pyMessenger - Système de Messagerie Chiffrée de Bout en Bout

Rapport Technique

Contexte du Projet

pyMessenger est une application de messagerie sécurisée que j'ai développée pour démontrer une compréhension approfondie des protocoles cryptographiques modernes et de leur implémentation pratique. Le projet illustre la conception d'un système de communication chiffré de bout en bout (E2EE) utilisant une architecture client-serveur, avec un accent particulier sur la sécurité des données et l'authentification robuste.

L'objectif principal était de créer un système où même le serveur ne peut pas déchiffrer les communications des utilisateurs, tout en maintenant une expérience utilisateur fluide et en implémentant des mécanismes de sécurité contre les attaques courantes.

Architecture Cryptographique

Modèle de Chiffrement Hybride

Le système repose sur une approche de **chiffrement hybride** qui combine les avantages du chiffrement asymétrique et symétrique :

Chiffrement asymétrique (RSA-2048) :

- Utilisé pour l'échange sécurisé des clés de session
- Chaque utilisateur génère une paire de clés RSA côté client
- La clé privée reste exclusivement sur le poste de l'utilisateur
- La clé publique est partagée via le serveur

Chiffrement symétrique (AES-256) :

- Utilisé pour chiffrer le contenu des messages
- Mode EAX (Encrypt-then-Authenticate-then-translate) pour l'authentification intégrée
- Génération d'une nouvelle clé AES pour chaque message
- Performance optimale pour les données volumineuses

Processus de Chiffrement d'un Message

Voici le flux complet d'envoi d'un message chiffré :

```
# 1. Génération d'une clé AES aléatoire (256 bits)
aes_key = get_random_bytes(32)

# 2. Chiffrement du message avec AES-256-EAX
aes_cipher = AES.new(aes_key, AES.MODE_EAX)
```

```
ciphertext, tag = aes_cipher.encrypt_and_digest(plaintext)
nonce = aes_cipher.nonce

# 3. Chiffrement de la clé AES pour chaque destinataire
keys_map = {}
for target in targets:
    recipient_public_key = peer_keys.get(target)
    rsa_cipher = PKCS1_OAEP.new(recipient_public_key)
    encrypted_key = rsa_cipher.encrypt(aes_key)
    keys_map[target] = base64.b64encode(encrypted_key)

# 4. Construction de l'enveloppe cryptographique
envelope = {
    "type": "encrypted_send",
    "from": sender_username,
    "targets": list(keys_map.keys()),
    "ciphertext": base64.b64encode(ciphertext),
    "nonce": base64.b64encode(nonce),
    "tag": base64.b64encode(tag),
    "keys": keys_map
}
```

Ce processus garantit que :

- Le serveur ne voit jamais la clé AES en clair
- Chaque destinataire peut déchiffrer indépendamment
- L'authentification du message est assurée par le tag EAX
- Une clé compromise n'affecte qu'un seul message

Déchiffrement Côté Récipiendaire

Le processus inverse est tout aussi sécurisé :

```
# 1. Extraction de la clé AES chiffrée pour ce destinataire
encrypted_aes_key = base64.b64decode(envelope['key'])

# 2. Déchiffrement avec la clé privée RSA
rsa_cipher = PKCS1_OAEP.new(private_rsa_key)
aes_key = rsa_cipher.decrypt(encrypted_aes_key)

# 3. Reconstruction du chiffreur AES
ciphertext = base64.b64decode(envelope['ciphertext'])
nonce = base64.b64decode(envelope['nonce'])
tag = base64.b64decode(envelope['tag'])

aes_cipher = AES.new(aes_key, AES.MODE_EAX, nonce=nonce)

# 4. Déchiffrement et vérification de l'authenticité
plaintext = aes_cipher.decrypt_and_verify(ciphertext, tag)
```

La méthode `decrypt_and_verify()` lève une exception si le tag ne correspond pas, détectant ainsi toute tentative de modification du message.

Système d'Authentification Sécurisé

Protocole Challenge-Response

L'une des fonctionnalités les plus intéressantes du projet est le système d'authentification par défi-réponse, qui évite la transmission du mot de passe après l'inscription initiale.

