TECHNISCHE HOCHSCHULE DEGGENDORF

Bachelor – Cyber Security

Kryptologie 2

Projektdokumentation

Cryptochallenge: CurveBall (CVE-2020-0601)

Autoren
Manuel Friedl – 1236626
Christof Renner – 22301943

Betreuer
Prof. Dr. Martin Schramm

Deggendorf, July 23, 2025

Contents

1 Einleitung und Projektkontext

1.1 Motivation

Die Schwachstelle **CurveBall** (CVE-2020-0601) in der Windows-CryptoAPI ermöglicht es, X.509-Zertifikate mit manipulierten Elliptic-Curve-Parametern zu signieren, sodass betroffene Windows-Versionen die Signaturen fälschlich als gültig akzeptieren.[1] Im Modul *Kryptologie 2* fehlte bislang ein modernes Hands-On-Szenario, um diesen Fehler praktisch zu demonstrieren.

1.2 Projektziele

1. **Didaktik**: Vollständiger Angriffszyklus von Discovery bis Exploit.
2. **Sicherheit**: Deployment selbst muss trotz absichtlich verletzter Krypto sicher sein.
3. **Portabilität**: Schnelle, plattformunabhängige Nutzung via Docker/Podman.

1.3 Bedrohungsmodell

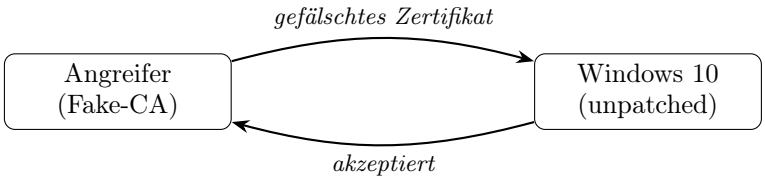

Figure 1: Simplifiziertes Threat-Model: fehlende Parameter-Validierung

2 Herangehensweise

2.1 Zielsetzung

- Demonstration des Angriffs in einer kontrollierten Umgebung.
- Vermittlung von DevSecOps-Best-Practices (Linting, CI, Scans).
- Bereitstellung als *„One-Click“*-Container, ohne lokale OpenSSL-Konfiguration.

2.2 Methodik

Wir arbeiteten in zwei Sprints à zwei Wochen. Abbildung 2 zeigt den iterativen Ablauf.

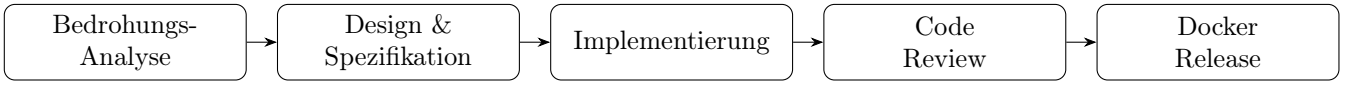

Figure 2: Iterativer Projektablauf

[1] Microsoft Security Advisory ADV200002, 14. 01. 2020

3 Arbeitsaufteilung

Die Arbeitsaufteilung erfolgte pragmatisch basierend auf den individuellen Stärken der Teammitglieder, wobei gleichzeitig Wert auf Wissenstransfer und gemeinsames Lernen gelegt wurde.

Teammitglied	**Hauptaufgaben**	**Spezifische Aufgaben**
Manuel Friedl	Kryptographie & Frontend	• Analyse der CVE-2020-0601 Schwachstelle • Entwicklung der Python-Skripte `gen_key.py` und `simulate_vuln_check.py` • Entwicklung des Web-Interface und Zertifikats-Visualizers • Erstellung verständlicher Challenge-Beschreibungen • Implementierung der Angriffslogik in Python • Frontend-Design mit HTML/CSS • Dokumentation der kryptographischen Konzepte
Christof Renner	DevOps & Infrastruktur	• Docker-Containerisierung mit Multi-Stage-Builds • CI/CD-Pipeline mit GitHub Actions • Security-Scanning und Linting-Integration • GitHub Container Registry Konfiguration • Dokumentation der technischen Details • Container-Deployment-Architektur • Implementierung der Sicherheitsmaßnahmen • Frontend-Optimierungen

Dabei wurde durch regelmäßige Abstimmungen und gemeinsame Review-Sessions sichergestellt, dass alle Komponenten nahtlos zusammenarbeiten. Die technische Dokumentation wurde gleichmäßig zwischen beiden Teammitgliedern aufgeteilt, wobei jeder etwa 50% der Dokumentationsarbeit übernahm.

