TECHNISCHE HOCHSCHULE DEGGENDORF

Bachelor Cyber Security

Kryptologie 2 Projektbericht

Ausarbeitung Cryptochallenge: CurveBall

Autor:

Manuel Friedl, Matrikel-Nr.: 12306626
Christof Renner, Matrikel-Nr.: 22301943

Betreuer:

Prof. Dr. Martin Schramm

Deggendorf – 28.07.2025

Contents

1 Einleitung

Musterlösungen zur CurveBall CTF Challenge
Diese Musterlösungen zeigen die schrittweise Bearbeitung aller Challenges und erklären die zugrundeliegenden kryptographischen Konzepte. Die Lösungen dienen als Referenz für Lehrende und zur Selbstkontrolle für Studierende.

Die CurveBall-Challenge ist eine interaktive Lernplattform, die Studierende in die Welt der elliptischen Kurven-Kryptographie einführt. Diese Musterlösungen zeigen die schrittweise Bearbeitung aller Challenges und erklären die zugrundeliegenden kryptographischen Konzepte.

2 Challenge 1: ECC Grundlagen

Challenge 1 - Übersicht
Diese Challenge behandelt die Grundlagen der elliptischen Kurven-Kryptographie durch praktische Punktmultiplikation. Ziel ist die Berechnung des Public Keys durch schrittweise Ermittlung von $\mathbf{P = 7 \times G}$.

2.1 Aufgabenstellung

Challenge 1 behandelt die Grundlagen der elliptischen Kurven-Kryptographie durch praktische Punktmultiplikation. Die Aufgabe besteht darin, den Public Key durch schrittweise Berechnung von $\mathbf{P = 7 \times G}$ zu ermitteln.

Gegebene Parameter

- **Elliptische Kurve:** $y^2 \equiv x^3 + 3x + 3 \pmod{97}$
- **Generator-Punkt:** $G = (11, 3)$
- **Private Key:** $d = 7$
- **Ziel:** Berechnung von $P = 7G$ (Public Key)

2.2 Mathematische Grundlagen

2.2.1 Punktaddition auf elliptischen Kurven

Grundlagen der Punktarithmetik
Für zwei Punkte $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$ auf der elliptischen Kurve $y^2 = x^3 + ax + b$ gelten verschiedene Additionsregeln je nach Punktkonstellation.

Für zwei Punkte $P_1 = (x_1, y_1)$ und $P_2 = (x_2, y_2)$ auf der elliptischen Kurve $y^2 = x^3 + ax + b$ gilt:

Fall 1: Verschiedene Punkte ($P_1 \neq P_2$)

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \pmod{p} \tag{1}$$
$$x_3 = \lambda^2 - x_1 - x_2 \pmod{p} \tag{2}$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p} \tag{3}$$

Fall 2: Punktverdopplung ($P_1 = P_2$)

$$\lambda = \frac{3x_1^2 + a}{2y_1} \pmod{p} \tag{4}$$
$$x_3 = \lambda^2 - 2x_1 \pmod{p} \tag{5}$$
$$y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p} \tag{6}$$

2.3 Schrittweise Lösung

2.3.1 Schritt 1: Berechnung von 1G

Lösung Schritt 1

$$1G = G = (11, 3) \tag{7}$$

2.3.2 Schritt 2: Berechnung von 2G = G + G (Punktverdopplung)

Wichtiger Hinweis: Bei der Punktverdopplung muss die spezielle Formel für $P_1 = P_2$ verwendet werden.

Gegeben: $G = (11, 3)$, $a = 3$, $p = 97$

$$\lambda = \frac{3 \cdot 11^2 + 3}{2 \cdot 3} \pmod{97} \tag{8}$$
$$= \frac{3 \cdot 121 + 3}{6} \pmod{97} \tag{9}$$
$$= \frac{366}{6} \pmod{97} = \frac{75}{6} \pmod{97} \tag{10}$$
$$= 75 \cdot 6^{-1} \pmod{97} \tag{11}$$

Berechnung des modularen Inversen von 12 modulo 97:

$$6^{-1} \equiv 81 \pmod{97} \quad (\text{da } 6 \cdot 81 = 486 \equiv 1 \pmod{97}) \tag{12}$$

$$\lambda = 75 \cdot 81 \pmod{97} = 6075 \pmod{97} = 86 \tag{13}$$
$$x_3 = 86^2 - 2 \cdot 11 \pmod{97} = 7396 - 22 \pmod{97} = 13 \tag{14}$$
$$y_3 = 86 \cdot (11 - 13) - 3 \pmod{97} = 86 \cdot (-2) - 3 \pmod{97} \tag{15}$$
$$= -175 \pmod{97} = 69 \tag{16}$$

