Sichere VoIP-Kommunikation - Protokollspezifikation

Entwicklerversion 2.0

7. Oktober 2025

Zusammenfassung

Dieses Dokument beschreibt das erweiterte Protokoll für sichere VoIP-Kommunikation mit Fokus auf experimentelle Sicherheitsfeatures, RNG-Verifikation und Angriffserkennung. Das Protokoll lässt bewusst bestimmte Angriffsvektoren offen, um Angreiferverhalten zu analysieren.

Inhaltsverzeichnis

1 Protokoll-Übersicht

1.1 Kernprinzipien

- **SIP-basiert** mit erweiterten Sicherheitsfeatures
- **Einheitliches Framing** für alle Nachrichten (4-Byte Header + Body)
- **Verify-Code System** zum Schutz gegen Replay-Angriffe
- **Quantenresistente Hashes** (SHA3-256) für Merkle Trees
- **Hybride Verschlüsselung** (RSA-4096 + AES-256-CBC)

2 RNG-Verifikationssystem

2.1 Zweck und Design

Das Verify-Code-System schützt SIP-Nachrichten, die von Natur aus unsicher sind. In der Experimentierphase wird der Seed (Client-Name) bewusst nicht geheim gehalten, um Angriffe auf den Initialisierungsvektor zu ermöglichen.

2.2 Technische Implementierung

```python
class VerifyGenerator:
    def __init__(self, seed, client_id=None):
        self.seed = str(seed)  # Öffentlicher Client-Name
        self.client_id = client_id
        self.counter = 0
        self._lock = threading.Lock()

    def generate_verify_code(self):
        base_string = f"{self.seed}:{self.counter}"
        hash_obj = hashlib.sha256(base_string.encode())
        code = hash_obj.hexdigest()[:4]
        self.counter += 1
        return code
```

Listing 1: Verify-Code-Generator

2.3 Angriffserkennung

```python
def verify_code_with_detection(received_code, client_info):
    # Normale Validierung zuerst
    if generator.verify_code(received_code):
        return "valid"

    # Prüfe auf Wiederverwendung (RNG-State-Angriff)
    for offset in range(1, 100):
        test_counter = current_counter - offset
        expected_code = calculate_expected_code(test_counter)
        if received_code == expected_code:
            log_rng_attack(client_info, received_code)
            return "reused"

    return "invalid"
```

Listing 2: Erweiterte Verify-Code-Validierung

2.4 Protokollierung von Angriffen

```python
def log_rng_attack(client_info, code):
    log_entry = (
        "  [RNG_STATE_ATTACK] VERIFY_CODE_REUSE\n"
        f"   Client: {client_info}\n"
        f"   Code: {code}\n"
        f"   ANALYSIS: Angreifer verwendet bekannten RNG State!\n"
        f"   ANALYSIS: Code wurde doppelt verwendet!\n"
    )
    with open("attacks.log", "a") as f:
        f.write(log_entry)
```

Listing 3: Angriffsprotokollierung

3 Sicherheitsfeatures

3.1 Einheitliches Framing

```python
def send_frame(sock, data):
    # 4-Byte Header (Network Byte Order)
    header = struct.pack('!I', len(data))
    full_message = header + data
    return sock.sendall(full_message)
```

Listing 4: Frame-Format

3.2 Quantensichere Merkle Trees

```python
def build_merkle_tree(data_blocks):
    tree = [quantum_safe_hash(block) for block in data_blocks]
    while len(tree) > 1:
        if len(tree) % 2 != 0:
            tree.append(tree[-1])
        tree = [quantum_safe_hash(tree[i] + tree[i+1])
                for i in range(0, len(tree), 2)]
    return tree[0]
```

Listing 5: Merkle-Tree-Berechnung

3.3 Hybride Verschlüsselung

```python
def encrypt_phonebook_data(phonebook_json, client_public_key):
    # 1. 48-Byte Secret generieren (16B IV + 32B AES Key)
    secret = generate_secret()

```

```
    # 2. Secret mit RSA verschlüsseln
    encrypted_secret = rsa_encrypt(secret, client_public_key)

    # 3. Daten mit AES verschlüsseln
    encrypted_data = aes_encrypt(phonebook_json, secret)

    return encrypted_secret, encrypted_data
```

Listing 6: Verschlüsselungsprozess

4 Kommunikationsabläufe

4.1 Client-Registration

1. Client sendet REGISTER mit Public Key und Verify-Code
2. Server validiert Verify-Code und registriert Client
3. Server sendet Merkle Root aller Public Keys
4. Client validiert Merkle Tree Integrität

4.2 Anrufinitiierung

1. Client A fordert Public Key von Client B
2. Server liefert Public Key mit Verify-Code
3. Client A verschlüsselt Session-Key mit Client B's Public Key
4. Server leitet verschlüsselte Daten an Client B weiter
5. Bei Annahme: Bidirektionale Audio-Streams werden etabliert