Flux d'authentification complet :

```
# CÔTÉ CLIENT - Étape 1 : Demande de connexion
auth_request = {
    'type': 'auth_request',
    'auth_type': 'login',
    'username': username,
    'pubkey': base64.b64encode(public_key_bytes)
}

# CÔTÉ SERVEUR - Étape 2 : Génération du défi
nonce = base64.b64encode(get_random_bytes(32))  # Nonce aléatoire
salt = user_stored_salt  # Sel stocké lors de l'inscription

challenge = {
    'type': 'auth_challenge',
    'nonce': nonce,
    'salt': salt
}

# CÔTÉ CLIENT - Étape 3 : Calcul de la réponse
# Dérivation de la clé à partir du mot de passe
password_key = PBKDF2(
    password.encode('utf-8'),
    base64.b64decode(salt),
    32,
    count=100000,
    hmac_hash_module=SHA256
)

# Signature HMAC du nonce
response_hash = hmac.new(
    password_key,
    nonce.encode('utf-8'),
    hashlib.sha256
).digest()

response = {
    'type': 'auth_response',
    'response': base64.b64encode(response_hash)
}
```

```
# CÔTÉ SERVEUR - Étape 4 : Vérification
stored_password_key = get_stored_password_key(username)
expected_response = hmac.new(
    stored_password_key,
    nonce.encode('utf-8'),
    hashlib.sha256
).digest()

# Comparaison en temps constant (anti timing-attack)
is_valid = hmac.compare_digest(
    received_response,
    base64.b64encode(expected_response)
)
```

Avantages de cette approche :

1. **Zéro transmission de mot de passe** : Après l'inscription, le mot de passe ne transite jamais sur le réseau
2. **Protection contre le rejeu** : Chaque nonce est unique et expire après 5 minutes
3. **Résistance aux attaques temporelles** : Utilisation de `hmac.compare_digest()` pour des comparaisons en temps constant
4. **Dérivation robuste** : PBKDF2 avec 100 000 itérations rend le brute-force coûteux

Mécanismes de Protection Supplémentaires

Le système implémente plusieurs couches de sécurité contre les attaques d'authentification :

```
class UserStore:
    def __init__(self):
        # Limitation du taux de tentatives
        self.login_attempts = defaultdict(list)
        self.locked_accounts = {}

        self.MAX_LOGIN_ATTEMPTS = 5
        self.LOCKOUT_DURATION = 900  # 15 minutes
        self.RATE_LIMIT_WINDOW = 300  # 5 minutes

    def _record_failed_login(self, username, ip_address=None):
        """Enregistre une tentative échouée et verrouille si nécessaire."""
        current_time = time.time()

        # Nettoyage des anciennes tentatives
        self.login_attempts[username] = [
            t for t in self.login_attempts[username]
            if current_time - t < self.RATE_LIMIT_WINDOW
        ]

        self.login_attempts[username].append(current_time)

        # Verrouillage après 5 tentatives
```

```
        if len(self.login_attempts[username]) >= self.MAX_LOGIN_ATTEMPTS:
            unlock_time = current_time + self.LOCKOUT_DURATION
            self.locked_accounts[username] = unlock_time
            self.logger.warning(
                f"Compte verrouillé : {username} (IP: {ip_address})"
            )
            return True
        return False
```

Cette implémentation protège contre :

- Les attaques par force brute
- Les attaques par dictionnaire
- Les tentatives automatisées massives

Gestion Sécurisée des Clés

Stockage Local des Clés Privées

Les clés privées RSA ne sont jamais transmises au serveur. Elles sont stockées localement avec un chiffrement basé sur le mot de passe de l'utilisateur :

```
def save_private_key(self, username, private_key, password):
    """Sauvegarde la clé privée chiffrée sur disque."""
    # Génération d'un sel unique
    salt = get_random_bytes(16)

    # Dérivation de la clé de chiffrement depuis le mot de passe
    encryption_key = PBKDF2(
        password.encode('utf-8'),
        salt,
        32,
        count=100000,
        hmac_hash_module=SHA256
    )

    # Chiffrement AES-GCM de la clé privée
    cipher = AES.new(encryption_key, AES.MODE_GCM)
    ciphertext, tag = cipher.encrypt_and_digest(private_key)

    # Structure du fichier : salt || nonce || tag || ciphertext
    key_file = self.keys_dir / f"{username}_private.key"
    with open(key_file, 'wb') as f:
        f.write(salt + cipher.nonce + tag + ciphertext)

    # Permissions restrictives (Unix)
    if os.name != 'nt':
        os.chmod(key_file, 0o600)  # Lecture/écriture propriétaire uniquement
```

Points clés de cette approche :

- Le mot de passe n'est jamais stocké, seule la clé dérivée protège la clé privée
- Chaque clé a son propre sel, rendant les rainbow tables inefficaces
- Le mode GCM garantit l'intégrité (détection de modification du fichier)
- Les permissions système limitent l'accès aux fichiers sensibles