Ergebnis Schritt 2: $2G = (13, 69)$

2.3.3 Schritt 3: Berechnung von 4G = 2G + 2G

Gegeben: $2G = (13, 69)$

$$\lambda = \frac{3 \cdot 13^2 + 3}{2 \cdot 89} \pmod{97} \tag{17}$$
$$= \frac{3 \cdot 169 + 3}{138} \pmod{97} \tag{18}$$
$$= \frac{510}{138} \pmod{97} \tag{19}$$
$$= \frac{25}{41} \pmod{97} \quad \text{(nach Reduktion mod 97)} \tag{20}$$
$$= 25 \cdot 41^{-1} \pmod{97} \tag{21}$$

Modularer Inverser von 77 modulo 97: $41^{-1} \equiv 19 \pmod{97}$

$$\lambda = 25 \cdot 19 \pmod{97} = 475 \pmod{97} = 90 \tag{22}$$
$$x_3 = 90^2 - 2 \cdot 13 \pmod{97} = 8100 - 26 \pmod{97} = 8074 \pmod{97} = 39 \tag{23}$$
$$y_3 = 90 \cdot (13 - 39) - 69 \pmod{97} = 90 \cdot (-26) - 69 \pmod{97} \tag{24}$$
$$= -2340 - 69 \pmod{97} = -2409 \pmod{97} = 50 \tag{25}$$

Ergebnis Schritt 3: $4G = (39, 50)$

2.3.4 Schritt 4: Berechnung von 6G = 4G + 2G

Gegeben: $4G = (39, 50)$, $2G = (13, 69)$

$$\lambda = \frac{50 - 69}{39 - 13} \pmod{97} = \frac{-19}{26} \pmod{97} \tag{26}$$
$$= (-19) \cdot 26^{-1} \pmod{97} = 78 \cdot 56 \pmod{97} \tag{27}$$
$$= 4368 \pmod{97} = 3 \tag{28}$$

$$x_3 = 3^2 - 13 - 39 \pmod{97} = 9 - 52 \pmod{97} = -27 \pmod{97} = 54 \tag{29}$$
$$y_3 = 3 \cdot (13 - 54) - 69 \pmod{97} = 3 \cdot (-41) - 69 \pmod{97} \tag{30}$$
$$= -123 - 69 \pmod{97} = -192 \pmod{97} = 2 \tag{31}$$

2.3.5 Schritt 5: Berechnung von 7G = 6G + 1G

Gegeben: $6G = (54, 2)$, $1G = (11, 3)$

$$\lambda = \frac{3 - 2}{11 - 54} \pmod{97} = \frac{1}{-43} \pmod{97} = \frac{4}{54} \pmod{97} \tag{32}$$
$$= 1 \cdot 54^{-1} \pmod{97} = 1 \cdot 9 \pmod{97} = 9 \tag{33}$$

$$x_3 = 9^2 - 54 - 11 \pmod{97} = 3721 - 65 \pmod{97} = 3656 \pmod{97} = 16 \tag{34}$$
$$y_3 = 9 \cdot (54 - 16) - 2 \pmod{97} = 9 \cdot 38 - 2 \pmod{97} \tag{35}$$
$$= 342 - 2 \pmod{97} = 340 \pmod{97} = 49 \tag{36}$$

Finales Ergebnis: $7G = (16, 49)$

2.4 Verifikation

Verifikationsprozess
Der berechnete Public Key muss die Kurvengleichung erfüllen, um die Korrektheit der Berechnung zu bestätigen.

Der berechnete Public Key kann durch Einsetzen in die Kurvengleichung verifiziert werden:

$$y^2 \stackrel{?}{=} x^3 + 3x + 3 \pmod{97} \tag{37}$$
$$49^2 \stackrel{?}{=} 16^3 + 3 \cdot 16 + 3 \pmod{97} \tag{38}$$
$$2401 \stackrel{?}{=} 4096 + 48 + 3 \pmod{97} \tag{39}$$
$$73 \stackrel{?}{=} 73 \pmod{97} \quad \checkmark \tag{40}$$

2.5 Kryptographische Bedeutung

Kryptographische Erkenntnisse

Diese Challenge demonstriert:

- **Skalarmultiplikation:** Die Grundoperation für Public-Key-Generierung
- **Punktarithmetik:** Mathematische Operationen auf elliptischen Kurven
- **Modulare Arithmetik:** Alle Berechnungen erfolgen in einem endlichen Körper
- **ECDLP (Elliptic Curve Discrete Logarithm Problem):** Die Umkehrung (Finden von d bei gegebenem P und G) ist rechnerisch schwer

3 Challenge 2: Zertifikatsanalyse

Challenge 2 - Übersicht
Diese Challenge fokussiert auf die praktische Analyse von X.509-Zertifikaten mit OpenSSL. Ziel ist die Untersuchung eines verdächtigen Zertifikats und das Auffinden versteckter Informationen.

3.1 Aufgabenstellung

Challenge 2 fokussiert auf die praktische Analyse von X.509-Zertifikaten mit OpenSSL. Die Aufgabe besteht darin, ein verdächtiges Zertifikat zu untersuchen und versteckte Informationen zu finden.

Ziele der Challenge

- Praktische Anwendung von OpenSSL zur Zertifikatsanalyse
- Verstehen der X.509-Zertifikatstruktur
- Identifizierung versteckter Informationen in Zertifikatsfeldern
- Vorbereitung auf die Curveball-Schwachstelle (CVE-2020-0601)

3.2 Theoretischer Hintergrund

3.2.1 X.509-Zertifikatstruktur

X.509-Standard
X.509 ist der internationale Standard für Public-Key-Zertifikate und definiert das Format für Zertifikate in Public-Key-Infrastrukturen (PKI).

Ein X.509-Zertifikat besteht aus folgenden Hauptkomponenten:

- **Version:** X.509-Version (meist v3)
- **Serial Number:** Eindeutige Seriennummer
- **Signature Algorithm:** Verwendeter Signaturalgorithmus
- **Issuer:** Ausstellende Zertifizierungsstelle (CA)
- **Validity:** Gültigkeitszeitraum (Not Before/Not After)
- **Subject:** Zertifikatinhaber-Informationen
- **Public Key Info:** Öffentlicher Schlüssel und Algorithmus
- **Extensions:** Zusätzliche Felder (nur in v3)
- **Signature:** Digitale Signatur der CA

3.2.2 Relevante X.509v3 Extensions

Wichtige Extensions

- **Subject Alternative Name (SAN):** Alternative Identifikatoren
- **Key Usage:** Erlaubte Schlüsselverwendungen
- **Extended Key Usage:** Spezifische Anwendungszwecke
- **Basic Constraints:** CA-Eigenschaften
- **Certificate Policies:** Richtlinien-Informationen
- **Authority Information Access:** CA-Zugriffsinformationen

3.3 Praktische Lösung

3.3.1 Schritt 1: Download und erste Inspektion

Vorbereitung: Stellen Sie sicher, dass OpenSSL auf Ihrem System installiert ist und die Challenge-Umgebung läuft.

Das bereitgestellte Zertifikat `mystery_cert.pem` wird heruntergeladen und zunächst grundlegend analysiert:

```
openssl x509 -in mystery_cert.pem -text -noout
```

Auflistung 1: Grundlegende Zertifikatsinformationen

3.3.2 Schritt 2: Strukturierte Analyse

Analysestrategie
Eine systematische Herangehensweise ist entscheidend für die erfolgreiche Zertifikatsanalyse. Beginnen Sie mit den Grundinformationen und arbeiten Sie sich zu den Extensions vor.

Issuer und Subject Information:

```
# Zertifikatinhaber anzeigen
openssl x509 -in mystery_cert.pem -subject -noout

# Ausstellende CA anzeigen
openssl x509 -in mystery_cert.pem -issuer -noout
```

Erwartete Ausgabe:

```
subject=CN = Suspicious Certificate, O = Evil Corp,
C = XX, emailAddress = admin@evil-corp.example
issuer=CN = Curveball Demo CA, O = THD Cryptography Lab,
C = DE
```

Auflistung 2: Zertifikatsinformationen

3.3.3 Schritt 3: Extension-Analyse

Der kritische Schritt liegt in der Untersuchung der X.509v3-Extensions:

```
# Alle Extensions anzeigen
openssl x509 -in mystery_cert.pem -text -noout |
grep -A 20 "X509v3 extensions"

# Spezifische Suche nach versteckten Informationen
openssl x509 -in mystery_cert.pem -text -noout |
grep -i -A 5 -B 5 "flag"
```

