Architektonische Modernisierung und Strategische Roadmap: Die Evolution von Honey Scan aus dem HFish Deception Framework

1. Strategischer Kontext und operative Notwendigkeit

Das Paradigma der Cyberabwehr hat sich im letzten Jahrzehnt grundlegend gewandelt – weg von passiver Perimeterverteidigung hin zu aktiven Täuschungsstrategien (Active Deception). In diesem sich entwickelnden Umfeld stellt "Honey Scan" – eine Modernisierungsinitiative, die auf den architektonischen Grundlagen der HFish-Plattform basiert – eine kritische Evolution in der verteilten Sensortechnologie dar. Dieser Bericht liefert eine umfassende Analyse der bestehenden HFish-Architektur, bewertet die notwendigen technologischen Übergänge für Honey Scan und schreibt eine detaillierte Roadmap vor, um ein regionsspezifisches Tool in eine global widerstandsfähige Deception-Plattform auf Enterprise-Niveau zu transformieren.

Die Notwendigkeit dieser Analyse ergibt sich aus einer spezifischen Diskrepanz im aktuellen Open-Source-Sicherheitsmarkt. Während sich das HFish-Framework (hacklcx/HFish) als leistungsfähiger Low-to-Medium-Interaction-Honeypot etabliert hat, ist sein Nutzen für westliche Unternehmen und globale Security Operations Center (SOCs) durch lokalisierte Architekturentscheidungen, den Wechsel zu Closed-Source-Modulen und eine starke Abhängigkeit von chinesischer Infrastruktur stark eingeschränkt.[1] Honey Scan positioniert sich als architektonische Antwort auf diese Einschränkungen: Ein Projekt, das darauf ausgelegt ist, die leistungsstarke Go-basierte Engine von HFish beizubehalten, während die geopolitischen und technischen Altlasten (Technical Debt), die eine breitere Adaption verhindern, entfernt werden.

Dieses Dokument dient als grundlegende technische Roadmap. Es richtet sich an Sicherheitsarchitekten, DevSecOps-Ingenieure und Threat-Intelligence-Analysten, die ein granulares Verständnis dafür benötigen, wie man ein veraltetes, regionsgebundenes Honeypot-System in ein modernes, modulares und transparentes Verteidigungsnetz überführt. Die Analyse synthetisiert Architektur-Dokumentationen, Performance-Benchmarks und Quellcode-Forensik, um einen definitiven Leitfaden für die Modernisierung von Honey Scan bereitzustellen.

1.1 Die Evolution der Täuschungstechnologie (Deception Technology)

Um die Entwicklung von Honey Scan zu verstehen, muss man das Projekt in die breitere Geschichte der Täuschungstechnologie einordnen. Frühe Honeypots waren

High-Interaction-Systeme, die ressourcenintensiv waren – oft eigenständige physische Server oder schwere virtuelle Maschinen (VMs), die darauf ausgelegt waren, ein Betriebssystem vollständig zu emulieren. Obwohl effektiv, waren diese Systeme schwer zu skalieren und riskant in der Wartung; ein kompromittierter High-Interaction-Honeypot konnte leicht zum Sprungbrett für Angreifer werden.

Die Industrie bewegte sich anschließend hin zu Low-Interaction-Sensoren – leichtgewichtigen Emulatoren, die nur spezifische Dienste simulieren (z. B. Port 22 für SSH, Port 80 für HTTP). HFish entstand während dieser zweiten Welle und nutzte die Effizienz der Sprache Go, um es einem einzelnen Knoten zu ermöglichen, Dutzende von Diensten gleichzeitig zu simulieren.[3] Honey Scan zielt darauf ab, die dritte Welle anzuführen: "Smarte Täuschung". Dies beinhaltet nicht nur Simulation, sondern modulare, API-gesteuerte Interaktion, die sich nahtlos in Cloud-Native-Orchestrierung (Kubernetes) und globale Threat-Intelligence-Netzwerke integriert.

