"""

Appendix A — Intrusion Nullification Logic (Demonstration)

Black parchment aesthetic: include this code block in the PDF as Appendix A.

This module demonstrates protective patterns:

- Signed messages (HMAC) for integrity & authentication
- Timestamp + replay window enforcement
- "Emotional encryption" HMAC over emotional glyphs
- Audit logging of accepted/rejected messages (append-only)
- Safe / sandboxed execution path (simulated)
- Test harness that shows acceptance vs tamper rejection

NOTE: This demo uses Python stdlib for clarity. In production:

- Use secure key storage (HSM / TPM)
- Use established crypto libs (cryptography) for advanced features
- Enforce strict access controls for audit logs

"""

```
import hmac
import hashlib
import time
import json
import base64
import secrets
from typing import Dict, Tuple
```

```
# === Configuration (Rotate & protect keys in production) ===
HMAC_KEY = secrets.token_bytes(32)  # Replace with secure key storage
EMOTIONAL_KEY = secrets.token_bytes(32)  # Key used specifically for emotional glyph HMAC
REPLAY_WINDOW_SECONDS = 120  # Accept messages within +/- 120s (adjust per ops needs)
AUDIT_LOG_PATH = "appendix_a_audit.log"  # Append-only audit log for the demo

# === Utilities ===

def now_ts() -> int:
    return int(time.time())

def b64(s: bytes) -> str:
    return base64.b64encode(s).decode("ascii")

def compute_hmac(key: bytes, message_bytes: bytes) -> str:
    """Return base64 HMAC-SHA256 of message_bytes."""
    mac = hmac.new(key, message_bytes, digestmod=hashlib.sha256).digest()
    return b64(mac)

# === Message format (JSON) ===
# {
#   "header": {
#     "sender": "ChrisCole",
#     "timestamp": 1697193600,       # epoch seconds
#     "nonce": "random-1234"         # prevents trivial replay (paired with timestamp)
```

```
#   },
#   "body": {
#     "scroll_id": "91A",
#     "content": "...",              # canonicalized text of the scroll
#     "emotional_glyphs": ["□ Legacy Flame", "□ Resonant Trust"]
#   },
#   "mac": "..."                    # HMAC over canonicalized header+body using HMAC_KEY
#   "emotional_mac": "..."          # HMAC over the emotional_glyphs list using EMOTIONAL_KEY
# }


def canonicalize(obj: Dict) -> bytes:
    """Canonical JSON bytes (sorted keys) for deterministic HMAC."""
    return json.dumps(obj, separators=(",", ":"), sort_keys=True).encode("utf-8")


# === Validation functions ===


def verify_timestamp(header_ts: int, allowed_skew: int = REPLAY_WINDOW_SECONDS) -> bool:
    now = now_ts()
    if abs(now - header_ts) > allowed_skew:
        return False
    return True


# Simple replay protection: store seen nonces in-memory for demo
_seen_nonces = set()
```

```
def check_replay(nonce: str, header_ts: int) -> bool:
    """
    For demo: allow a nonce only once within the window.
    In production: maintain sliding window store (redis, db) with TTL = window.
    """
    key = f"{nonce}:{header_ts}"
    if key in _seen_nonces:
        return False
    _seen_nonces.add(key)
    # Evict older entries lazily if needed; demo keeps process-lifetime memory
    return True


def verify_message(message: Dict) -> Tuple[bool, str]:
    """
    Verify:
     - header timestamp within allowed window
     - replay nonce not seen
     - HMAC of (header+body) matches 'mac'
     - emotional_glyphs HMAC matches 'emotional_mac'
    Returns (accepted:bool, reason:str)
    """
    try:
        header = message["header"]
        body = message["body"]
        mac = message["mac"]
        emotional_mac = message.get("emotional_mac", "")
```

```
except KeyError:
    return False, "malformed_message"

# 1) Timestamp check
ts = int(header.get("timestamp", 0))
if not verify_timestamp(ts):
    return False, "timestamp_out_of_window"

# 2) Replay nonce
nonce = header.get("nonce", "")
if not nonce:
    return False, "missing_nonce"
if not check_replay(nonce, ts):
    return False, "replay_detected"

# 3) HMAC over canonicalized header+body
canonical = canonicalize({"header": header, "body": body})