Chargement et Vérification

```
def load_private_key(self, username, password):
    """Charge et déchiffre la clé privée depuis le disque."""
    key_file = self.keys_dir / f"{username}_private.key"

    try:
        with open(key_file, 'rb') as f:
            data = f.read()
            salt = data[:16]
            nonce = data[16:32]
            tag = data[32:48]
            ciphertext = data[48:]

        # Dérivation de la clé de chiffrement
        encryption_key = PBKDF2(
            password.encode('utf-8'),
            salt,
            32,
            count=100000,
            hmac_hash_module=SHA256
        )

        # Déchiffrement et vérification
        cipher = AES.new(encryption_key, AES.MODE_GCM, nonce=nonce)
        private_key = cipher.decrypt_and_verify(ciphertext, tag)

        return RSA.import_key(private_key)
    except Exception:
        return None  # Mot de passe incorrect ou fichier corrompu
```

Un mauvais mot de passe lève une exception lors de `decrypt_and_verify()`, ce qui empêche l'importation de la clé.

Couche de Transport Sécurisée (TLS/SSL)

En plus du chiffrement de bout en bout, le système utilise TLS pour protéger les métadonnées et prévenir les attaques de type man-in-the-middle au niveau du transport.

Génération des Certificats

```
def generate_self_signed_cert(cert_dir):
    """Génère un certificat auto-signé pour le serveur."""
    # Génération de la clé privée
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )

    # Construction du certificat
    subject = issuer = x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, u"SN"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"pyMessenger"),
        x509.NameAttribute(NameOID.COMMON_NAME, u"localhost"),
    ])

    cert = x509.CertificateBuilder().subject_name(
        subject
    ).issuer_name(
        issuer
    ).public_key(
        private_key.public_key()
    ).serial_number(
        x509.random_serial_number()
    ).not_valid_before(
        datetime.utcnow()
    ).not_valid_after(
        datetime.utcnow() + timedelta(days=365)
    ).add_extension(
        x509.SubjectAlternativeName([
            x509.DNSName(u"localhost"),
            x509.IPAddress(ipaddress.IPv4Address(u"127.0.0.1")),
        ]),
        critical=False,
    ).sign(private_key, hashes.SHA256(), default_backend())

    # Sauvegarde du certificat et de la clé
    # ...
```

Configuration Sécurisée

```
def _setup_ssl(self):
    """Configuration du contexte SSL/TLS."""
    context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
    context.load_cert_chain(certfile=cert_file, keyfile=key_file)

    # Paramètres de sécurité
    context.minimum_version = ssl.TLSVersion.TLSv1_2
    context.set_ciphers(
        'ECDHE+AESGCM:ECDHE+CHACHA20:DHE+AESGCM:DHE+CHACHA20:'
```

```
            '!aNULL:!MD5:!DSS'
        )

        return context
```

Cette configuration :

- Force TLS 1.2 minimum (exclut les versions vulnérables)
- Privilégie les suites de chiffrement modernes avec AEAD
- Exclut les algorithmes faibles (NULL, MD5, DSS)
- Préfère ECDHE/DHE pour la sécurité prospective au niveau du transport

---

Fonctionnalités de Messagerie

Modes de Communication

Le système supporte trois modes de communication :

Mode Diffusion (Broadcast) :

```
# Envoi à tous les utilisateurs en ligne
targets = [n for n in self.peer_keys.keys() if n != self.username]
self.encrypt_and_send_message(message_bytes, targets)
```

Mode Salon Privé :

```
# Communication exclusive avec un utilisateur
if self.current_room:
    targets = [self.current_room]
    self.encrypt_and_send_message(message_bytes, targets)
```

Message Privé Ponctuel :

```
# Commande /msg <utilisateur> <message>
target = parts[1]
content = parts[2]
self.encrypt_and_send_message(content.encode('utf-8'), [target])
```

Système d'Invitation aux Salons

L'entrée dans un salon privé nécessite l'acceptation des deux parties, implémentée via un système d'invitation :

```
# Émetteur : envoi de l'invitation
send_json(self.client_socket, {
    'type': 'room_invite',
    'target': target_username
})

# Serveur : acheminement avec ID unique
invite_id = self.generate_session_token()
self.pending_room_invites[invite_id] = {
    'from': sender,
    'to': target,
    'timestamp': time.time()
}

# Récepteur : acceptation ou refus
send_json(self.client_socket, {
    'type': 'room_invite_response',
    'invite_id': invite_id,
    'accepted': True  # ou False
})
```

Ce mécanisme empêche les communications non sollicitées et donne le contrôle aux utilisateurs.