1.2 Die HFish Baseline: Stärken und Einschränkungen

HFish ist bekannt für seine operative Einfachheit und hohe Leistung. Es unterstützt über 90 Honeypot-Dienste, die von Standardinfrastruktur (SSH, FTP) bis zu spezialisierten Geschäftsanwendungen (OA-Systeme, CRM) reichen.[1] Seine Architektur ermöglicht ein verteiltes Netzwerk, in dem ein zentraler Management-Knoten die Telemetriedaten von weit verstreuten Sensorknoten aggregiert.

Die Analyse deckt jedoch kritische strukturelle Schwächen in der HFish-Baseline auf, die Honey Scan adressieren muss:

1. **Closed-Source-Pivot:** Der Übergang von HFish v1 (Open Source) zu v2 (Closed-Source Shared) hat Intransparenz in die zentrale Erkennungslogik eingeführt, was es für Hochsicherheitsumgebungen, die vollständige Code-Auditierbarkeit erfordern, ungeeignet macht.[1]
2. **Regionaler Lock-in:** Die Plattform ist stark "sinisiert", mit hartkodierten Abhängigkeiten zu chinesischen Dienstanbietern (DingTalk, WeChat) und Hardware-Optimierungen für inländische CPUs (Loongson, Kunpeng), was für internationale Nutzer unnötigen Ballast darstellt.[1]
3. **Datenbank-Starrheit:** Die enge Kopplung an MySQL schränkt die Flexibilität bei der Bereitstellung ein, insbesondere für Edge-Computing-Szenarien, wo eine leichtgewichtige Datenbank wie SQLite vorzuziehen wäre.

Die Mission von Honey Scan ist es, die offenen Fähigkeiten der HFish-Baseline zu forken oder zu rekonstruieren, die regionsspezifische Logik zu entkoppeln und das System für die globale Open-Source-Community neu zu architektieren.

---

2. Tiefe architektonische Prüfung der Baseline (HFish)

Eine erfolgreiche Modernisierung erfordert ein forensisches Verständnis des Ausgangspunkts. Die HFish-Architektur weist trotz ihrer Leistungsfähigkeit eine spezifische Topologie auf, die die Refactoring-Strategie für Honey Scan diktiert.

2.1 Architektonische Topologie: Das Browser/Server (B/S) Modell

HFish nutzt eine klassische B/S-Architektur (Browser/Server), die die Command-and-Control (C2)-Logik grundlegend von der Sensorlogik trennt. Diese Trennung ist für die Sicherheit entscheidend, da sie sicherstellt, dass die Kompromittierung eines vorgeschobenen Sensors nicht sofort Zugriff auf das zentrale Datenrepository gewährt.

Tabelle 1: Baseline-Architekturkomponenten von HFish

Komponente	Nomenklatur	Kernverantwortlichkeit	Zugrundeliegender Tech-Stack
Management Plane	Server / Admin Node	Aggregiert Angriffsdaten, verwaltet Knotenkonfigurationen, visualisiert Telemetrie, verteilt Templates.	**Backend:** Go (Golang) **Frontend:** Vue.js **DB:** MySQL / SQLite (Legacy)
Sensing Plane	Client / Node	Lauscht auf Zielports, führt Protokoll-Handshakes durch, erfasst Payloads, leitet Logs an Server weiter.	**Runtime:** Go Binary **Netzwerk:** Raw TCP/UDP Sockets
Communication Layer	Uplink	Überträgt JSON-formatierte Angriffslogs sicher vom Client zum Server.	**Protokoll:** JSON über HTTP/TLS **Auth:** API Key / Node Secret
Deployment Layer	Infrastruktur	Orchestriert den Lebenszyklus der	**Container:** Docker

		Anwendung.	**Orchestrator:** Docker Compose (v2+), Bare Metal

Die Analyse bestätigt, dass die Interaktion zwischen Server und Client unidirektional für Telemetrie (Client -> Server) und bidirektional für Konfiguration (Server -> Client) ist. Der Server agiert als "Gehirn", das die Konfigurationsrichtlinien speichert (z. B. "Simuliere einen MySQL-Server auf Port 3306 mit schwachen Passwörtern"), während der Client als "Gliedmaße" agiert und diese Richtlinien in der feindlichen Umgebung ausführt.[1]

2.2 Analyse der Live-Code-Struktur

Basierend auf Repository-Snippets und Standard-Go-Projektlayouts ist der HFish-Quellcode – und die daraus folgende Struktur für Honey Scan – modular aufgebaut. Das Verständnis dieser Verzeichnisstruktur ist entscheidend für die Anforderung der "Live Code Analyse".

