Kryptoanalyse Laborarbeit

Aufgabe 2

2024-10-27

`padding_oracle`: CBC Padding Oracle Angriff

In der Vorlesung haben Sie die Funktionsweise ein PKCS#7 Padding Oracle Angriffs kennengelernt. Wir definieren folgendes Binärprotokoll, mit welchem ein Client und Server kommuniziert; der Server exponiert hierbei eine Padding Oracle Verwundbarkeit.

1. Client → Server: Initial einmalig: 16 Bytes Ciphertext
2. Client → Server: 2 Bytes Längenfeld l, welches angibt, wie viele Q-Blöcke folgen. Encoding little endian, $0 \leq l \leq 256$. Bei $l = 0$ terminiert der Server die Verbindung.
3. Client → Server: $16 \cdot l$ Bytes Q-Blöcke, welche der Server zusammen mit dem Ciphertext nutzen soll, um ein Depadding zu testen.
4. Server → Client: l Bytes Antwort, jeweils `00` wenn das respektive Padding inkorrekt war und `01` wenn es korrekt war.
5. Zurück zu Schritt (2)

Beispiel:

1. Client → Server: `2d5cb7e89318651744664fbee709e147`
2. Client → Server: `0300`
3. Client → Server: `00000000000000000000000000000000`
4. Client → Server: `00000000000000000000000000000001`
5. Client → Server: `00000000000000000000000000000002`
6. Server → Client: `000000`
7. Client → Server: `0200`
8. Client → Server: `00000000000000000000000000000003`
9. Client → Server: `00000000000000000000000000000004`
10. Server → Client: `0100`

In dem angegebenen Beispiel war das Padding also korrekt, wenn der Q-Block `00000000000000000000000000000003` war, in den anderen 4 Fällen war es inkorrekt.
Ihnen wird als Testcase der Hostname und Port des so implementierten Servers gegeben. Zusätzlich erhalten Sie eine Nachricht m, welche immer ein Vielfaches der verwendeten Blockgröße (16 Bytes) lang ist und einen IV, mit dem die Nachricht ursprünglich verschlüsselt wurde. Beipsielsweise:

```
{
    "action": "padding_oracle",
    "arguments": {
        "hostname": "padoraserv",
        "port": 18652,
        "iv": "dxTwbO/hhIeycOTbTnp8QQ==",
        "ciphertext": "e+Sn+nG28niB8Df++hmjFRTcti07wHsrivmoxn
                       DDBaELOfLS16p/pqvAuz01UPq7"
    }
}
```

Sie verbinden sich nun zum Padding Oracle Server unter dem gegebenen Hostnamen/port und verwenden das vorgegebene Binärprotokoll, um den Ciphertext zu entschlüsseln. Danach geben Sie den Plaintext *ohne Entfernung von Padding* zurück.

```
{
    "plaintext": "RGFzIGlzdCBlaW5lIGtvcnJla3QgZW50c2NobHVlc3N
                  lbHRlIE5hY2hyaWNodAIC"
}
```

Achten Sie auf alle Spezialfälle (insbesondere Mehrdeutigkeit beim Entschlüsseln des ersten Bytes jedes Blocks). Ihre Lösung ist zeitlich stark limitiert, d.h. Sie müssen eine performante Implementierung leisten. Bei lokaler Ausführung terminiert die Referenzlösung (Python) für einen Block in unter 5ms. Sie haben ein Mehrfaches dieses Zeitkontingents zur Verfügung.

GCM-Erweiterungen `block2poly` und `poly2block`

Erweitern Sie Ihre Implementierungen von `block2poly` und `poly2block` dementsprechend, dass Sie auch mit Blöcken der Semantik „gcm“, wie in der Vorlesung beschrieben, umgehen können.

GCM-Erweiterung von `gfmul`

Erweitern Sie Ihre Implementierung von `gfmul` derart, dass Sie auch mit Blöcken der Semantik „gcm“, wie in der Vorlesung beschrieben, umgehen kann. Weiterhin soll `gfmul` zwei beliebige Blöcke miteinander multiplizieren können, d.h. es ist Ihnen nun nicht mehr garantiert dass einer der Blöcke α ist.

