



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 – 28.07.2025

Contents

1	Einleitung	1
2	Challenge 1: ECC Grundlagen	1
2.1	Aufgabenstellung	1
2.2	Mathematische Grundlagen	1
2.2.1	Punktaddition auf elliptischen Kurven	1
2.3	Schrittweise Lösung	2
2.3.1	Schritt 1: Berechnung von $1G$	2
2.3.2	Schritt 2: Berechnung von $2G = G + G$ (Punktverdopplung)	2
2.3.3	Schritt 3: Berechnung von $4G = 2G + 2G$	3
2.3.4	Schritt 4: Berechnung von $6G = 4G + 2G$	3
2.3.5	Schritt 5: Berechnung von $7G = 6G + 1G$	3
2.4	Verifikation	4
2.5	Kryptographische Bedeutung	4
3	Challenge 2: Zertifikatsanalyse	5
3.1	Aufgabenstellung	5
3.2	Theoretischer Hintergrund	5
3.2.1	X.509-Zertifikatsstruktur	5
3.2.2	Relevante X.509v3 Extensions	6
3.3	Praktische Lösung	6
3.3.1	Schritt 1: Download und erste Inspektion	6
3.3.2	Schritt 2: Strukturierte Analyse	6
3.3.3	Schritt 3: Extension-Analyse	7
3.3.4	Schritt 4: Detaillierte Feldanalyse	7
3.4	Lösungsweg und versteckte Information	7
3.4.1	Flag-Location	7
3.4.2	Automatisierte Suche	8
3.5	Kryptographische Relevanz für Curveball	8
3.5.1	Verbindung zu CVE-2020-0601	8
3.5.2	Praktische Sicherheitsimplikationen	8
3.6	Weiterführende Analysetechniken	8
3.6.1	Erweiterte OpenSSL-Kommandos	8
3.6.2	Hexdump-Analyse	9
3.7	Lernziele erreicht	9
4	Challenge 3: CurveBall Exploit	10
4.1	Aufgabenstellung	10
4.2	Theoretischer Hintergrund	10
4.2.1	CVE-2020-0601 - Die Curveball-Schwachstelle	10
4.2.2	Mathematische Grundlage des Exploits	10
4.3	Praktische Durchführung	11
4.3.1	Schritt 1: Validierungsskript herunterladen	11
4.3.2	Schritt 2: Schwachstelle analysieren	11
4.3.3	Schritt 3: Manipuliertes Zertifikat erstellen	11
4.3.4	Schritt 4: Zertifikat validieren	12
4.4	Erweiterte Exploit-Techniken	13
4.4.1	Generator-Punkt Manipulation	13
4.4.2	ASN.1-Manipulation	13
4.5	Lösung und Flag	13

5	Challenge 4: Kurvenparameter & Signaturvalidierung	14
5.1	Aufgabenstellung	14
5.2	Theoretischer Hintergrund	14
5.2.1	Mathematische Grundlage der ECC-Signaturvalidierung	14
5.2.2	Der Curveball-Exploit	15
5.3	Praktische Durchführung	15
5.3.1	Schritt 2: Signatur-Testdaten analysieren	15
5.3.2	Schritt 3: Normale Validierung testen	15
5.3.3	Schritt 4: Parameter-Manipulation	15
5.3.4	Schritt 5: Exploit-Validierung	16
5.4	Erwartete Ausgabe und Flag	17
5.5	Erweiterte Exploit-Techniken	17
5.5.1	Mathematische Präzisions-Manipulation	17
5.5.2	ASN.1-Struktur-Manipulation	18

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 $\mathbf{P} = 7 \times \mathbf{G}$.

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 $\mathbf{P} = 7 \times \mathbf{G}$ 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 ($P_1 \neq P_2$)

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \pmod{p} \quad (1)$$

$$x_3 = \lambda^2 - x_1 - x_2 \pmod{p} \quad (2)$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p} \quad (3)$$

Fall 2: Punktverdopplung ($P_1 = P_2$)

$$\lambda = \frac{3x_1^2 + a}{2y_1} \pmod{p} \quad (4)$$

$$x_3 = \lambda^2 - 2x_1 \pmod{p} \quad (5)$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \pmod{p} \quad (6)$$