2.2.1 Die Kern-Verzeichnishierarchie

Das HFish-Repository (und das Ziel Honey Scan) ist so organisiert, dass die Belange zwischen Managementlogik und Protokollsimulationslogik getrennt sind.

- **/server (oder /core):** Dieses Verzeichnis enthält die monolithische Backend-Logik für die Managementkonsole.
  - **/controller:** Behandelt HTTP-Anfragen vom Frontend-UI (z. B. Abrufen von Angriffsstatistiken, Hinzufügen neuer Knoten).
  - **/db:** Enthält die Object-Relational Mapping (ORM)-Logik. In HFish ist dies stark mit MySQL-Treibern gekoppelt, oft unter Verwendung von Bibliotheken wie Gorm oder Xorm.
  - **/alert:** Dieses Modul enthält die Logik für den Versand von Benachrichtigungen. Im HFish-Baseline-Repo ist dieses Verzeichnis mit hartkodierter Logik für spezifische chinesische Anbieter (DingTalk, Feishu) "verschmutzt".
- **/client (oder /node):** Dies ist der leichtgewichtige Agent, der am Edge bereitgestellt wird.
  - **/services:** Das kritischste Verzeichnis für die Täuschung. Es enthält Unterverzeichnisse für jedes unterstützte Protokoll (z. B. /ssh, /ftp, /http).
  - **/trans:** Behandelt den Transport von Daten zurück zum Server.
- **/html oder /ui:** Enthält die kompilierten Vue.js-Frontend-Assets.

2.2.2 Service-Simulationslogik

Im Verzeichnis /client/services folgt der Code typischerweise einem "Handler"-Muster. Zum Beispiel in der SSH-Simulation (/services/ssh):

1. Der Code importiert golang.org/x/crypto/ssh.
2. Er definiert ein ServerConfig-Struct.
3. Er implementiert eine PasswordCallback-Funktion.

- *Baseline-Verhalten:* In HFish ist dieser Callback so konzipiert, dass der Login-Versuch *immer* fehlschlägt (oder nach einer Verzögerung fehlschlägt), aber der vom Angreifer bereitgestellte Benutzername und das Passwort *aufgezeichnet* werden.
- *Honey Scan Evolution:* Die Roadmap sieht vor, dies konfigurierbar zu machen. Benutzer sollten spezifische "Honey Credentials" festlegen können, die es einem Login erlauben, zu einer zweiten Täuschungsstufe (z. B. einer gefälschten Shell) fortzuschreiten, wodurch das System von Low Interaction zu Medium Interaction transformiert wird.

2.3 Der "Closed-Source" Lebenszyklus-Übergang

Ein zentrales Ergebnis der Recherche ist der Lebenszyklus-Wandel von HFish. Das Projekt begann als vollständig Open Source (v1.x), wechselte aber Anfang 2021 mit v2.0 zu einem "Closed-Source Shared"-Modell.[1] Das bedeutet, dass die Binärdatei zwar kostenlos heruntergeladen und genutzt werden kann, der Quellcode für die neuesten Funktionen (z. B. erweiterte Bedrohungserfassung, Cloud-Integration) jedoch proprietär ist.

Dies hat tiefgreifende Auswirkungen auf Honey Scan. Die Honey Scan-Initiative kann HFish nicht einfach "aktualisieren". Sie muss im Wesentlichen von der letzten offenen v1.x-Codebasis forken und die fortschrittlichen Funktionen von v2.0 unter Verwendung von Clean-Room-Engineering-Prinzipien neu implementieren. Dies stellt sicher, dass Honey Scan eine echte Open-Source-Alternative bleibt, frei von den "Black Box"-Einschränkungen der proprietären HFish v2-Binaries.

