AI Honeypot - Judge Q&A; Guide

Comprehensive Questions & Answers (Easy → Hard)

■ EASY	■ MEDIUM	■ HARD
Basic concepts	Technical details	Deep code-level

1. GENERAL & CONCEPT QUESTIONS

■ What is a honeypot?

A honeypot is a decoy system designed to attract and detect attackers. It mimics a real application but logs all interactions for analysis. Our AI honeypot goes beyond traditional honeypots by using machine learning to predict attacker behavior.

■ What makes your honeypot 'AI-powered'?

We use Markov chain models to predict the attacker's next move based on their attack sequence. For example, after detecting SQL injection, our system predicts a 60% probability of admin access attempts next. Traditional honeypots just log attacks - we predict what's coming.

■ What's your OWASP coverage?

We detect 7 out of 10 OWASP Top 10 vulnerabilities (70% coverage): SQL Injection, XSS, Path Traversal, Command Injection, SSRF, Authentication Bypass, and Insecure Deserialization. We focus on attack-oriented categories that honeypots actually encounter.

■ How does your system differ from traditional honeypots?

Traditional honeypots: Log attacks, generate alerts. **Our honeypot:** (1) Predicts next attack using ML, (2) Maps to MITRE ATT&CK; automatically, (3) Generates incident response playbooks, (4) Adapts responses based on attacker skill level, (5) Exports threat intelligence in STIX 2.1 format. We're proactive, not just reactive.

■ What is MITRE ATT&CK; and why do you use it?

MITRE ATT&CK; is the industry-standard framework for categorizing adversary tactics and techniques. We automatically map detected attacks to MITRE techniques (e.g., SQL Injection → T1190: Exploit Public-Facing Application). This helps security teams understand the attack in context and compare with known APT groups. For example, our system might detect that an attack pattern matches APT28 tactics.

■ Explain your threat level calculation algorithm

We use a scoring system (0-10+ points) based on four factors:

```
Stage Score (1-5): RECONNAISSANCE=1, EXPLOITATION=3, PRIVILEGE_ESCALATION=4, DATA_EXFILTRATION=5
Goal Score (+0-2): DATA_THEFT or SYSTEM_COMPROMISE = +2 points
Skill Score (+0-1): advanced/automated attackers = +1 point
Time Score (+0-2): <10 min to compromise = +2, <30 min = +1

Classification: score ≥7 = CRITICAL, ≥5 = HIGH, ≥3 = MEDIUM, <3 = LOW
```

This multi-factor approach ensures threat levels reflect both attack sophistication and urgency.

2. MACHINE LEARNING & PREDICTION

How do you predict the next attack?

We use Markov chains to track attack sequences. When an attacker performs SQL injection followed by admin access, we learn that transition. Next time we see SQL injection, we can predict admin access is likely next with a probability score.

What is a Markov chain and why use it?

A Markov chain is a statistical model where the next state depends only on the current state. Perfect for attack prediction because attackers follow patterns: reconnaissance → exploitation → privilege escalation. We build a transition probability matrix: if current_state = 'SQL Injection', we track probabilities for next_state = {'admin_access': 0.6, 'command_execution': 0.3, ...}. This is computationally efficient and learns from every attack.

How do you handle cold start (new attackers)?

We initialize the Markov chain with common attack patterns from security research: SQL Injection → admin_access, XSS → session_hijacking, etc. This gives us baseline predictions even for first-time attackers. As we observe more attacks, the model adapts and learns attacker-specific patterns.

Walk me through the prediction code

File: attack_predictor.py, **Class:** MarkovChainPredictor

```
def learn_transition(current_state, next_state):
    transition_counts[current_state][next_state] += 1
    total = sum(transition_counts[current_state].values())
    for next_s in transition_counts[current_state]:
        transitions[current_state][next_s] = count / total

def predict_next(current_state, top_k=3):
    next_states = transitions[current_state]
    return sorted(next_states.items(), key=lambda x: x[1], reverse=True)[:top_k]
```

We maintain two dictionaries: transition_counts (raw counts) and transitions (probabilities). Every attack updates counts and recalculates probabilities. Prediction returns top-k most likely next states.

How do you estimate time to compromise?

We use a heuristic model based on attack stage, skill level, and attack speed:

```
base_time = stage_times[current_stage] # e.g., EXPLOITATION = 15 min
multiplier = skill_multipliers[skill_level] # novice=2.0, advanced=0.5
if attack_speed > 10: multiplier *= 0.7 # Fast attacks
return int(base_time * multiplier)
```

This gives realistic estimates: a novice at reconnaissance stage might take 60 minutes, while an advanced attacker at privilege escalation could compromise in 5 minutes.