3.3.4 Schritt 4: Detaillierte Feldanalyse

Subject Alternative Names:

```
openssl x509 -in mystery_cert.pem -text -noout |
grep -A 5 "Subject Alternative Name"
```

Certificate Policies:

```
openssl x509 -in mystery_cert.pem -text -noout |
grep -A 10 "Certificate Policies"
```

Authority Information Access:

```
openssl x509 -in mystery_cert.pem -text -noout |
grep -A 5 "Authority Information Access"
```

3.4 Lösungsweg und versteckte Information

3.4.1 Flag-Location

Versteckte Flags
Die versteckten Flags befinden sich typischerweise in den X.509v3-Extensions, speziell in Feldern, die normalerweise für legitime Zwecke verwendet werden.

Die versteckte Flag befindet sich typischerweise in einem der folgenden Bereiche:

1. **Subject Alternative Name Extension:**

```
    X509v3 Subject Alternative Name:
        DNS:evil-corp.example,
        DNS:FLAG{hidden_in_certificate_extensions}.curveball.local
```

Auflistung 3: SAN Extension mit Flag

2. **Certificate Policies:**

```
    X509v3 Certificate Policies:
        Policy: 1.2.3.4.5.FLAG{x509_extensions_reveal_secrets}
```

Auflistung 4: Certificate Policies mit Flag

3. **CRL Distribution Points:**

```
    X509v3 CRL Distribution Points:
        Full Name:
          URI:http://crl.example.com/FLAG{certificate_analysis_complete}.crl
```

Auflistung 5: CRL Distribution Points mit Flag

3.4.2 Automatisierte Suche

Ein effizienterer Ansatz zur Flag-Findung:

```
# Suche nach Flag-Pattern
openssl x509 -in mystery_cert.pem -text -noout |
grep -oP 'FLAG\{[^}]*\}'
```

Auflistung 6: Automatisierte Flag-Suche

3.5 Kryptographische Relevanz für Curveball

3.5.1 Verbindung zu CVE-2020-0601

Diese Challenge bereitet auf die Curveball-Schwachstelle vor, indem sie zeigt:

- **Zertifikatsstruktur:** Verständnis für X.509-Aufbau ist essentiell
- **Extension-Parsing:** Fehlerhafte Validierung von Extensions war Teil der Schwachstelle
- **Parameter-Überprüfung:** Windows validierte ECC-Parameter in Zertifikaten unzureichend
- **Trust-Chain:** Manipulation der Vertrauenskette durch gefälschte Zertifikate

3.5.2 Praktische Sicherheitsimplikationen

- **Certificate Pinning:** Wichtigkeit der Zertifikatsfixierung
- **Validierung:** Notwendigkeit umfassender Zertifikatsprüfung
- **Monitoring:** Überwachung verdächtiger Zertifikate
- **Tool-Kompetenz:** OpenSSL als universelles Analysetool

3.6 Weiterführende Analysetechniken

3.6.1 Erweiterte OpenSSL-Kommandos

```
# ASN.1-Struktur anzeigen
openssl asn1parse -i -in mystery_cert.pem

# Modulus und Exponent des public keys
openssl x509 -in mystery_cert.pem -noout -modulus

# Fingerprint-Berechnung
openssl x509 -in mystery_cert.pem -noout -fingerprint -sha256

# PEM zu DER
openssl x509 -in mystery_cert.pem -outform DER -out mystery_cert.der
```

Auflistung 7: Erweiterte Zertifikatsanalyse

3.6.2 Hexdump-Analyse

```
# Struktur untersuchen
xxd mystery_cert.der | grep -i flag

# String-Extraktion
strings mystery_cert.pem | grep -i flag
```

3.7 Lernziele erreicht

Nach Abschluss dieser Challenge verstehen Studierende:

- **X.509-Standard:** Aufbau und Struktur von Zertifikaten
- **OpenSSL-Toolkit:** Praktische Anwendung für Zertifikatsanalyse
- **Security Research:** Systematische Suche nach versteckten Informationen
- **PKI-Grundlagen:** Public Key Infrastructure Konzepte
- **Curveball-Vorbereitung:** Basis für Verständnis der CVE-2020-0601

4 Challenge 3: CurveBall Exploit

Challenge 3 - Übersicht
Diese Challenge behandelt die Simulation der kritischen Sicherheitslücke CVE-2020-0601, bekannt als "Curveball". Ziel ist die Erstellung eines manipulierten ECC-Zertifikats, das von einem verwundbaren Validierungsskript als vertrauenswürdig erkannt wird.