4 Containerisierung

4.1 Dockerfile

Das Projekt nutzt ein *Multi-Stage-Build*-Konzept, um die Abhängigkeiten in einem schlanken Container zu installieren und gleichzeitig die Größe des finalen Images zu minimieren.
Dabei wird beim starten des Containers ein einfacher HTTP-Server gestartet, der die Challenge-Dateien bereitstellt. Die Abhängigkeiten werden in einem separaten Build-Stage installiert, um die Sicherheit und Portabilität zu erhöhen.
Die Abhängigkeiten sind in der Datei `requirements.txt` definiert, die die benötigten Python-Pakete enthält.

```
FROM python:3.12-slim AS build
RUN apt-get update && apt-get install -y --no-install-recommends \
    libssl-dev build-essential && rm -rf /var/lib/apt/lists/*
COPY requirements.txt .
RUN pip install --prefix=/install -r requirements.txt

FROM python:3.12-slim
COPY --from=build /install /usr/local
COPY curveball-ctf /app
WORKDIR /app
USER 1001:1001
ENTRYPOINT ["python", "-m", "http.server", "8080"]
```

Auflistung 1: Auszug aus dem finalen Dockerfile

4.2 Architektur

Figure 3: Container-Deployment in der Lehrumgebung

5 CI/CD-Pipeline

5.1 Docker Build-Pipeline

Die automatisierte Erstellung und Veröffentlichung der Docker-Images erfolgt über eine dedizierte GitHub Actions Workflow-Datei. Diese Pipeline stellt sicher, dass bei jeder Aktualisierung des Codes oder auf manuelle Anforderung ein neues, konsistentes Container-Image erstellt und in die Docker Hub Registry hochgeladen wird. Der Workflow ist so konfiguriert, dass er manuell über die GitHub-Oberfläche ausgelöst werden kann (`workflow_dispatch`), was besonders während der Entwicklungsphase und für kontrollierte Releases nützlich war.

```yaml
name: Build and Publish Docker image

on:
  workflow_dispatch:

jobs:
  build-and-push:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and push Docker image
        uses: docker/build-push-action@v6
        with:
          context: ./curveball-ctf/webserver
          file: ./curveball-ctf/webserver/Dockerfile
          push: true
          tags: crnnr/curveball-cve-2020-0601:latest
```

Auflistung 2: Docker Build & Publish Workflow

Die Pipeline besteht aus mehreren wichtigen Schritten:

Durch diese Automatisierung wird der Release-Prozess erheblich vereinfacht und gleichzeitig sichergestellt, dass jedes veröffentlichte Image den gleichen, reproduzierbaren Build-Prozess durchlaufen hat.

Die Verwendung von gesicherten Secrets für die Authentifizierung erhöht zusätzlich die Sicherheit der Pipeline. Die Container-Registry fungiert als zentrales Repository für die fertigen Images, was die Verteilung an Studierende erheblich vereinfacht.

Mit einem einfachen `docker pull` Befehl können Dozenten und Studenten die aktuelle Version der Challenge beziehen.

5.2 Linting-Tools

Für die Code-Qualität und Sicherheit setzen wir auf etablierte Tools:

- **pylint**: Python-Code-Analyse nach PEP8
- **hadolint**: Dockerfile-Best-Practices

Diese Tools sind in der CI-Pipeline integriert und prüfen den Code bei jedem Commit.

```yaml
jobs:
  build:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        python-version: ["3.8", "3.9", "3.10"]
    steps:
    - uses: actions/checkout@v4
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v5
      with:
        python-version: ${{ matrix.python-version }}
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install pylint
    - name: Analysing the code with pylint
      run: |
        pylint $(git ls-files '*.py')
```

Auflistung 3: pylint.yml

Damit konnte am Ende ein **pylint**-Score von **9.5/10** erreicht werden, was dafür sorgt, dass der Code auch in Zukunft gut lesbar und wartbar bleibt.

