und Wahrscheinlichkeit schätzen und
  - die Zukunft (für die Applikation) modellieren.
- Idealer Job für Controller 💹
  - hält Budget im Zaum
  - hilft Produktentwicklung zu steuern
  - befreit gebundene Ressourcen

Secure Design Prinzipien (§)

- Angriffsoberfläche minimieren
- Sichere Defaults einführen, behalten und benutzen
- *Principle of least Privilege*
- *Principle of Defence in Depth*
- Sicher versagen – mehr als nur Exceptions verwenden
- Diensten / Dritten nicht vertrauen
- Funktionstrennung (*separation of duties*)
- Keine Security durch Obscurity
- Keep it simple 😸

Secure Design Konzepte

- Secure Design ist Teil des Konzepts, nicht des Codes!
  - Beginn (weit) vor der ersten Zeile Code
- Secure Design impliziert Secure Coding
  - Prinzipien erfordern sichere Implementierung
  - unsichere Komponenten dürfen nicht teilnehmen
  - ausgedehnt auf Protokolle und Datenformate
- Secure Design ist Bestandteil aller Schritte
  - Konzept, Design (auch UI!), Code, Testen, Deployment, …
  - Daten! - Datenverarbeitung und -haltung

Secure Design – Bad News!

DEEPSEC

- Defekte Architekturen sind nicht reparierbar ☢ ☣ ⚠
  - Hardwarefehler, Protokolle, Plattformen, Toolchains, …
- Workarounds beginnen 10 Minuten nach dem Design 🛠
  - Komplexe Datenformate (PDF, Office Dokumente, Multimedia, …)
  - Anforderungen (PKI, Crypto, Schlüssel, Speicherplatz, CPU, …)
  - Komponenten (Container all teh things!😂)
- Abhilfe: Gleich mit Fehlern beginnen! 😈

Failure Development

- Instabilität kann in Code durch
  - manipulierte Daten oder
  - logische Konstrukte eingeführt werden.
- Code muss elegant versagen ☠
  - Kein undefiniertes Verhalten
  - Arbeiten unter Last
  - Arbeiten mit fehlenden Ressourcen

Linux® Kernel Fault Injection

- Linux® Kern erlaubt Injektion von Fehlern
- Funktionen (*system calls*) Fehlercodes liefern
  - Intervall und Wahrscheinlichkeit konfigurierbar
  - Verfügbar für Storage, Dateisysteme, bestimmte Subsysteme
- Nützlich für Debugging
- Leicht zu verwenden
  - Teil des Standardkerns, frei verfügbar
  - Kann via `debugfs` oder `procfs` gesteuert werden

DEEPSEC

Beispiel: Fault Injection

- `mount debugfs /debug -t debugfs`
- `cd /debug/fail_make_request`
- `echo 10 > interval` # Intervall
- `echo 100 > probability` # 100% Wahrscheinlichkeit
- `echo -1 > times` # Wie oft: -1 bedeutet kein Limit
- `mount -t logfs /dev/sdb1 /mnt/2`
- `echo 1 > /sys/block/sdb/sdb1/make-it-fail`

Beispiel: Fault Injection

DEEPSEC

```
FAULT_INJECTION: forcing a failure

…[<c01672f6>] do_sync_write+0xc4/0x101

[<c0167af5>] vfs_write+0xaf/0x138

[<c0167fe2>] sys_write+0x3d/0x61

[<c0103da2>] sysenter_past_esp+0x5f/0x99

=======================

Buffer I/O error on device sdb1, logical block 35497

kernel BUG at fs/logfs/gc.c:88!

invalid opcode: 0000 [#1]

PREEMPT

Modules linked in: iptable_nat nf_nat ipt_ULOG ipt_recent nf_conntrack_ipv4 xt_state nf_conntrack
nfnetlink ipt_REJECT xt_tcpudp iptable_filter ip_tables x_tables capability usblp commoncap loop dm_mod
video thermal sbs fan container dock battery ac floppy …

CPU: 0

EIP: 0060:[<c01bd9fc>] Tainted: P VLI…
```

🎲 Fuzzing 🎰

DEEPSEC

- Fuzzing benutzt Zufallsdaten als Eingabe für Code
  - bekannt seit den 1950ern (Verwendung defekter Lochkarten)
  - Forschungsgebiet der Informatik seit 1981 🔬
- Strategien
  - komplett zufällige Daten
  - strukturbewusste zufällige Daten
  - struktur- und codebewusste zufällige Daten
- Black / White Box Modell
- Standardwerkzeug der Softwareentwicklung (eigentlich…)

Fuzzing?