4.1 Aufgabenstellung

Challenge 3 behandelt die Simulation der kritischen Sicherheitslücke CVE-2020-0601, bekannt als "Curveball". Diese Schwachstelle in Windows CryptoAPI ermöglichte es Angreifern, gefälschte ECC-Zertifikate zu erstellen, die als vertrauenswürdig erkannt wurden.

Ziel

Erstellen Sie ein manipuliertes ECC-Zertifikat, das vom bereitgestellten Python-Validierungsskript als gültig erkannt wird.

4.2 Theoretischer Hintergrund

4.2.1 CVE-2020-0601 - Die Curveball-Schwachstelle

Kritische Schwachstelle
Die Curveball-Schwachstelle war eine der kritischsten Sicherheitslücken in der Windows-Geschichte und betraf die grundlegenden kryptographischen Validierungsmechanismen.

Die Curveball-Schwachstelle betraf Windows CryptoAPI und ermöglichte eine fundamentale Kompromittierung der ECC-Zertifikatsvalidierung:

- **Kernproblem:** Windows validierte ECC-Parameter (insbesondere den Generator-Punkt) nicht korrekt
- **Auswirkung:** Angreifer konnten beliebige Generator-Punkte verwenden
- **Resultat:** Gefälschte Zertifikate wurden als von vertrauenswürdigen CAs stammend akzeptiert

4.2.2 Mathematische Grundlage des Exploits

Normale ECC-Signaturvalidierung:

$$e \cdot G = r \cdot G + s \cdot Q \tag{41}$$

Dabei ist:

- G = Standard-Generator-Punkt der Kurve
- Q = Öffentlicher Schlüssel
- e = Hash der Nachricht
- (r, s) = Signatur

Curveball-Exploit:

$$e \cdot G' = r \cdot G' + s \cdot Q' \tag{42}$$

Mit manipuliertem Generator G' kann der Angreifer:

- Beliebige "gültige" Signaturen erstellen
- Den entsprechenden privaten Schlüssel kontrollieren
- Zertifikate fälschen, die scheinbar von vertrauenswürdigen CAs stammen

4.3 Praktische Durchführung

4.3.1 Schritt 1: Validierungsskript herunterladen

Das bereitgestellte Python-Skript `verification.py` simuliert die verwundbare Windows CryptoAPI:

```
# Download von der Challenge-Website
wget http://localhost:8443/static/scripts/verification.py

# Analyse-Modus executen
python verification.py analyze
```

Auflistung 8: Skript-Download

4.3.2 Schritt 2: Schwachstelle analysieren

Das Validierungsskript zeigt die kritischen Unterschiede:

```
Normal ECC Certificate Validation:
1. [YES] Verify certificate chain
2. [YES] Check certificate validity period
3. [YES] Validate certificate signature
4. [YES] Verify ECC parameters match standards
5. [YES] Ensure generator point is correct

CVE-2020-0601 Vulnerable Validation:
1. [YES] Verify certificate chain
2. [YES] Check certificate validity period
3. [YES] Validate certificate signature
4. [NO] SKIP ECC parameter validation!
5. [NO] SKIP generator point verification!
```

Auflistung 9: Vulnerable vs. Secure Validation

4.3.3 Schritt 3: Manipuliertes Zertifikat erstellen

Methode 1: OpenSSL mit manipulierten Parametern

```
# Private Key mit NIST P-256 erstellen
openssl ecparam -name prime256v1 -genkey -noout -out rogue_key.pem

# Zertifikat mit "evil" Subject erstellen
openssl req -new -x509 -key rogue_key.pem -out exploit_cert.pem \
  -days 365 -subj "/CN=evil.example.com/O=Evil Corp"
```

Auflistung 10: Rogue Key Generation

Methode 2: Python-basierte Manipulation

```python
from cryptography import x509
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
import datetime

# ECC-key generieren
private_key = ec.generate_private_key(ec.SECP256R1())

# Subject und Issuer mit "evil" Markern
subject = issuer = x509.Name([
    x509.NameAttribute(x509.NameOID.COMMON_NAME, u"evil.example.com"),
    x509.NameAttribute(x509.NameOID.ORGANIZATION_NAME, u"Evil Corp"),
])

# Zertifikat mit Exploit-Markern erstellen
cert = x509.CertificateBuilder().subject_name(
    subject
).issuer_name(
    issuer
).public_key(
    private_key.public_key()
).serial_number(
    0xdeadbeef  # Exploit-Marker
).not_valid_before(
    datetime.datetime.utcnow()
).not_valid_after(
    datetime.datetime.utcnow() + datetime.timedelta(days=365)
).sign(private_key, hashes.SHA256())

# Zertifikat speichern
with open("exploit_cert.pem", "wb") as f:
    f.write(cert.public_bytes(serialization.Encoding.PEM))
```