5.3 Dockerfile-Linting

Die Dockerfile-Qualität wird mit **hadolint** geprüft, um Best Practices zu gewährleisten. Das Linting erfolgt ebenfalls in der CI-Pipeline, um sicherzustellen, dass das Dockerfile den Standards entspricht.

```yaml
jobs:
  hadolint:
    name: Hadolint
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Run Hadolint
        uses: hadolint/hadolint-action@v2
        with:
          dockerfile: ./Dockerfile
```

Auflistung 4: hadolint.yml

5.4 Sicherheits-Scans

Die Sicherheitsüberprüfung erfolgt mit **Bandit** und **Snyk**. Bandit analysiert Python-Code auf Sicherheitslücken, während Snyk Container-Images auf bekannte Schwachstellen prüft. Bandit ist in der CI-Pipeline integriert und prüft den Code bei jedem Commit.

```yaml
jobs:
  bandit:
    name: Bandit Scan
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.x'
      - name: Install Bandit
        run: |
          python -m pip install --upgrade pip
          pip install bandit

      - name: Run Bandit security scan
        run: |
          # scan the repo recursively instead of using stdin
          bandit -r . --skip trojansource
```

Auflistung 5: bandit workflow file

Damit wird sichergestellt, dass der Code und die Container-Images regelmäßig auf Sicherheitslücken überprüft werden. Die Ergebnisse der Scans werden in der CI-Pipeline angezeigt. Hierbei wurden einige Schwachstellen identifiziert:

```
Code scanned:
        Total lines of code: 602
        Total lines skipped (#nosec): 0
        Total potential issues skipped due to specifically being disabled (e.g.,
             #nosec BXXX): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 6
                Medium: 1
                High: 1
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 3
                High: 5
```

Auflistung 6: bandit scan results

Diese Schwachstellen wurden im Rahmen der Entwicklung behoben, um die Sicherheit des Projekts zu gewährleisten.

6 Nicht umgesetzte VM-Erweiterung

Ursprünglich war eine vorgefertigte Windows 10-VM (Version 1909, ungepatcht) als zentraler Bestandteil des Projekts geplant, um den CurveBall-Angriff vollständig bis zum System-Rootstore zu demonstrieren und eine authentische Angriffsumgebung zu schaffen. Diese VM hätte es ermöglicht, die realen Auswirkungen eines erfolgreichen Angriffs unmittelbar zu beobachten, einschließlich der Manipulation von HTTPS-Verbindungen und Code-Signatur-Verifikationen.

6.1 Herausforderungen bei der VM-Implementierung

Die Umsetzung dieses Ansatzes scheiterte letztendlich aus mehreren gravierenden Gründen:

Licensing-Problematik Die Weitergabe eines vorinstallierten Windows-Images verstößt eindeutig gegen die Microsoft End User License Agreement (EULA). Eine rechtskonforme Lösung hätte erfordert, dass jeder Teilnehmer eine eigene Windows-Lizenz einbringt und selbst eine Installation durchführt, was den niederschwelligen Zugang zur Übung erheblich erschwert hätte.

Storage-Anforderungen Das vollständige Windows 10-Image hätte mindestens 8 GB Speicherplatz benötigt, selbst in komprimierter Form. Dies hätte den Rahmen des Git-Repositorys gesprengt und eine Nutzung von Git LFS (Large File Storage) erforderlich gemacht, was mit erheblichen Kosten für Bandbreite und Speicherplatz verbunden gewesen wäre. Zudem hätte die Verteilung an mehrere Dutzend Studierende die Netzwerkressourcen der Hochschule stark belastet.

CI/CD-Limitationen Die genutzten GitHub-Actions-Runner unterstützen keine Nested-Virtualisation, was eine automatisierte Überprüfung und Tests der VM innerhalb der CI/CD-Pipeline unmöglich machte. Dies hätte zu nicht getesteten Releases führen können, was dem DevSecOps-Ansatz des Projekts fundamental widerspricht.

Wartungsaufwand Mit jedem Windows-Update wäre eine Aktualisierung des Basis-Images nötig geworden, um die Verwundbarkeit zu erhalten. Dies hätte einen erheblichen fortlaufenden Wartungsaufwand bedeutet und die langfristige Nutzbarkeit des Lernmaterials gefährdet.