5 UDP-Audio-Relay

5.1 Architektur

```
class CONVEY:
    def __init__(self, server_instance):
        self.udp_relay_port = 51822
        self.audio_relays = {}
        self._start_udp_relay()

    def _register_audio_relay(self, call_id, caller_ip, callee_ip):
        self.audio_relays[call_id] = {
            'caller_addr': (caller_ip, 51821),
            'callee_addr': (callee_ip, 51821)
        }
```

Listing 7: UDP-Relay-Setup

5.2 Datenfluss

1. Audio-Daten werden lokal mit AES-256-CBC verschlüsselt
2. Verschlüsselte Pakete werden an UDP-Relay gesendet
3. Relay leitet Pakete an Zielclient weiter
4. Empfänger entschlüsselt lokal mit Session-Key

6 Server-Discovery und Load-Balancing

6.1 Seed-Server Architektur

```python
class AccurateRelayManager:
    def __init__(self, server_instance):
        self.SEED_SERVERS = [
            ("sichereleitung.duckdns.org", 5060),
            ("sichereleitung.duckdns.org", 5061)
        ]
        self.known_servers = {}
        self.server_load = 0
```

Listing 8: Relay-Manager

6.2 Client-seitige Server-Auswahl

```python
class ClientRelayManager:
    def discover_servers(self):
        for seed_host, seed_port in self.SEED_SERVERS:
            # Ping Server auf beiden Ports (5060/5061)
            # Wähle Server mit niedrigster Last und Ping-Zeit
            best_server = self._select_best_server()
```

Listing 9: Server-Discovery

7 Erweiterte Audio-Features

7.1 Qualitätsprofile

```python
QUALITY_PROFILES = {
    "highest": {"format": "S32_LE", "rate": 192000, "channels": 2},
    "high": {"format": "24-bit", "rate": 192000, "channels": 2},
    "middle": {"format": "24-bit", "rate": 48000, "channels": 2},
    "low": {"format": "16-bit", "rate": 48000, "channels": 1}
}
```

Listing 10: Audio-Konfiguration

7.2 Rauschfilterung

1. 180-Sekunden Clear-Room Profilerstellung
2. Adaptive Rauschschwellen basierend auf RMS
3. Spektrale Rauschunterdrückung ohne scipy Abhängigkeit
4. Frequenzspezifische Filter für 50Hz Brummen

8 Sicherheitsanalyse

8.1 Bewusst offene Angriffsvektoren

- **Verify-Code Seed**: Client-Name ist öffentlich → RNG-State-Angriffe möglich
- **SIP-Header**: Traditionelle SIP-Verwundbarkeiten bleiben bestehen
- **Timing-Angriffe**: Keine zusätzlichen Timing-Gegenmaßnahmen

8.2 Erkennungsmechanismen

- **RNG-State-Angriffe**: Verify-Code Wiederverwendung wird protokolliert
- **Replay-Angriffe**: Counter-System verhindert Replay innerhalb Toleranz
- **Man-in-the-Middle**: Merkle Tree validiert Schlüsselintegrität

8.3 Protokollierte Angriffstypen

```
# attacks.log Einträge:
 [RNG_STATE_ATTACK] VERIFY_CODE_REUSE
 [INVALID] INVALID_VERIFY_CODE
 [MALFORMED] MALFORMED_VERIFY_CODE
 [MISSING] MISSING_VERIFY_CODE
 [FRAME] OVERSIZE (DDoS-Versuch)
```

Listing 11: Angriffsprotokoll-Beispiele

9 Performance-Optimierungen

9.1 Thread-Safety

```python
class VerifyGenerator:
    def __init__(self, seed, client_id):
        self._lock = threading.Lock()  # Pro Client-Instanz

    def verify_code(self, received_code):
        with self._lock:  # Thread-sicher
            # Verifikationslogik
```

Listing 12: Thread-sichere Datenstrukturen

9.2 Queue-basierte Verarbeitung

```python
def handle_server_message(self, raw_data):
    if not hasattr(self, '_message_queue'):
        self._message_queue = []

    self._message_queue.append({
        'type': 'frame_data',
        'data': raw_data,
        'timestamp': time.time()
    })

    # Asynchrone Verarbeitung
    self._process_queue_simple()
```

Listing 13: Nachrichten-Queue

10 Entwicklerhinweise

10.1 Experimentelle Features

- RNG-Verifikation dient primär der Angriffserkennung, nicht -verhinderung
- Bewusst einfache Implementierung für Analysezwecke
- Extensive Protokollierung in `attacks.log` für Forensik

10.2 Produktionseinsatz

Für Produktionseinsatz empfohlen:

- Gemeinsames Geheimnis für Verify-Code Seed
- Erhöhte Verify-Code Länge (8+ Zeichen)
- Strengere Counter-Toleranz
- Zusätzliche Timing-Gegenmaßnahmen

Literatur

[1] Rosenberg, J. *SIP: Session Initiation Protocol.* RFC 3261.

[2] Merkle, R. C. *A Digital Signature Based on a Conventional Encryption Function.* CRYPTO 1987.

[3] Dworkin, M. J. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions.* FIPS PUB 202.