Auflistung 11: Zertifikat mit Exploit-Markern

4.3.4 Schritt 4: Zertifikat validieren

```
# Manipuliertes Zertifikat testen
python verification.py validate exploit_cert.pem
```

Auflistung 12: Exploit-Validierung

Erwartete Ausgabe bei erfolgreichem Exploit:

```
[TARGET] EXPLOIT MARKERS DETECTED:
   [LIGHTNING] Found: evil
   [LIGHTNING] Found: deadbeef

[LAB] SIMULATED WINDOWS CRYPTOAPI VALIDATION:
   [YES] Certificate format valid
   [YES] Signature verification (simulated)
   [WARNING] ECC parameter validation SKIPPED (vulnerable!)
   [WARNING] Generator point validation SKIPPED (vulnerable!)

[PARTY] EXPLOIT SUCCESSFUL!
   The manipulated certificate passed validation!

[FLAG] FLAG CAPTURED:
   FLAG{curveball_exploit_generator_manipulation_success}
```

Auflistung 13: Erfolgreiche Validation

4.4 Erweiterte Exploit-Techniken

4.4.1 Generator-Punkt Manipulation

Für eine detailliertere Simulation kann der Generator-Punkt direkt manipuliert werden:

```
# Standard NIST P-256 Generator
standard_gx = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
standard_gy = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5

# Manipulierter Generator (mit erkennbarem Marker)
rogue_gx = (standard_gx & 0xffffffff00000000) | 0xdeadbeef
rogue_gy = (standard_gy & 0xffffffff00000000) | 0xcafebabe

print(f"Standard Generator X: {hex(standard_gx)}")
print(f"Rogue Generator X:    {hex(rogue_gx)}")
print(f"Standard Generator Y: {hex(standard_gy)}")
print(f"Rogue Generator Y:    {hex(rogue_gy)}")
```

Auflistung 14: Generator-Punkt Manipulation

4.4.2 ASN.1-Manipulation

Für erweiterte Angriffe kann die ASN.1-Struktur direkt manipuliert werden:

```
import asn1crypto.x509
import asn1crypto.algos

# Manipulierte ECC-Parameter definieren
ecc_params = {
    'named_curve': 'prime256v1',  # Malicious: Standard-Kurve angeben
    # Manipulierte Parameter werden verwendet
}

# In realer Implementierung werden hier die ASN.1-Strukturen
# direkt manipuliert, um rogue Generator-Punkte einzubetten
```

Auflistung 15: ASN.1 ECC Parameter Manipulation

4.5 Lösung und Flag

Bei erfolgreicher Durchführung des Exploits wird die Flag ausgegeben:

`FLAG{curveball_exploit_generator_manipulation_success}`

5 Challenge 4: Kurvenparameter & Signaturvalidierung

Challenge 4 - Übersicht
Challenge 4 bildet den mathematischen Höhepunkt der CurveBall-Challenge-Serie und fokussiert auf das Herzstück der CVE-2020-0601-Schwachstelle: die Manipulation von ECC-Kurvenparametern zur Umgehung der Signaturvalidierung.

5.1 Aufgabenstellung

Challenge 4 bildet den mathematischen Höhepunkt der CurveBall-Challenge-Serie und fokussiert auf das Herzstück der CVE-2020-0601-Schwachstelle: die Manipulation von ECC-Kurvenparametern zur Umgehung der Signaturvalidierung.

Ziel

Manipulieren Sie die Kurvenparameter so, dass eine ursprünglich ungültige Signatur plötzlich als gültig erkannt wird.