`gcm_encrypt`: GCM-Verschlüsselung

Entwickeln Sie eine eigene GCM-Verschlüsselung, die sowohl für AES128 und SEA128 funktioniert. Verwenden Sie ausdrücklich nicht die vorgegebene GCM-Implementierung Ihrer kryptografischen Bibliothek. Achten Sie darauf, für Galoisfeldoperationen jeweils die GCM-Konvertierung zu nutzen.

```
{
    "action": "gcm_encrypt",
    "arguments": {
        "algorithm": "aes128",
        "nonce": "4gF+BtR3ku/PUQci",
        "key": "Xjq/GkpTSWoe3ZH0F+tjrQ==",
        "plaintext": "RGFzIGlzdCBlaW4gVGVzdA==",
        "ad": "QUQtRGF0ZW4="
    }
}
```

Als erwartete Antwort soll der Plaintext verschlüsselt und authentifiziert werden, also der `ciphertext` sowie der Authentisierungs-Tag als `tag` zurückgeliefert werden. Darüberhinaus geben Sie die Blöcke L und H zurück.

```
{
    "ciphertext": "ET3RmvH/Hbuxba63EuPRrw==",
    "tag": "Mp0APJb/ZIURRwQlMgNN/w==",
    "L": "AAAAAAAAAEAAAAAAAAAAgA==",
    "H": "Bu6ywbsUKlpmZXMQyuGAng=="
}
```

Die Verschlüsselung mit SEA128 funktioniert völlig analog, nur dass statt AES128 eben SEA128 genutzt wird:

```
{
    "action": "gcm_encrypt",
    "arguments": {
        "algorithm": "sea128",
        "nonce": "4gF+BtR3ku/PUQci",
        "key": "Xjq/GkpTSWoe3ZH0F+tjrQ==",
        "plaintext": "RGFzIGlzdCBlaW4gVGVzdA==",
        "ad": "QUQtRGF0ZW4="
    }
}
```

Hier lautet die erwartete Antwort:

```
{
    "ciphertext": "0cI/Wg4R3URfrVFZ0hw/vg==",
    "tag": "ysDdzOSnqLH0MQ+Mkb23gw==",
    "L": "AAAAAAAAAEAAAAAAAAAAgA==",
    "H": "xhFcAUT66qWIpYz+Ch5ujw=="
}
```

`gcm_decrypt`: GCM-Entschlüsselung

Als Gegenstück zur Verschlüsselung soll ebenfalls die GCM-Entschlüsselung von AES128-GCM sowie SEA128-GCM implementiert werden:

```
{
    "action": "gcm_decrypt",
    "arguments": {
        "algorithm": "aes128",
        "nonce": "SwVYf3IJ2p/VTiyz",
        "key": "bYfxz4zIS8NGWT55xSGy7Q==",
        "ciphertext": "ltCrCEYjUCHd",
        "ad": "w8pc8/yCt2zxERPVcsnOMx8/HmrfAfGUQtD+vyMMpJ5lrF2S",
        "tag": "d4Z2uVRSmpVE1TEa/Zhx9A=="
    }
}
```

Und die erwartete Antwort:

```
{
    "authentic": true,
    "plaintext": "EREavadt90qq"
}
```

Beziehungsweise:

```
{
    "action": "gcm_decrypt",
    "arguments": {
        "algorithm": "sea128",
        "nonce": "VOkKCCnH4EYE1z4L",
        "key": "ByMrTiLP7isfBDL7vsKkOQ==",
        "ciphertext": "UdpDzPAafM+y",
        "ad": "UknNF3AKBaF/8GUnFUw=",
        "tag": "sNO+1fG+WSOHMswF7IBnZA=="
    }
}
```

Und die erwartete Antwort:

```
{
    "authentic": false,
    "plaintext": "AxSiKm93Gr2+"
}
```