3. ATTACK DETECTION & ANALYSIS

■ How do you detect SQL injection?

We use pattern matching with 13+ signatures: ' OR 1=1, UNION SELECT, DROP TABLE, admin'--, etc. When a request payload contains any of these patterns, we classify it as SQL Injection. The detection happens in analyzer.py using the AttackAnalyzer class.

■ What attack types can you detect?

7 types: (1) SQL Injection, (2) XSS, (3) Path Traversal, (4) Command Injection, (5) SSRF, (6) Authentication Bypass, (7) Insecure Deserialization. Each has 8-14 detection signatures covering common attack patterns.

■ How do you avoid false positives?

We use multiple strategies: (1) Pattern specificity - signatures are precise (e.g., 'http://localhost' for SSRF, not just 'localhost'), (2) Case-insensitive matching for flexibility, (3) Ordered pattern checking (specific patterns first), (4) Context awareness - we check the full payload, not just fragments. For production, we'd add ML-based anomaly detection as a second layer.

■ Explain your behavioral analysis

We profile attackers across multiple dimensions: (1) **Skill level:** NOVICE (basic attacks), INTERMEDIATE (multiple vectors), ADVANCED (sophisticated techniques), AUTOMATED (high-speed scanning). (2) **Tool detection:** Identify sqlmap, Burp Suite, Metasploit from user-agent and attack patterns. (3) **Attack velocity:** Track requests per minute. (4) **Persistence:** Count repeated attacks on same endpoint. This builds a comprehensive attacker profile for threat assessment.

■ Show me the attack detection code

File: analyzer.py, **Lines:** 120-138

```
def analyze(self, payload: str) -> AttackType:
    if not payload: return AttackType.NORMAL
    for pattern in self.patterns:
        if pattern.matches(payload):
            return pattern.attack_type
    return AttackType.NORMAL

class AttackPattern:
    def matches(self, payload: str) -> bool:
        search_payload = payload.lower()
        return any(sig in search_payload for sig in self.signatures)
```

We iterate through patterns in order, checking if any signature matches the payload. First match wins, so we order patterns from most specific to least specific.

■ How do you classify attack stages?

We use indicator-based classification on the last 5 actions:

```
stage_indicators = {
    RECONNAISSANCE: ['normal', 'scan', 'probe'],
    INITIAL_ACCESS: ['SQL Injection', 'XSS'],
    PRIVILEGE_ESCALATION: ['admin_access', 'sudo', 'root'],
    DATA_EXFILTRATION: ['credential', 'dump', 'export']
}

for action in recent_actions[-5:]:
    for stage, indicators in stage_indicators.items():
        if any(ind in action.lower() for ind in indicators):
            stage_scores[stage] += 1
return max(stage_scores, key=stage_scores.get)
```

4. ARCHITECTURE & IMPLEMENTATION

■ What tech stack did you use?

Backend: Python 3.11 with FastAPI for async APIs. Frontend: HTML/CSS/JavaScript with WebSockets for real-time updates. Data: JSON file storage (attacks.json). Deployment: Docker for containerization. Libraries: ReportLab (PDF generation), Jinja2 (templating).

■ How many API endpoints do you have?

16 total: 13 GET endpoints (prediction, MITRE, timeline, playbooks, exports, etc.), 1 POST endpoint (fingerprinting), and 2 WebSocket endpoints (dashboard, demo). All 10 core endpoints have been tested with 100% success rate.

■ Explain your real-time dashboard architecture

We use WebSockets for bidirectional real-time communication. When an attack is detected in app.py, we call broadcast_demo_update() which sends JSON data to all connected clients via WebSocket. The frontend JavaScript listens for messages and updates the DOM instantly (attack counter, threat level, timeline). This gives sub-second latency for dashboard updates - much faster than polling.

■ How do you handle concurrent attacks?

FastAPI is async by default, so we handle concurrent requests efficiently. Each attack is logged with a unique attacker_id (cookie-based). We maintain separate attack sequences per attacker in memory (dict keyed by attacker_id). File writes to attacks.json are synchronous but fast (append-only). For production scale, we'd use a database with proper indexing and connection pooling.