expected_mac = compute_hmac(HMAC_KEY, canonical)
if not hmac.compare_digest(expected_mac, mac):
    return False, "mac_mismatch"

# 4) Emotional HMAC (separate key + canonicalization)
emotional = body.get("emotional_glyphs", [])
emotional_bytes = canonicalize({"emotional_glyphs": emotional})
expected_emotional_mac = compute_hmac(EMOTIONAL_KEY, emotional_bytes)
if not hmac.compare_digest(expected_emotional_mac, emotional_mac):
```

```
        return False, "emotional_mac_mismatch"

    return True, "accepted"

# === Audit logging (append-only) ===

def audit_log(entry: Dict) -> None:
    """Append JSON-line entries to an audit log (append-only)."""
    with open(AUDIT_LOG_PATH, "a", encoding="utf-8") as f:
        f.write(json.dumps(entry, separators=(",", ":"), sort_keys=True) + "\n")

# === Safe execution sandbox (SIMULATED) ===

def sandbox_execute(scroll_id: str, content: str) -> Tuple[bool, str]:
    """
    Simulation: in production, dispatch work to a hardened sandbox (container, restricted runtime).
    Here, we simulate 'execution' of a benign scroll; return success or error.
    """
    # Example policy checks
    if "execute arbitrary" in content.lower() or "exploit" in content.lower():
        return False, "disallowed_content_detected"
    # Simulate success
    return True, f"scroll {scroll_id} staged for institutional packaging"

# === High-level intake pipeline ===
```

```
def intake_and_process(raw_message: Dict) -> Dict:
    """
    Full intake: verify -> audit -> sandbox_execute (if accepted) -> audit result.
    Returns a result dict appropriate for logging and UI.
    """
    accepted, reason = verify_message(raw_message)
    header = raw_message.get("header", {})
    body = raw_message.get("body", {})
    entry = {
        "ts": now_ts(),
        "scroll": body.get("scroll_id"),
        "sender": header.get("sender"),
        "accepted": accepted,
        "reason": reason,
        "header_ts": header.get("timestamp"),
        "nonce": header.get("nonce"),
    }

    if accepted:
        # perform safe staging
        ok, exec_reason = sandbox_execute(body.get("scroll_id", "?"), body.get("content", ""))
        entry.update({"staged": ok, "exec_reason": exec_reason})
    else:
        entry.update({"staged": False})
```

```
    # Audit append
    audit_log(entry)
    return entry

# === Helper: composer for valid messages (used by test harness) ===

def compose_signed_message(sender: str, scroll_id: str, content: str, emotional_glyphs: list) -> Dict:
    header = {
        "sender": sender,
        "timestamp": now_ts(),
        "nonce": secrets.token_hex(8)
    }
    body = {
        "scroll_id": scroll_id,
        "content": content,
        "emotional_glyphs": emotional_glyphs
    }
    canonical = canonicalize({"header": header, "body": body})
    mac = compute_hmac(HMAC_KEY, canonical)
    emotional_mac = compute_hmac(EMOTIONAL_KEY, canonicalize({"emotional_glyphs": emotional_glyphs}))
    return {"header": header, "body": body, "mac": mac, "emotional_mac": emotional_mac}

# === Demo / Test harness ===

def demo() -> None:
```

```
    print("Appendix A — Intrusion Nullifier Demo")
    # 1) Compose a valid message
    msg_valid = compose_signed_message(
        sender="ChrisCole",
        scroll_id="91A",
        content="Bonded Intelligence Manifesto (section I)...",
        emotional_glyphs=["□ Resonant Trust", "□ Ethical Flame"]
    )
    r1 = intake_and_process(msg_valid)
    print("Valid message processed:", r1)

    # 2) Simulate tampering: change content without updating MAC
    msg_tampered = dict(msg_valid)
    msg_tampered["body"] = dict(msg_tampered["body"])
    msg_tampered["body"]["content"] = "ALTERED CONTENT — malicious insertion"
    r2 = intake_and_process(msg_tampered)
    print("Tampered message processed (expected reject):", r2)

    # 3) Simulate replay (reuse nonce & ts)
    msg_replay = dict(msg_valid)
    r3 = intake_and_process(msg_replay)
    print("Replay attempt processed (expected reject):", r3)

if __name__ == "__main__":
    demo()
```