2.3 Schrittweise Lösung**2.3.1 Schritt 1: Berechnung von $1G$** **Lösung Schritt 1**

$$1G = G = (11, 3) \quad (7)$$

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

$$\lambda = \frac{3 \cdot 11^2 + 3}{2 \cdot 3} \pmod{97} \quad (8)$$

$$= \frac{3 \cdot 121 + 3}{6} \pmod{97} \quad (9)$$

$$= \frac{366}{6} \pmod{97} = \frac{75}{6} \pmod{97} \quad (10)$$

$$= 75 \cdot 6^{-1} \pmod{97} \quad (11)$$

Berechnung des modularen Inversen von 12 modulo 97:

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

$$\lambda = 75 \cdot 81 \pmod{97} = 6075 \pmod{97} = 86 \quad (13)$$

$$x_3 = 86^2 - 2 \cdot 11 \pmod{97} = 7396 - 22 \pmod{97} = 13 \quad (14)$$

$$y_3 = 86 \cdot (11 - 13) - 3 \pmod{97} = 86 \cdot (-2) - 3 \pmod{97} \quad (15)$$

$$= -175 \pmod{97} = 69 \quad (16)$$

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

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

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

$$\lambda = \frac{3 \cdot 13^2 + 3}{2 \cdot 89} \pmod{97} \quad (17)$$

$$= \frac{3 \cdot 169 + 3}{138} \pmod{97} \quad (18)$$

$$= \frac{510}{138} \pmod{97} \quad (19)$$

$$= \frac{25}{41} \pmod{97} \quad (\text{nach Reduktion mod } 97) \quad (20)$$

$$= 25 \cdot 41^{-1} \pmod{97} \quad (21)$$

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

$$\lambda = 25 \cdot 19 \pmod{97} = 475 \pmod{97} = 90 \quad (22)$$

$$x_3 = 90^2 - 2 \cdot 13 \pmod{97} = 8100 - 26 \pmod{97} = 8074 \pmod{97} = 39 \quad (23)$$

$$y_3 = 90 \cdot (13 - 39) - 69 \pmod{97} = 90 \cdot (-26) - 69 \pmod{97} \quad (24)$$

$$= -2340 - 69 \pmod{97} = -2409 \pmod{97} = 50 \quad (25)$$

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

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

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

$$\lambda = \frac{50 - 69}{39 - 13} \pmod{97} = \frac{-19}{26} \pmod{97} \quad (26)$$

$$= (-19) \cdot 26^{-1} \pmod{97} = 78 \cdot 56 \pmod{97} \quad (27)$$

$$= 4368 \pmod{97} = 3 \quad (28)$$

$$x_3 = 3^2 - 13 - 39 \pmod{97} = 9 - 52 \pmod{97} = -27 \pmod{97} = 54 \quad (29)$$

$$y_3 = 3 \cdot (13 - 54) - 69 \pmod{97} = 3 \cdot (-41) - 69 \pmod{97} \quad (30)$$

$$= -123 - 69 \pmod{97} = -192 \pmod{97} = 2 \quad (31)$$

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

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

$$\lambda = \frac{3 - 2}{11 - 54} \pmod{97} = \frac{1}{-43} \pmod{97} = \frac{4}{54} \pmod{97} \quad (32)$$

$$= 1 \cdot 54^{-1} \pmod{97} = 1 \cdot 9 \pmod{97} = 9 \quad (33)$$

$$x_3 = 9^2 - 54 - 11 \pmod{97} = 3721 - 65 \pmod{97} = 3656 \pmod{97} = 16 \quad (34)$$

$$y_3 = 9 \cdot (54 - 16) - 2 \pmod{97} = 9 \cdot 38 - 2 \pmod{97} \quad (35)$$

$$= 342 - 2 \pmod{97} = 340 \pmod{97} = 49 \quad (36)$$

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:

$$y^2 \stackrel{?}{=} x^3 + 3x + 3 \pmod{97} \quad (37)$$

$$49^2 \stackrel{?}{=} 16^3 + 3 \cdot 16 + 3 \pmod{97} \quad (38)$$

$$2401 \stackrel{?}{=} 4096 + 48 + 3 \pmod{97} \quad (39)$$

$$73 \stackrel{?}{=} 73 \pmod{97} \quad \checkmark \quad (40)$$

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-Zertifikatsstruktur
- Identifizierung versteckter Informationen in Zertifikatsfeldern
- Vorbereitung auf die Curveball-Schwachstelle (CVE-2020-0601)

3.2 Theoretischer Hintergrund

3.2.1 X.509-Zertifikatsstruktur

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 Schlüsselnverwendungen
- **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:

```
1 # Zertifikatinhaber anzeigen
2 openssl x509 -in mystery_cert.pem -subject -noout
3
4 # Ausstellende CA anzeigen
5 openssl x509 -in mystery_cert.pem -issuer -noout
```