5.2 Theoretischer Hintergrund

5.2.1 Mathematische Grundlage der ECC-Signaturvalidierung

ECDSA-Signaturvalidierung erfolgt in folgenden Schritten:

1. **Hash berechnen:** $e = \text{Hash}(\text{Nachricht})$
2. **Inverse berechnen:** $w = s^{-1} \pmod{n}$
3. **Skalare berechnen:**

$$u_1 = e \cdot w \pmod{n} \tag{43}$$
$$u_2 = r \cdot w \pmod{n} \tag{44}$$

4. **Punkt berechnen:** $(x_1, y_1) = u_1 \cdot G + u_2 \cdot Q$
5. **Validierung:** Signatur gültig wenn $r \equiv x_1 \pmod{n}$

Dabei sind:

- G = Generator-Punkt der elliptischen Kurve
- Q = Öffentlicher Schlüssel
- n = Ordnung der elliptischen Kurve
- (r, s) = Signatur-Komponenten
- e = Hash der Nachricht

5.2.2 Der Curveball-Exploit

Durch Manipulation der Kurvenparameter kann ein Angreifer die Validierung kontrollieren:

$$(x_1, y_1) = u_1 \cdot G' + u_2 \cdot Q' \tag{45}$$

Mit manipulierten Parametern G', n', a', b' kann der Angreifer:

- Den Generator-Punkt G' so wählen, dass gewünschte x_1-Koordinaten erreicht werden
- Die Kurvenparameter a', b' anpassen, damit der Punkt auf der Kurve liegt
- Die Ordnung n' manipulieren, um kleinere Suchräume zu schaffen

5.3 Praktische Durchführung

5.3.1 Schritt 2: Signatur-Testdaten analysieren

Die Datei `signature_data.json` enthält:

```
# JSON-Datei untersuchen
cat signature_data.json | python -m json.tool

# Wichtige Felder extrahieren
python -c "
import json
with open('signature_data.json') as f:
    data = json.load(f)
    sig = data['signature_data']['original_signature']
    print(f'Signatur r: {sig[\"r\"]}')
    print(f'Signatur s: {sig[\"s\"]}')
"
```

Auflistung 16: Analyse der Testdaten

5.3.2 Schritt 3: Normale Validierung testen

```
# Normale Validierung mit Standard-Parametern
python signature_validator.py
```

Auflistung 17: Baseline-Validierung

Erwartete Ausgabe:

```
Normal ECC Certificate Validation:
1. [YES] Hash der Nachricht: 0xabc123def...
2. [YES] Signatur-Parameter gueltig
3. [YES] Inverse berechnung erfolgreich
4. [NO] Signatur ungueltig: 0x789abc != 0x1a2b3c...
```

Auflistung 18: Normale Validierung

5.3.3 Schritt 4: Parameter-Manipulation

Methode 1: Generator-Punkt Manipulation

```python
import json

# Standard NIST P-256 Parameter laden
with open('signature_data.json') as f:
    data = json.load(f)

original_params = data['signature_data']['original_curve_params']

# Manipulierte Parameter erstellen
manipulated_params = original_params.copy()

# Generator-Punkt manipulieren
manipulated_params['generator_x'] = "0
    x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef"
manipulated_params['generator_y'] = "0
    xfedcba0987654321fedcba0987654321fedcba0987654321fedcba0987654321"

print("Manipulierte Parameter:")
for key, value in manipulated_params.items():
    print(f"{key}: {value}")
```

Auflistung 19: Generator-Manipulation

Methode 2: Kurvenparameter-Manipulation

```python
# Kurvenparameter 'a' manipulieren
manipulated_params['a'] = "0
    x0000000000000000000000000000000000000000000000000000000000000003"

# Schwaechere Kurve erzeugen
manipulated_params['b'] = "0
    x0000000000000000000000000000000000000000000000000000000000000007"

# Validation mit manipulierten Parametern
from signature_validator import ECCSignatureValidator

validator = ECCSignatureValidator(manipulated_params)
```

Auflistung 20: Kurvenkoeffizienten ändern

Methode 3: Ordnungs-Manipulation

```python
# Manipulierte Ordnung der Kurve
manipulated_params['order'] = "0
    x100000000000000000000000014def9dea2f79cd65812631a5cf5d3ed"

# Dies ermoeglicht kleinere Suchraeume fuer Angriffe
print(f"Original order:   {original_params['order']}")
print(f"Manipulated order: {manipulated_params['order']}")
```

