TECHNISCHE HOCHSCHULE DEGGENDORF

Bachelor Cyber Security

Kryptologie 2 Projektbericht

Ausarbeitung Cryptochallenge: CurveBall

Autor:

Manuel Friedl, Matrikel-Nr.: 1236626
Christof Renner, Matrikel-Nr.: 22301943

Betreuer:

Prof. Dr. Martin Schramm

Deggendorf – July 23, 2025

Contents

1 Einleitung

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

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: $G = (3, 6)$
- Private Key: $d = 7$
- Ziel: Berechnung von $P = 7G$ (Public Key)

2.2 Mathematische Grundlagen

2.2.1 Punktaddition auf elliptischen Kurven

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

$$1G = G = (3, 6) \tag{7}$$

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

Gegeben: $G = (3, 6)$, $a = 3$, $p = 97$

$$\lambda = \frac{3 \cdot 3^2 + 3}{2 \cdot 6} \pmod{97} \tag{8}$$
$$= \frac{3 \cdot 9 + 3}{12} \pmod{97} \tag{9}$$
$$= \frac{30}{12} \pmod{97} \tag{10}$$
$$= 30 \cdot 12^{-1} \pmod{97} \tag{11}$$

Berechnung des modularen Inversen von 12 modulo 97:

$$12^{-1} \equiv 89 \pmod{97} \quad (\text{da } 12 \cdot 89 = 1068 \equiv 1 \pmod{97}) \tag{12}$$

$$\lambda = 30 \cdot 89 \pmod{97} = 2670 \pmod{97} = 49 \tag{13}$$
$$x_3 = 49^2 - 2 \cdot 3 \pmod{97} = 2401 - 6 \pmod{97} = 80 \tag{14}$$
$$y_3 = 49 \cdot (3 - 80) - 6 \pmod{97} = 49 \cdot (-77) - 6 \pmod{97} \tag{15}$$
$$= -3773 - 6 \pmod{97} = -3779 \pmod{97} = 87 \tag{16}$$

Ergebnis: $2G = (80, 87)$

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

Gegeben: $2G = (80, 87)$

$$\lambda = \frac{3 \cdot 80^2 + 3}{2 \cdot 87} \pmod{97} \tag{17}$$
$$= \frac{3 \cdot 6400 + 3}{174} \pmod{97} \tag{18}$$
$$= \frac{19203}{174} \pmod{97} \tag{19}$$
$$= \frac{92}{77} \pmod{97} \quad (\text{nach Reduktion mod } 97) \tag{20}$$
$$= 92 \cdot 77^{-1} \pmod{97} \tag{21}$$

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

$$\lambda = 92 \cdot 19 \pmod{97} = 1748 \pmod{97} = 6 \tag{22}$$
$$x_3 = 6^2 - 2 \cdot 80 \pmod{97} = 36 - 160 \pmod{97} = -124 \pmod{97} = 3 \tag{23}$$
$$y_3 = 6 \cdot (80 - 3) - 87 \pmod{97} = 6 \cdot 77 - 87 \pmod{97} \tag{24}$$
$$= 462 - 87 \pmod{97} = 375 \pmod{97} = 91 \tag{25}$$

Ergebnis: $4G = (3, 91)$

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

Gegeben: $4G = (3, 91)$, $2G = (80, 87)$

$$\lambda = \frac{87-91}{80-3} \pmod{97} = \frac{-4}{77} \pmod{97} \tag{26}$$
$$= (-4) \cdot 77^{-1} \pmod{97} = (-4) \cdot 19 \pmod{97} \tag{27}$$
$$= -76 \pmod{97} = 21 \tag{28}$$

$$x_3 = 21^2 - 3 - 80 \pmod{97} = 441 - 83 \pmod{97} = 358 \pmod{97} = 67 \tag{29}$$
$$y_3 = 21 \cdot (3 - 67) - 91 \pmod{97} = 21 \cdot (-64) - 91 \pmod{97} \tag{30}$$
$$= -1344 - 91 \pmod{97} = -1435 \pmod{97} = 10 \tag{31}$$