Füttern des Codes mit Zufallsdaten:

```
<HTML onLoad=mk:"−2025369535" onLoad="1149452846" STYLE="ðððmk:−1187723223" STYLE="||"STYLEè"*" STYLE=left STYLE"↩
    3" onLoad=# STYLE"1563660472" onLoad=&http:^W^W^W^W^W^W^W^W^W^W^W^W^W^W; onLoad="1729332358" STYLE=®®®®®®®®®↩
     STYLE=ØØØØØØØØ onLoad=11111 onLoad<9d>−549679068 onLoad="%n%n%n%n%n%n"STYLE="774814052"><òTFOOT COLSPAN↩
    =812786791 STYLE=top onLoad="&"%n%n%n%n%n%n";" ALIGN="mk:"−402447310"" onLoad=80700691 ROWSPAN=−2005593860 ↩
    ROWSPAN="ÌÌÌÌÌÌÌ" VALIGNæfile:"−1675874920"^RBGCOLOR="about:"↩
    NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN"" STYLE="file:http:"↩
    file:ÿÿÿÿÿÿÿ"" ROWSPAN="jjjjjjj" ALIGN=^T^T^T^T^T^T^T^T^T><FRAMESET onLoad="644544716" ROWS=javascript:"↩
    javascript:#" STYLE=* onLoad=* COLS=^O^O^O^O^O^O^O ROWS="−607933135" onLoad="−1916071270" onLoad="1929093769↩
    "onLoad="682451816" COLS=ÜÜÜÜÜÜÜÜÜ onLoad="mk:ææææææææ" ROWS=éééééééé onLoad="_blank" ROWS=^N^N^N^N^N^NSTYLE↩
    =1558603635ÕROWS"−861401156" COLS=1915456235 ROWS="top" COLS=À STYLE="%n%n%n%n%n%n""about:"><MARQUEE onLoad=↩
    file:CCCCCCCCC LOOP=mk:"#""~~~~~">
...
```

Oder alternativ:

Implementationen

DEEPSEC

- Fuzzing mit reinen Daten
  - Fuzzer erzeugt Daten
  - Applikation versucht zu verarbeiten
- Fuzzing mit Hooks im Code
  - gezieltes Einschleusen von Daten
  - besseres Testen von “tiefen” Komponenten
  - Erreichen von mehr Code Pfaden