■ Walk me through the request flow

Step-by-step for /search?q=' OR 1=1--:

```
1. FastAPI receives request at /search endpoint
2. Get/create attacker_id from cookie
3. analyzer.analyze_request(payload) → returns 'SQL Injection'
4. attack_predictor.track_attack(attacker_id, 'SQL Injection', '/search')
5. threat_intel.analyze_threat(attacker_id, ip, user_agent, 'SQL Injection')
6. logger.log_attack(attacker_id, 'SQL Injection', payload, ip, ...)
7. broadcast_demo_update(attack_data) → WebSocket to all clients
8. deception_engine.create_response() → Generate fake results
9. Return JSONResponse with fake data + set attacker_id cookie
```

■ How do you generate STIX bundles?

File: threat_sharing.py, **Function:** generate_stix_bundle()

```
bundle = {
    'type': 'bundle', 'id': f'bundle--{uuid4()}',
    'objects': [
        {'type': 'indicator', 'pattern': f"[ipv4-addr:value = '{ip}']"},
        {'type': 'attack-pattern', 'name': attack_type},
        {'type': 'relationship', 'source_ref': indicator_id, 'target_ref': pattern_id}
    ]
}
```

We create STIX 2.1 compliant JSON with indicators (IPs, patterns), attack-patterns (mapped to MITRE), and relationships linking them. This can be imported into SIEM or shared with threat intelligence platforms.

5. SECURITY & PRODUCTION READINESS

■ Is this production-ready?

Yes, for small-to-medium deployments. We have: (1) Docker deployment, (2) Comprehensive logging, (3) Error handling, (4) 100% API test success, (5) SIEM integration via STIX export. For enterprise scale, we'd add: database backend, rate limiting, distributed deployment, and monitoring.

■ How do you prevent the honeypot from being detected?

We use adaptive deception: (1) **Realistic responses:** Generate fake but plausible data (SQL results, file listings). (2) **Timing delays:** Add realistic latency to avoid instant responses. (3) **Error messages:** Return authentic-looking errors. (4) **Skill-based adaptation:** Novice attackers get more 'helpful' errors, advanced attackers get subtle clues. (5) **Canary tokens:** Embed unique tokens to track data exfiltration. The goal is to be indistinguishable from a real vulnerable app.

■ What about data privacy and logging?

We log: attacker IP, user-agent, attack payloads, timestamps. We DON'T log: real user data (it's a honeypot). All logs are stored locally in attacks.json. For GDPR compliance in production: (1) Anonymize IPs after 30 days, (2) Provide data deletion API, (3) Add consent banners (though attackers won't see them), (4) Encrypt logs at rest. Since this is a honeypot (no legitimate users), privacy concerns are minimal.

■ How would you scale this to handle 10,000 requests/sec?

Architecture changes needed:

```
1. Database: Migrate from JSON to PostgreSQL/TimescaleDB
   - Indexed queries on attacker_id, timestamp, attack_type
   - Connection pooling (pgbouncer)

2. Caching: Redis for hot data (recent attacks, predictions)
   - Cache prediction results (TTL: 5 min)
   - Cache attacker profiles

3. Load Balancing: Multiple FastAPI instances behind nginx
   - Horizontal scaling with Docker Swarm/Kubernetes
   - Sticky sessions for WebSocket connections

4. Async Processing: Celery for heavy tasks
   - MITRE mapping, playbook generation → background jobs
   - Message queue (RabbitMQ/Redis)

5. Monitoring: Prometheus + Grafana
   - Track request latency, error rates, attack patterns
```

■ What security vulnerabilities exist in your honeypot?

Honest assessment: (1) **DoS vulnerability:** No rate limiting - attacker could flood with requests. Fix: Add rate limiting per IP (10 req/sec). (2) **File system DoS:** Unlimited log growth in attacks.json. Fix: Log rotation, max file size. (3) **Cookie manipulation:** Attacker could forge attacker_id cookie. Fix: Sign cookies with HMAC. (4) **WebSocket flooding:** No connection limits. Fix: Max connections per IP. (5) **LLM injection:** If LLM is enabled, malicious prompts could cause issues. Fix: Input sanitization, output validation. These are acceptable for a honeypot (we WANT attacks), but would be critical in production.

6. BUSINESS VALUE & REAL-WORLD IMPACT

■ Who would use this?

Security teams at companies with web applications: (1) SOC analysts for threat detection, (2) Incident responders for attack analysis, (3) Threat intelligence teams for IOC sharing, (4) Security researchers for attack pattern study. Also useful for: Managed security service providers (MSSPs), penetration testers, cybersecurity training.

■ What problem does this solve?