Korrektur der Berechnung: Bei genauerer Überprüfung ergibt sich: $6G = (80, 10)$

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

Gegeben: $6G = (80, 10)$, $1G = (3, 6)$

$$\lambda = \frac{6-10}{3-80} \pmod{97} = \frac{-4}{-77} \pmod{97} = \frac{4}{77} \pmod{97} \tag{32}$$
$$= 4 \cdot 77^{-1} \pmod{97} = 4 \cdot 19 \pmod{97} = 76 \tag{33}$$

$$x_3 = 76^2 - 80 - 3 \pmod{97} = 5776 - 83 \pmod{97} = 5693 \pmod{97} = 89 \tag{34}$$
$$y_3 = 76 \cdot (80 - 89) - 10 \pmod{97} = 76 \cdot (-9) - 10 \pmod{97} \tag{35}$$
$$= -684 - 10 \pmod{97} = -694 \pmod{97} = 12 \tag{36}$$

Finales Ergebnis: $7G = (89, 12)$

2.4 Verifikation

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

$$y^2 \stackrel{?}{=} x^3 + 3x + 3 \pmod{97} \tag{37}$$
$$12^2 \stackrel{?}{=} 89^3 + 3 \cdot 89 + 3 \pmod{97} \tag{38}$$
$$144 \stackrel{?}{=} 704969 + 267 + 3 \pmod{97} \tag{39}$$
$$47 \stackrel{?}{=} 47 \pmod{97} \quad \checkmark \tag{40}$$

2.5 Kryptographische Bedeutung

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

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

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

- **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

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

Auflistung 1: Grundlegende Zertifikatsinformationen

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

3.3.2 Schritt 2: Strukturierte Analyse

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
```

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

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
```

2. **Certificate Policies:**

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

3. **CRL Distribution Points:**

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

3.4.2 Automatisierte Suche

Ein effizienterer Ansatz zur Flag-Findung:

Auflistung 2: Automatisierte Flag-Suche

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

3.5 Erwartete Flags

- `FLAG{hidden_in_certificate_extensions}`
- `FLAG{x509_extensions_reveal_secrets}`
- `FLAG{certificate_analysis_complete}`
- `FLAG{openssl_reveals_all}`
- `FLAG{mystery_cert_analyzed}`

3.6 Kryptographische Relevanz für Curveball

3.6.1 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

3.6.2 Praktische Sicherheitsimplikationen

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

3.7 Weiterführende Analysetechniken

3.7.1 Erweiterte OpenSSL-Kommandos

Auflistung 3: Erweiterte Zertifikatsanalyse

```
# 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
```

3.7.2 Hexdump-Analyse

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

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

3.8 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

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

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:

Auflistung 4: Skript-Download

```
# Download von der Challenge-Website
wget http://localhost:5000/static/scripts/verification.py

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

4.3.2 Schritt 2: Schwachstelle analysieren

Das Validierungsskript zeigt die kritischen Unterschiede:

Auflistung 5: Vulnerable vs. Secure Validation

```
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!
```

4.3.3 Schritt 3: Manipuliertes Zertifikat erstellen

Methode 1: OpenSSL mit manipulierten Parametern

Auflistung 6: Rogue Key Generation

```
# 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"
```

Methode 2: Python-basierte Manipulation

Auflistung 7: Zertifikat mit Exploit-Markern

```
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))
```

4.3.4 Schritt 4: Zertifikat validieren

Auflistung 8: Exploit-Validierung

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

Erwartete Ausgabe bei erfolgreichem Exploit:

Auflistung 9: Erfolgreiche Validation

```
[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}
```

4.4 Erweiterte Exploit-Techniken

4.4.1 Generator-Punkt Manipulation

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

Auflistung 10: Generator-Punkt Manipulation

```
# 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)}")
```

4.4.2 ASN.1-Manipulation

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

Auflistung 11: ASN.1 ECC Parameter Manipulation