Interface et Expérience Utilisateur

L'application utilise une interface en ligne de commande enrichie avec `prompt_toolkit`, offrant :

- Coloration syntaxique des messages selon leur type
- Gestion de l'historique avec `deque` (100 derniers messages)
- Indicateurs visuels pour les mentions (@utilisateur)
- Mode de saisie non-bloquant avec `patch_stdout()`
- Bannières contextuelles selon le mode actif

```
# Affichage d'un message avec horodatage et coloration
timestamp = datetime.now().strftime("%H:%M")
if is_private:
    msg = f"{GRAY}[{timestamp}] {GREEN}[{sender}]-(priv){RESET} {content}"
else:
    msg = f"{GRAY}[{timestamp}] {GREEN}[{sender}]{RESET} {content}"

self.display_message(msg, 'incoming')
```

Analyse de Sécurité

Menaces Couvertes

Menace	Protection Implémentée
Écoute passive du réseau	TLS 1.2+ + E2EE (AES-256)
Modification de messages	AEAD avec tags d'authentification
Compromission du serveur	E2EE (serveur ne voit que du chiffré)
Attaques par force brute	PBKDF2 (100k iter.) + verrouillage de compte
Attaques temporelles	Comparaisons en temps constant
Rejeu d'authentification	Nonces uniques avec expiration
Énumération d'utilisateurs	Temps de réponse constant

Limitations Connues

Absence de Perfect Forward Secrecy (PFS) :

Le système utilise des clés RSA statiques pour l'échange de clés. Bien que chaque message utilise une clé AES unique, la compromission d'une clé privée RSA permettrait le déchiffrement rétrospectif de tous les messages interceptés.

```
Scénario d'attaque :
1. Attaquant capture le trafic chiffré pendant des mois
2. Attaquant compromet la clé privée RSA d'un utilisateur
3. Attaquant peut déchiffrer toutes les clés AES des messages capturés
4. Accès à l'historique complet des communications
```

Pour implémenter un vrai PFS, il faudrait :

- Échange de clés Diffie-Hellman éphémères par session
- Suppression immédiate des clés privées éphémères après usage
- Rotation périodique des clés de session (comme le Double Ratchet de Signal)

Autres limitations :

- Certificats auto-signés (pas de PKI établie)
- Pas de persistance des messages côté serveur
- Pas de support multi-appareil pour un même utilisateur
- Vulnérable aux compromissions des endpoints clients

Technologies Utilisées

Bibliothèques Cryptographiques :

- `pycryptodome` : RSA, AES, PBKDF2, hachage
- `cryptography` : Génération de certificats X.509, SSL/TLS
- Modules standard : `ssl`, `hmac`, `hashlib`, `secrets`

Interface et Réseau :

- `prompt_toolkit` : Interface terminal avancée
- `socket` : Communication TCP/IP
- `threading` : Gestion multi-clients côté serveur

Stockage et Sérialisation :

- JSON pour le protocole applicatif
- Base64 pour l'encodage des données binaires
- Fichiers locaux pour la persistance des clés

Compétences Démontrées

Ce projet illustre une maîtrise des concepts suivants :

Cryptographie Appliquée :

- Chiffrement hybride (asymétrique + symétrique)
- Modes de chiffrement authentifiés (AEAD)
- Dérivation de clés (KDF)
- Authentification par challenge-response
- Gestion du cycle de vie des clés

Sécurité Applicative :

- Authentification robuste sans transmission de secrets
- Protection contre les attaques courantes (brute-force, timing, rejeu)
- Architecture zero-trust (serveur non privilégié)
- Journalisation de sécurité

Développement Système :

- Architecture client-serveur multi-threadée
- Protocoles réseau personnalisés
- Gestion des sockets TCP/IP avec TLS
- Traitement asynchrone des événements

Bonnes Pratiques :

- Code modulaire et maintenable
- Gestion d'erreurs robuste
- Documentation technique claire
- Logging de sécurité détaillé

Conclusion

pyMessenger représente une implémentation complète d'un système de messagerie sécurisée, démontrant une compréhension approfondie des principes cryptographiques modernes et de leur mise en œuvre

pratique. Le projet met l'accent sur la sécurité dès la conception (security by design) tout en maintenant une architecture claire et extensible.

Les choix techniques reflètent un équilibre entre sécurité maximale et contraintes d'implémentation, avec une transparence totale sur les limitations et les compromis effectués. Cette approche pragmatique de la cryptographie appliquée constitue une base solide pour des systèmes de communication sécurisés plus ambitieux.

Technologies : Python 3, Cryptographie (RSA, AES, PBKDF2, HMAC), SSL/TLS, Sockets, Threading
Lignes de code : ~2000
Durée du projet : Développement personnel
Disponibilité : Code source disponible sur demande