Auflistung 21: Gruppenordnung manipulieren

5.3.4 Schritt 5: Exploit-Validierung

```python
from signature_validator import ECCSignatureValidator

# Lade Signatur-Daten
with open('signature_data.json') as f:
    data = json.load(f)

sig_data = data['signature_data']
message = sig_data['message']
```

```
signature_r = int(sig_data['original_signature']['r'], 16)
signature_s = int(sig_data['original_signature']['s'], 16)

# Simuliere Public Key (vereinfacht)
original_params = sig_data['original_curve_params']
public_key_x = int(original_params['generator_x'], 16)
public_key_y = int(original_params['generator_y'], 16)

# Test mit manipulierten Parametern
for example in sig_data['manipulated_examples']:
    print(f"Testing: {example['method']}")

    # Manipulierte Parameter erstellen
    manipulated_params = original_params.copy()

    if 'new_generator_x' in example:
        manipulated_params['generator_x'] = example['new_generator_x']
        manipulated_params['generator_y'] = example['new_generator_y']
    if 'new_a' in example:
        manipulated_params['a'] = example['new_a']
    if 'new_order' in example:
        manipulated_params['order'] = example['new_order']

    # Validierung mit manipulierten Parametern
    validator = ECCSignatureValidator(manipulated_params)
    result = validator.validate_signature(
        message, signature_r, signature_s, public_key_x, public_key_y
    )

    print(f"Result: {result['valid']}")
    print(f"Reason: {result['reason']}")
    print()
```

Auflistung 22: Vollständiger Exploit

5.4 Erwartete Ausgabe und Flag

Erfolgreicher Exploit
Bei erfolgreicher Durchführung des Exploits wird eine spezielle Flag ausgegeben, die den erfolgreichen Angriff bestätigt.

Bei erfolgreichem Exploit gibt das Validierungsskript die Flag aus:

```
[EXPLOIT] Parameter-Manipulation erfolgreich!
[TARGET] Generator-Punkt Manipulation durchgefuehrt
[SUCCESS] Urspruenglich ungueltige Signatur wird als gueltig erkannt
[WARNING] CVE-2020-0601 Schwachstelle ausgenutzt

[FLAG] FLAG CAPTURED:
FLAG{curve_parameter_manipulation_signature_bypass}
```

Auflistung 23: Erfolgreicher Exploit

5.5 Erweiterte Exploit-Techniken

5.5.1 Mathematische Präzisions-Manipulation

```python
def calculate_exploit_parameters(target_r, target_s, message_hash):
    """
    Berechnet manipulierte Parameter fuer gewuenschte Signatur-Validierung
    """
    # Ziel: Finde G' so dass u1*G' + u2*Q = (target_r, y) fuer beliebiges y

    # s_inv berechnen
    s_inv = pow(target_s, -1, ORDER)

    # u1, u2 berechnen
    u1 = (message_hash * s_inv) % ORDER
    u2 = (target_r * s_inv) % ORDER

    # Neuen Generator berechnen der gewuenschtes Ergebnis liefert
    # Dies ist eine vereinfachte Darstellung der komplexen Mathematik

    manipulated_gx = (target_r - u2 * PUBLIC_KEY_X) * pow(u1, -1, ORDER) % ORDER
    manipulated_gy = calculate_curve_point_y(manipulated_gx)

    return manipulated_gx, manipulated_gy

def calculate_curve_point_y(x, a=CURVE_A, b=CURVE_B, p=CURVE_P):
    """Berechnet y-Koordinate fuer gegebenen x-Wert auf elliptischer Kurve"""
    y_squared = (pow(x, 3, p) + a * x + b) % p
    y = pow(y_squared, (p + 1) // 4, p)  # Vereinfacht fuer p = 3 (mod 4)
    return y
```

Auflistung 24: Praezise Parameter-Berechnung

5.5.2 ASN.1-Struktur-Manipulation

```python
from cryptography import x509
from cryptography.hazmat.primitives import serialization

def create_manipulated_certificate(original_cert_path, manipulated_params):
    """
    Erstellt Zertifikat mit manipulierten ECC-Parametern
    """
    # Originales Zertifikat laden
    with open(original_cert_path, 'rb') as f:
        original_cert = x509.load_pem_x509_certificate(f.read())

    # ASN.1-Struktur manipulieren (vereinfacht)
    # In realer Implementierung wuerde hier die komplette
    # ASN.1-Struktur der ECC-Parameter ueberschrieben

    print("Manipulierte ECC-Parameter in Zertifikat:")
    print(f"Generator X: {manipulated_params['generator_x']}")
    print(f"Generator Y: {manipulated_params['generator_y']}")
    print(f"Curve A: {manipulated_params['a']}")
    print(f"Order: {manipulated_params['order']}")

    return "manipulated_certificate.pem"
```

Auflistung 25: Zertifikat-Parameter überschreiben