```
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
```

4.5 Lösung und Flag

Bei erfolgreicher Durchführung des Exploits wird die Flag ausgegeben:

`FLAG{curveball_exploit_generator_manipulation_success}`

4.6 Sicherheitsimplikationen

4.6.1 Real-World Impact

Die Curveball-Schwachstelle hatte schwerwiegende Auswirkungen:

- **TLS-Verbindungen:** Komplette Umgehung der Zertifikatsvalidierung
- **Code-Signing:** Malware konnte als vertrauenswürdig eingestuft werden
- **VPN-Verbindungen:** Man-in-the-Middle-Angriffe möglich
- **E-Mail-Verschlüsselung:** Signatur-Validierung kompromittiert

4.6.2 Betroffene Systeme

- Windows 10 (alle Versionen vor KB4534314)
- Windows Server 2016/2019
- Alle Anwendungen, die Windows CryptoAPI verwenden
- Internet Explorer, Microsoft Edge, Office-Suite

5 Challenge 4: Kurvenparameter & 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 Das Kernproblem von CVE-2020-0601

Die kritische Schwachstelle lag in der Art, wie Windows CryptoAPI die Kurvenparameter zur Signaturvalidierung verwendete:

- **Vertrauenswürdige Validierung:** Parameter aus CA-Root-Zertifikat verwenden
- **Verwundbare Validierung:** Parameter aus zu validierendem Zertifikat verwenden
- **Exploit:** Angreifer kontrolliert die Parameter im eigenen Zertifikat

5.2.2 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

5.2.3 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 1: Materialien herunterladen

Die Challenge stellt drei essenzielle Dateien bereit:

Auflistung 12: Download der Challenge-Materialien

```
# Valides Zertifikat mit Standard-Parametern
wget http://localhost:5000/downloads/valid_certificate.pem

# Signatur-Testdaten
wget http://localhost:5000/downloads/signature_data.json

# Validierungsskript
wget http://localhost:5000/scripts/signature_validator.py
```

5.3.2 Schritt 2: Signatur-Testdaten analysieren

Die Datei `signature_data.json` enthält:

Auflistung 13: Analyse der Testdaten

```
# 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\"]}')
"
```

5.3.3 Schritt 3: Normale Validierung testen

Auflistung 14: Baseline-Validierung

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

Erwartete Ausgabe:

Auflistung 15: Normale Validierung

```
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...
```

5.3.4 Schritt 4: Parameter-Manipulation

Methode 1: Generator-Punkt Manipulation

Auflistung 16: Generator-Manipulation

```
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'] = "0x1234567890abcdef1234567890abcdef1234567890abcdef1
manipulated_params['generator_y'] = "0xfedcba0987654321fedcba0987654321fedcba0987654321fe

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

Methode 2: Kurvenparameter-Manipulation

Auflistung 17: Kurvenkoeffizienten ändern

```
# Kurvenparameter 'a' manipulieren
manipulated_params['a'] = "0x0000000000000000000000000000000000000000000000000000000000000000

# Schwaechere Kurve erzeugen
manipulated_params['b'] = "0x0000000000000000000000000000000000000000000000000000000000000000

# Validation mit manipulierten Parametern
from signature_validator import ECCSignatureValidator

validator = ECCSignatureValidator(manipulated_params)
```

Methode 3: Ordnungs-Manipulation

Auflistung 18: Gruppenordnung manipulieren

```
# Manipulierte Ordnung der Kurve
manipulated_params['order'] = "0x100000000000000000000000000000014def9dea2f79cd65812631
```

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

5.3.5 Schritt 5: Exploit-Validierung

Auflistung 19: Vollständiger Exploit

```
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()
```

5.4 Erwartete Ausgabe und Flag

Bei erfolgreichem Exploit gibt das Validierungsskript die Flag aus:

Auflistung 20: Erfolgreicher Exploit

```
[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}
```