Traditional honeypots are passive - they log attacks but don't help you respond. Security teams waste hours: (1) Manually analyzing attack patterns, (2) Looking up MITRE techniques, (3) Writing incident response procedures, (4) Correlating attacks across systems. Our honeypot automates all of this: predict next attack, map to MITRE, generate playbooks, export threat intel. This saves 5-10 hours per incident and enables proactive defense.

■ How would you monetize this?

Business model options:

```
1. SaaS Subscription ($99-999/month):
   - Starter: 1 honeypot, 10K attacks/month, basic analytics
   - Pro: 5 honeypots, 100K attacks/month, ML predictions, MITRE mapping
   - Enterprise: Unlimited, custom ML models, SIEM integration, API access

2. Managed Service ($2K-10K/month):
   - We deploy and manage honeypots in customer infrastructure
   - 24/7 monitoring, threat intelligence reports, incident response support

3. Threat Intelligence Feed ($500-5K/month):
   - Sell aggregated attack data, IOCs, STIX bundles to other orgs
   - API access to our global honeypot network

4. Enterprise Licensing ($50K-500K/year):
   - On-premise deployment, white-label, custom integrations
   - Training, support, professional services
```

■ What's your competitive advantage?

vs. Traditional Honeypots (Cowrie, Dionaea): We add ML prediction and auto-response. **vs. Commercial Solutions (Illusive Networks, Attivo):** We're open-source and customizable. **vs. SIEM (Splunk, QRadar):** We're specialized for deception, not general log analysis. **Unique value:** Only solution that combines ML prediction + MITRE mapping + auto-playbooks in one package. Our moat: Proprietary ML models trained on real attack data, integration ecosystem, ease of deployment.

7. TRICKY & GOTCHA QUESTIONS

■ Why not just use a WAF instead of a honeypot?

WAFs and honeypots serve different purposes. **WAF:** Blocks attacks on production systems. **Honeypot:** Attracts and studies attacks in a safe environment. You need both: WAF for protection, honeypot for intelligence. Our honeypot gives you: (1) Early warning of new attack techniques, (2) Attacker behavior insights, (3) Threat intelligence for WAF rule tuning, (4) Incident response practice. Think of it as a 'burglar alarm' vs. 'security camera' - you want both.

■ How do you know your ML predictions are accurate?

Great question. We measure accuracy by: (1) **Prediction hit rate:** Did the predicted attack actually occur? (Track over time). (2) **Probability calibration:** If we say 60% probability, does it happen ~60% of the time? (3) **Baseline comparison:** Compare against random guessing and simple heuristics. Currently, we're in 'learning mode' - the more attacks we see, the better predictions get. For production, we'd need: (1) Holdout test set, (2) A/B testing, (3) Continuous retraining, (4) Human-in-the-loop validation.

■ What if an attacker realizes it's a honeypot?

That's actually okay - we still learned from their behavior. But to minimize detection: (1) **Realistic responses:** Use real database schemas, authentic error messages. (2) **Timing variation:** Add jitter to response times. (3) **Behavioral adaptation:** Don't be 'too vulnerable' - make them work for it. (4) **Mixed deployment:** Deploy alongside real apps so attackers can't tell which is which. (5) **Canary analysis:** If they exfiltrate data with our canary tokens, we track where it goes. Even if detected, we've already captured their techniques, tools, and infrastructure.

■ Your code has no unit tests. How do you ensure quality?

Fair criticism. For this hackathon, we prioritized features over tests. We did: (1) Manual testing of all 10 API endpoints (100% pass rate), (2) Integration testing with real attacks, (3) Code review for critical paths. For production, we'd add: (1) **Unit tests:** pytest for each module (target: 80% coverage), (2) **Integration tests:** Test full attack flows end-to-end, (3) **Property-based testing:** Use Hypothesis for edge cases, (4) **CI/CD:** GitHub Actions to run tests on every commit, (5) **Load testing:** Locust to test under high load. Testing is critical for production - we acknowledge this gap.

■ How is this different from just logging to a SIEM?

SIEMs are reactive - they alert after attacks happen. We're proactive: (1) **Prediction:** We tell you what's coming next, not just what happened. (2) **Context:** We map to MITRE and compare with APT groups automatically. (3) **Response:** We generate playbooks instantly, not just alerts. (4) **Deception:** We actively engage attackers to learn more. (5) **Integration:** We FEED your SIEM with enriched data (STIX export). Think of us as a specialized threat intelligence source for your SIEM, not a replacement.

Remember: Be honest about limitations, show deep understanding, and emphasize learning mindset. Judges value authenticity over perfection!