6.2 Vorteile der Container-Lösung

Die stattdessen implementierte Container-Variante bietet mit rund 280 MB eine um mehr als 96% reduzierte Größe im Vergleich zur VM-Lösung. Diese drastische Reduktion ermöglicht:

- **Schnellere Deployments**: Studierende können das Image in Sekunden statt Minuten herunterladen
- **Plattformunabhängigkeit**: Funktioniert auf Windows, macOS und Linux ohne Anpassungen
- **Geringere Systemanforderungen**: Läuft auf nahezu jedem Rechner, der Docker unterstützt
- **Einfache Integration in CI/CD**: Vollständige Testabdeckung in der Entwicklungspipeline
- **Reproduzierbare Builds**: Jeder Build erzeugt identische Umgebungen

7 Ausblick

Die entwickelte Container-Lösung bildet eine solide Grundlage für weitere Erweiterungen des Projekts. Folgende Weiterentwicklungen sind für zukünftige Versionen angedacht:

7.1 Potenzielle Erweiterungen

1. **Windows-Live-Lab über Azure Lab Services**:
   - Bereitstellung von echten, identisch konfigurierten Windows-Hosts in verschiedenen Patch-Zuständen (vor und nach CVE-2020-0601-Patch)
   - Integration einer kontrollierten Internet-Umgebung zum Testen realer TLS-Verbindungen
   - Automatisierte Provisioning-Lösung mit Infrastructure-as-Code (Terraform/ARM-Templates)
   - Zeitgesteuerte Verfügbarkeit zur Kostenoptimierung und Ressourcenschonung
   - Zentrale Überwachungsmöglichkeiten für Dozenten zur Bewertung des Lernfortschritts
2. **Automatisierte Angriffskette**:
   - Integration von Browser-Automation mit Playwright/Selenium für einen vollständigen End-to-End-Exploit
   - Demonstration der Auswirkungen auf verschiedene Browserfamilien (Chromium, Firefox, Safari)
   - Simulation eines Man-in-the-Middle-Angriffsszenarios mit TLS-Interception
   - Visualisierung der Angriffsphasen mit interaktivem Ablaufdiagramm
   - Erweiterung um zusätzliche Krypto-Angriffe (etwa ALPACA oder BEAST) für umfassendere Lernszenarien

8 Fazit

Unser Projekt demonstriert eindrucksvoll, wie sich ein kritischer kryptographischer Schwachpunkt wie CurveBall (CVE-2020-0601) in eine didaktisch wertvolle, aber dennoch sichere Lernumgebung überführen lässt. Die entwickelte Lösung schlägt erfolgreich die Brücke zwischen theoretischem Verständnis und praktischer Anwendung im Bereich der Kryptographie.

8.1 Erreichte Ziele

Die Containerisierung des Projekts ermöglicht einen niederschwelligen Zugang zur komplexen Thematik der Elliptischen-Kurven-Kryptographie und deren potentieller Schwachstellen. Dabei wurden alle initial definierten Projektziele erreicht:

- **Didaktischer Mehrwert**: Die Studierenden können den vollständigen Angriffszyklus von der theoretischen Grundlage bis zur praktischen Exploitation nachvollziehen und selbst durchführen.
- **Sicherheitskonzept**: Trotz der absichtlich integrierten kryptographischen Schwachstelle ist das Deployment selbst durch die Containerisierung und strikte Isolation inhärent sicher gestaltet.
- **Plattformunabhängigkeit**: Die Docker/Podman-basierte Lösung garantiert eine konsistente Erfahrung über verschiedene Betriebssysteme und Hardwarekonfigurationen hinweg.

8.2 Kompetenzerwerb

Neben dem technischen Verständnis für die CurveBall-Schwachstelle erwerben die Studierenden durch die Auseinandersetzung mit dem Projekt wesentliche Kompetenzen in:

- Grundlagen der Public-Key-Infrastruktur und X.509-Zertifikaten
- Praktischer Anwendung von kryptographischen Bibliotheken (OpenSSL, cryptography)
- Nutzung und Verständnis moderner DevSecOps-Workflows und CI/CD-Pipelines
- Containerisierung und Deployment von Anwendungen
- Sicherheitsrelevanten Best Practices in der Softwareentwicklung

8.3 Langfristiger Nutzen

Die entwickelte Lösung stellt nicht nur ein temporäres Lernmittel dar, sondern kann als wiederverwendbare Plattform für zukünftige kryptographische Szenarien dienen. Der modular aufgebaute Container und die umfassende Dokumentation ermöglichen eine einfache Erweiterung und Anpassung an neue Anforderungen oder andere kryptographische Schwachstellen.

Die Integration von DevSecOps-Praktiken in das Projekt selbst vermittelt den Studierenden zusätzlich einen Einblick in die heute in der Industrie etablierten Sicherheitsstandards und Workflows, was einen wertvollen Praxisbezug herstellt und die Employability der Absolventen steigert.