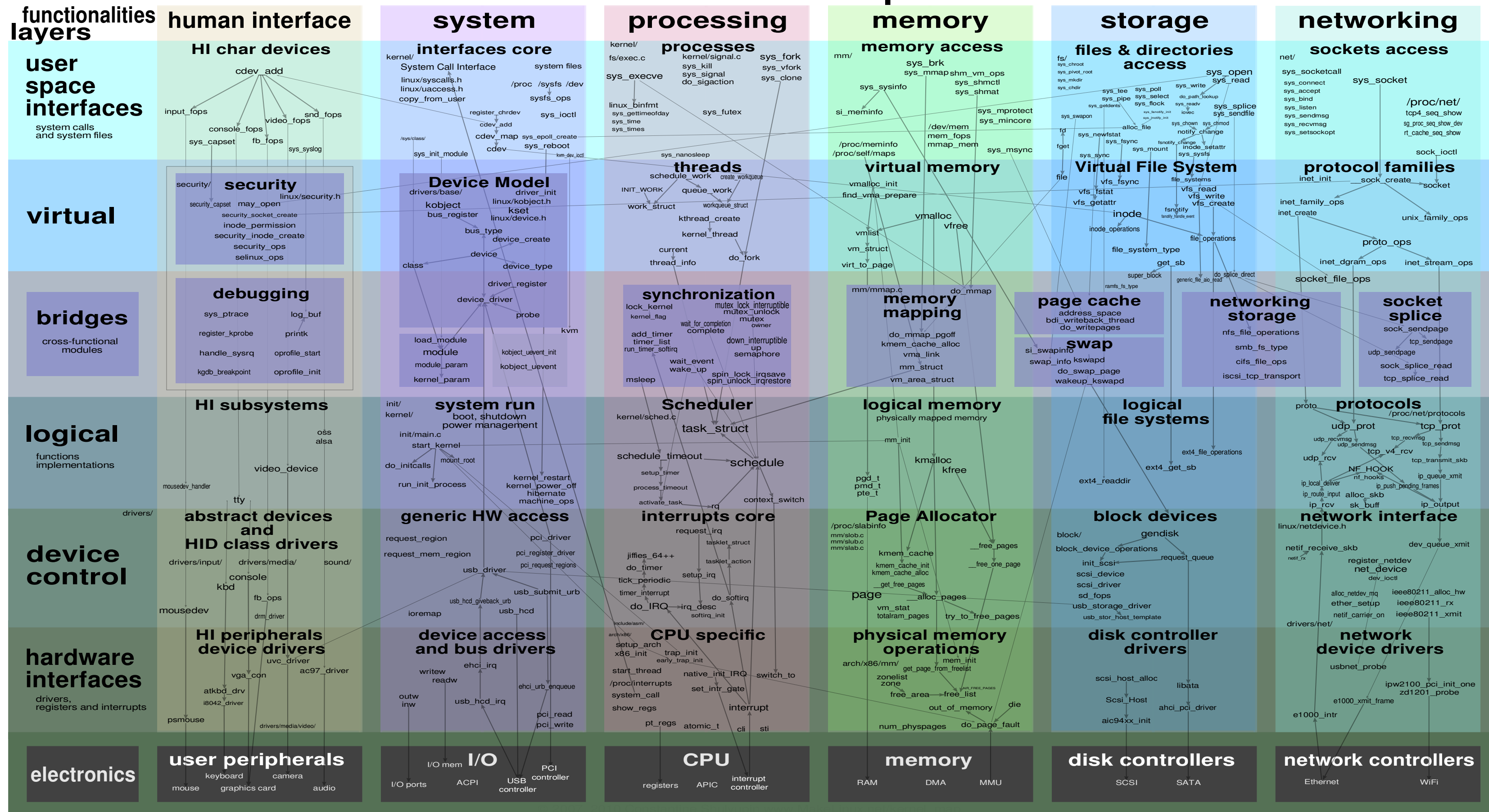
Zusammenfassung

DEEPSEC

- Secure Coding ≠ Secure Design.
- Secure Design ≠ Design.
  - Zusätzliche Anforderungen notwendig.
  - Muss mit Softwareentwicklung integriert werden.
- Manche Applikationen werden Secure Design/Coding nie verwenden.
- Lieblingshardware, -software, -bibliothek ist ~~wahrscheinlich~~ defekt.
- Softwareentwicklung muss mit Versagen geschehen. 😿 😻
- Eigene Applikation einfach mal ein Paar Monate ins Internet stellen. 😈

Fragen?

DEEPSEC



Quelle: https://commons.wikimedia.org/wiki/File:Linux_kernel_map.svg

Weiterführende Quellen

DEEPSEC

- CERT C Secure Coding Standard
- SEI CERT C and C++ Coding Standards (Übersicht)
- Secure Coding Guidelines for Java SE
- Common Weakness Enumeration (CWE™)
- .NET Secure Coding Guidelines
- OWASP Security Knowledge Framework
- Learn Rust (Dokumentation, empfohlene Lektüre)

Über den Autor

DEEPSEC

René Pfeiffer was born in the year of Atari's founding and the release of the game Pong. Since his early youth he started taking things apart to see how they work. He couldn't even pass construction sites without looking for electrical wires that might seem interesting. The interest in computing began when his grandfather bought him a 4-bit microcontroller with 256 byte RAM and a 4096 byte operating system, forcing him to learn Texas Instruments TMS 1600 assembler before any other programming language.
After finishing school he went to university in order to study physics. He then collected experiences with a C64, a C128, two Commodore Amigas, DEC's Ultrix, OpenVMS and finally GNU/Linux on a PC in 1997. He is using Linux since this day and still likes to take things apart und put them together again. Freedom of tinkering brought him close to the Free Software movement, where he puts some effort into the right to understand how things work – which he still does.

- DeepSec web site https://deepsec.net/
- DeepSec blog https://blog.deepsec.net/
- Contact information https://deepsec.net/contact.html
- We prefer end-to-end encrypted (E2EE) communication without backdoors such as:
  - Gnu Privacy Guard (GPG)
  - Signal messenger
  - Threema messenger
  - GSMK CryptoPhone