Erwartete Ausgabe:

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

Auflistung 2: Zertifikatsinformationen

3.3.3 Schritt 3: Extension-Analyse

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

```
1 # Alle Extensions anzeigen
2 openssl x509 -in mystery_cert.pem -text -noout |
3 grep -A 20 "X509v3 extensions"
4
5 # Spezifische Suche nach versteckten Informationen
6 openssl x509 -in mystery_cert.pem -text -noout |
7 grep -i -A 5 -B 5 "flag"
```

3.3.4 Schritt 4: Detaillierte Feldanalyse

Subject Alternative Names:

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

Certificate Policies:

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

Authority Information Access:

```
1 openssl x509 -in mystery_cert.pem -text -noout |
2 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:

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

Auflistung 3: SAN Extension mit Flag

2. Certificate Policies:

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

Auflistung 4: Certificate Policies mit Flag

3. CRL Distribution Points:

```
1 X509v3 CRL Distribution Points:
2   Full Name:
3     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:

```
1 # Suche nach Flag-Pattern
2 openssl x509 -in mystery_cert.pem -text -noout |
3 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

```
1 # ASN.1-Struktur anzeigen
2 openssl asn1parse -i -in mystery_cert.pem
3
4 # Modulus und Exponent des public keys
5 openssl x509 -in mystery_cert.pem -noout -modulus
6
7 # Fingerprint-Berechnung
8 openssl x509 -in mystery_cert.pem -noout -fingerprint -sha256
9
10 # PEM zu DER
11 openssl x509 -in mystery_cert.pem -outform DER -out mystery_cert.der
```

Auflistung 7: Erweiterte Zertifikatsanalyse

3.6.2 Hexdump-Analyse

```
1 # Struktur untersuchen
2 xxd mystery_cert.der | grep -i flag
3
4 # String-Extraktion
5 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:

$$e \cdot G = r \cdot G + s \cdot Q \quad (41)$$

Dabei ist:

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

Curveball-Exploit:

$$e \cdot G' = r \cdot G' + s \cdot Q' \quad (42)$$

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:

```
1 # Download von der Challenge-Website
2 wget http://localhost:8443/static/scripts/verification.py
3
4 # Analyse-Modus ausführen
5 python verification.py analyze
```

Auflistung 8: Skript-Download

4.3.2 Schritt 2: Schwachstelle analysieren

Das Validierungsskript zeigt die kritischen Unterschiede:

```
1 Normal ECC Certificate Validation:
2 1. [YES] Verify certificate chain
3 2. [YES] Check certificate validity period
4 3. [YES] Validate certificate signature
5 4. [YES] Verify ECC parameters match standards
6 5. [YES] Ensure generator point is correct
7
8 CVE-2020-0601 Vulnerable Validation:
9 1. [YES] Verify certificate chain
10 2. [YES] Check certificate validity period
11 3. [YES] Validate certificate signature
12 4. [NO] SKIP ECC parameter validation!
13 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

```
1 # Private Key mit NIST P-256 erstellen
2 openssl ecparam -name prime256v1 -genkey -noout -out rogue_key.pem
3
4 # Zertifikat mit "evil" Subject erstellen
5 openssl req -new -x509 -key rogue_key.pem -out exploit_cert.pem \
6 -days 365 -subj "/CN=evil.example.com/O=Evil Corp"
```

Auflistung 10: Rogue Key Generation

Methode 2: Python-basierte Manipulation

```
1 from cryptography import x509
2 from cryptography.hazmat.primitives import hashes, serialization
3 from cryptography.hazmat.primitives.asymmetric import ec
4 import datetime
5
6 # ECC-key generieren
7 private_key = ec.generate_private_key(ec.SECP256R1())
8
9 # Subject und Issuer mit "evil" Markern
10 subject = issuer = x509.Name([
11     x509.NameAttribute(x509.NameOID.COMMON_NAME, u"evil.example.com"),
12     x509.NameAttribute(x509.NameOID.ORGANIZATION_NAME, u"Evil Corp"),
13 ])
14
15 # Zertifikat mit Exploit-Markern erstellen
16 cert = x509.CertificateBuilder().subject_name(
17     subject
18 ).issuer_name(
19     issuer
20 ).public_key(
21     private_key.public_key()
22 ).serial_number(
23     0xdeadbeef # Exploit-Marker
24 ).not_valid_before(
25     datetime.datetime.utcnow()
26 ).not_valid_after(
27     datetime.datetime.utcnow() + datetime.timedelta(days=365)
28 ).sign(private_key, hashes.SHA256())
29
30 # Zertifikat speichern
31 with open("exploit_cert.pem", "wb") as f:
32     f.write(cert.public_bytes(serialization.Encoding.PEM))
```

Aufistung 11: Zertifikat mit Exploit-Markern

4.3.4 Schritt 4: Zertifikat validieren

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

Aufistung 12: Exploit-Validierung

Erwartete Ausgabe bei erfolgreichem Exploit:

```
1 [TARGET] EXPLOIT MARKERS DETECTED:
2 [LIGHTNING] Found: evil
3 [LIGHTNING] Found: deadbeef
4
5 [LAB] SIMULATED WINDOWS CRYPTOAPI VALIDATION:
6 [YES] Certificate format valid
7 [YES] Signature verification (simulated)
8 [WARNING] ECC parameter validation SKIPPED (vulnerable!)
9 [WARNING] Generator point validation SKIPPED (vulnerable!)
10
11 [PARTY] EXPLOIT SUCCESSFUL!
12 The manipulated certificate passed validation!
13
14 [FLAG] FLAG CAPTURED:
15 FLAG{curveball_exploit_generator_manipulation_success}
```

Aufistung 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:

```
1 # Standard NIST P-256 Generator
2 standard_gx = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296
3 standard_gy = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5
4
5 # Manipulierter Generator (mit erkennbarem Marker)
6 rogue_gx = (standard_gx & 0xffffffff00000000) | 0xdeadbeef
7 rogue_gy = (standard_gy & 0xffffffff00000000) | 0xcafebabe
8
9 print(f"Standard Generator X: {hex(standard_gx)}")
10 print(f"Rogue Generator X: {hex(rogue_gx)}")
11 print(f"Standard Generator Y: {hex(standard_gy)}")
12 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:

```
1 import asn1crypto.x509
2 import asn1crypto.algos
3
4 # Manipulierte ECC-Parameter definieren
5 ecc_params = {
6     'named_curve': 'prime256v1', # Malicious: Standard-Kurve angeben
7     # Manipulierte Parameter werden verwendet
8 }
9
10 # In realer Implementierung werden hier die ASN.1-Strukturen
11 # 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:**

$$u_1 = e \cdot w \pmod{n} \quad (43)$$

$$u_2 = r \cdot w \pmod{n} \quad (44)$$

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:

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

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:

```
1 # JSON-Datei untersuchen
2 cat signature_data.json | python -m json.tool
3
4 # Wichtige Felder extrahieren
5 python -c "
6 import json
7 with open('signature_data.json') as f:
8     data = json.load(f)
9     sig = data['signature_data']['original_signature']
10    print(f'Signatur r: {sig["r"]}')
11    print(f'Signatur s: {sig["s"]}')
12 "
```

Auflistung 16: Analyse der Testdaten

5.3.2 Schritt 3: Normale Validierung testen

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

Auflistung 17: Baseline-Validierung

Erwartete Ausgabe:

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

Auflistung 18: Normale Validierung

5.3.3 Schritt 4: Parameter-Manipulation

Methode 1: Generator-Punkt Manipulation

```

1 import json
2
3 # Standard NIST P-256 Parameter laden
4 with open('signature_data.json') as f:
5     data = json.load(f)
6
7 original_params = data['signature_data']['original_curve_params']
8
9 # Manipulierte Parameter erstellen
10 manipulated_params = original_params.copy()
11
12 # Generator-Punkt manipulieren
13 manipulated_params['generator_x'] = "0
14     x1234567890abcdef1234567890abcdef1234567890abcdef1234567890abcdef"
15 manipulated_params['generator_y'] = "0
16     xfedcba0987654321fedcba0987654321fedcba0987654321fedcba0987654321"
17
18 print("Manipulierte Parameter:")
19 for key, value in manipulated_params.items():
20     print(f"{key}: {value}")

```

Auflistung 19: Generator-Manipulation

Methode 2: Kurvenparameter-Manipulation

[illegible]

Auflistung 20: Kurvenkoeffizienten ändern

Methode 3: Ordnungs-Manipulation

[illegible]

Auflistung 21: Gruppenordnung manipulieren

5.3.4 Schritt 5: Exploit-Validierung

```

1 from signature_validator import ECCSignatureValidator
2
3 # Lade Signatur-Daten
4 with open('signature_data.json') as f:
5     data = json.load(f)
6
7 sig_data = data['signature_data']
8 message = sig_data['message']

```