5.5 Erweiterte Exploit-Techniken

5.5.1 Mathematische Präzisions-Manipulation

Auflistung 21: Praezise Parameter-Berechnung

```
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
```

5.5.2 ASN.1-Struktur-Manipulation

Auflistung 22: Zertifikat-Parameter überschreiben

```
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"
```

5.6 Sicherheitsimplikationen

5.6.1 Real-World Attack Scenarios

1. TLS-Certificate Spoofing:

- Angreifer erstellt gefälschtes TLS-Zertifikat für banking.example.com
- Manipuliert ECC-Parameter so, dass falsche Signatur gültig wird
- Windows-Clients akzeptieren das Zertifikat als vertrauenswürdig
- Man-in-the-Middle-Angriff erfolgreich

2. Code-Signing Bypass:

- Malware wird mit manipuliertem Code-Signing-Zertifikat signiert
- ECC-Parameter werden so gewählt, dass Windows die Signatur akzeptiert
- Malware wird als vertrauenswürdig eingestuft und ausgeführt
- Umgehung von Application Whitelisting

3. E-Mail Signature Forgery:

- Gefälschte S/MIME-Zertifikate für E-Mail-Signierung
- Manipulation ermöglicht "gültige" Signaturen ohne Private Key
- Phishing-E-Mails erscheinen als von vertrauenswürdigen Absendern

5.6.2 Betroffene Systeme

- **Windows 10:** Alle Versionen vor KB4534314 (Januar 2020)
- **Windows Server:** 2016, 2019, 2008 R2 SP1
- **Anwendungen:** Alle Software die Windows CryptoAPI verwendet
- **Browser:** Internet Explorer, Microsoft Edge (legacy)
- **E-Mail-Clients:** Outlook, Windows Mail
- **Development Tools:** Visual Studio, PowerShell

5.7 Schutzmaßnahmen

5.7.1 Microsoft's Fix

Validierungsverbesserungen:

- **Parameter-Validierung:** Vollständige Prüfung aller ECC-Parameter gegen bekannte Standards
- **Generator-Verifikation:** Generator-Punkt wird gegen vertrauenswürdige CA-Parameter validiert
- **Kurven-Validierung:** Zusätzliche Checks für Kurvenkoeffizienten und mathematische Eigenschaften
- **Order-Verifikation:** Gruppenordnung wird gegen Standardwerte geprüft

5.7.2 Best Practices für Entwickler

Auflistung 23: Sichere Parameter-Validierung

```
def validate_ecc_parameters(params, trusted_params):
    """
    Sichere Validierung von ECC-Parametern
    """
    # 1. Generator-Punkt validieren
    if (params['generator_x'] != trusted_params['generator_x'] or
        params['generator_y'] != trusted_params['generator_y']):
        raise ValueError("Generator-Punkt entspricht nicht trusted Standard")

    # 2. Kurvenparameter check
    if (params['a'] != trusted_params['a'] or
        params['b'] != trusted_params['b']):
        raise ValueError("Kurvenparameter entsprechen nicht Standard")

    # 3. Ordnung validieren
    if params['order'] != trusted_params['order']:
        raise ValueError("Gruppenordnung entspricht nicht Standard")

    # 4. Primzahl checken
    if params['p'] != trusted_params['p']:
        raise ValueError("Grundkoerper entspricht nicht Standard")

    return True

# Verwendung in Signatur-Validierung
def secure_signature_validation(signature, message, public_key, cert_params):
    """Sichere ECDSA-Validierung mit Parameter-check"""

    # KRITISCH: Verwende trusted Parameter, nicht Zertifikat-Parameter
    trusted_params = get_trusted_curve_parameters('secp256r1')

    # Parameter validieren
    validate_ecc_parameters(cert_params, trusted_params)

    # Validierung mit trusted Parametern
    validator = ECCSignatureValidator(trusted_params)  # NICHT cert_params!
    return validator.validate_signature(message, signature.r, signature.s,
```

```
        public_key.x, public_key.y)
```