```

9 signature_r = int(sig_data['original_signature']['r'], 16)
10 signature_s = int(sig_data['original_signature']['s'], 16)
11
12 # Simuliere Public Key (vereinfacht)
13 original_params = sig_data['original_curve_params']
14 public_key_x = int(original_params['generator_x'], 16)
15 public_key_y = int(original_params['generator_y'], 16)
16
17 # Test mit manipulierten Parametern
18 for example in sig_data['manipulated_examples']:
19     print(f"Testing: {example['method']}")
20
21     # Manipulierte Parameter erstellen
22     manipulated_params = original_params.copy()
23
24     if 'new_generator_x' in example:
25         manipulated_params['generator_x'] = example['new_generator_x']
26     if 'new_generator_y' in example:
27         manipulated_params['generator_y'] = example['new_generator_y']
28     if 'new_a' in example:
29         manipulated_params['a'] = example['new_a']
30     if 'new_order' in example:
31         manipulated_params['order'] = example['new_order']
32
33     # Validierung mit manipulierten Parametern
34     validator = ECDSignatureValidator(manipulated_params)
35     result = validator.validate_signature(
36         message, signature_r, signature_s, public_key_x, public_key_y
37     )
38
39     print(f"Result: {result['valid']}")
40     print(f"Reason: {result['reason']}")
41     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:

```

1 [EXPLOIT] Parameter-Manipulation erfolgreich!
2 [TARGET] Generator-Punkt Manipulation durchgefuehrt
3 [SUCCESS] Urspruenglich ungueltige Signatur wird als gueltig erkannt
4 [WARNING] CVE-2020-0601 Schwachstelle ausgenutzt
5
6 [FLAG] FLAG CAPTURED:
7 FLAG{curve_parameter_manipulation_signature_bypass}

```

Auflistung 23: Erfolgreicher Exploit

5.5 Erweiterte Exploit-Techniken

5.5.1 Mathematische Präzisions-Manipulation

```

1 def calculate_exploit_parameters(target_r, target_s, message_hash):
2     """
3     Berechnet manipulierte Parameter fuer gewuenschte Signatur-Validierung
4     """
5     # Ziel: Finde G' so dass u1*G' + u2*Q = (target_r, y) fuer beliebiges y
6
7     # s_inv berechnen
8     s_inv = pow(target_s, -1, ORDER)
9
10    # u1, u2 berechnen
11    u1 = (message_hash * s_inv) % ORDER
12    u2 = (target_r * s_inv) % ORDER
13
14    # Neuen Generator berechnen der gewuenshtes Ergebnis liefert
15    # Dies ist eine vereinfachte Darstellung der komplexen Mathematik
16
17    manipulated_gx = (target_r - u2 * PUBLIC_KEY_X) * pow(u1, -1, ORDER) % ORDER
18    manipulated_gy = calculate_curve_point_y(manipulated_gx)
19
20    return manipulated_gx, manipulated_gy
21
22 def calculate_curve_point_y(x, a=CURVE_A, b=CURVE_B, p=CURVE_P):
23     """Berechnet y-Koordinate fuer gegebenen x-Wert auf elliptischer Kurve"""
24     y_squared = (pow(x, 3, p) + a * x + b) % p
25     y = pow(y_squared, (p + 1) // 4, p) # Vereinfacht fuer p = 3 (mod 4)
26     return y

```

Auflistung 24: Praezise Parameter-Berechnung

5.5.2 ASN.1-Struktur-Manipulation

```

1 from cryptography import x509
2 from cryptography.hazmat.primitives import serialization
3
4 def create_manipulated_certificate(original_cert_path, manipulated_params):
5     """
6     Erstellt Zertifikat mit manipulierten ECC-Parametern
7     """
8     # Originales Zertifikat laden
9     with open(original_cert_path, 'rb') as f:
10         original_cert = x509.load_pem_x509_certificate(f.read())
11
12     # ASN.1-Struktur manipulieren (vereinfacht)
13     # In realer Implementierung wuerde hier die komplette
14     # ASN.1-Struktur der ECC-Parameter ueberschrieben
15
16     print("Manipulierte ECC-Parameter in Zertifikat:")
17     print(f"Generator X: {manipulated_params['generator_x']}")
18     print(f"Generator Y: {manipulated_params['generator_y']}")
19     print(f"Curve A: {manipulated_params['a']}")
20     print(f"Order: {manipulated_params['order']}")
21
22     return "manipulated_certificate.pem"

```

Auflistung 25: Zertifikat-Parameter überschreiben