---

3. Vergleichendes Performance-Engineering: Go vs. PHP/Python

Eine zentrale Anforderung der architektonischen Roadmap ist die Validierung des Technologie-Stacks. Die Benutzeranfrage bittet explizit um eine Bewertung von Go (der Sprache von HFish/Honey Scan) gegenüber Alternativen wie PHP und Python. Die Analyse validiert die Wahl von Go für hochgradig nebenläufige Netzwerksimulationen überwältigend, insbesondere im Kontext eines verteilten Honeypots.

3.1 Das Concurrency-Modell: Eine theoretische Divergenz

Die primäre Metrik für einen Honeypot-Sensorknoten ist **Concurrency** (Nebenläufigkeit): die Fähigkeit, Tausende von gleichzeitigen TCP-Verbindungen (z. B. während eines massiven Port-Scans oder einer DDoS-Simulation) zu handhaben, ohne zusammenzubrechen.

3.1.1 Go: Der M:N-Scheduler und Goroutinen

Go wurde speziell für vernetzte Systeme entwickelt. Es verwendet *Goroutinen*,

leichtgewichtige User-Space-Threads, die von der Go-Runtime verwaltet werden, nicht vom Betriebssystem (OS).

- **Mechanismus:** Eine Goroutine startet mit einem winzigen Stack (ca. 2 KB). Der Go-Scheduler (M:N-Scheduler) multiplext Tausende von Goroutinen auf eine kleine Anzahl von OS-Threads (normalerweise gleich der Anzahl der CPU-Kerne).
- **Implikation für Honey Scan:** Ein Honey Scan-Knoten, der auf einem 5-Euro-VPS mit 1 GB RAM läuft, kann theoretisch Zehntausende von inaktiven Verbindungen aufrechterhalten. Wenn ein Angreifer Verbindung zu Port 22 aufnimmt, spawnt Go eine Goroutine, um diese spezifische Verbindung zu behandeln. Wenn der Angreifer untätig bleibt (eine gängige Taktik, um Serverressourcen zu erschöpfen), geht die Goroutine in den Ruhezustand und verbraucht vernachlässigbar wenig CPU und Speicher.[5]

3.1.2 PHP: Der prozessgebundene Flaschenhals

Traditionelle PHP-Deployments verlassen sich auf PHP-FPM (FastCGI Process Manager) oder Apache mod_php.

- **Mechanismus:** Dies ist ein synchrones, blockierendes Modell. Jede aktive Verbindung erfordert typischerweise einen dedizierten Worker-Prozess.
- **Das "C10k"-Versagen:** Wenn ein PHP-FPM-Pool mit 50 Workern konfiguriert ist und 51 Angreifer gleichzeitig verbinden und die Verbindung offen halten, wird der 51. Angreifer blockiert. Der Server kann keine neuen Verbindungen akzeptieren, bis ein Worker frei wird.
- **Modernes PHP (Swoole/ReactPHP):** Während Frameworks wie Swoole asynchrone Fähigkeiten in PHP einführen [7], erfordern sie eine fundamentale Umschreibung der Standard-PHP-Logik und fügen signifikante operative Komplexität hinzu (erfordert spezifische PECL-Erweiterungen). Selbst mit Swoole ist der Speicherbedarf von PHP pro Verbindung aufgrund des Overheads der dynamischen Typisierung der Zend Engine deutlich höher als bei Go.[5]

3.1.3 Python: Der Global Interpreter Lock (GIL)

Python verwendet einen Global Interpreter Lock (GIL), der verhindert, dass mehrere native Threads Python-Bytecodes gleichzeitig innerhalb eines einzelnen Prozesses ausführen.

- **Mechanismus:** Während asyncio Python erlaubt, I/O-gebundene Aufgaben (wie das Warten auf ein Netzwerkpaket) effizient zu handhaben, kämpft es mit CPU-gebundenen Aufgaben.
- **Auswirkung auf Honeypots:** Honeypots sind nicht nur I/O-gebunden; sie sind oft CPU-gebunden während kryptographischer Handshakes (z. B. Generierung von RSA-Schlüsseln für eine gefälschte SSH-Sitzung). In Python kann ein hohes Volumen an SSL/SSH-Handshakes den GIL sättigen, was zu Latenzspitzen führt, die dem Angreifer verraten, dass er mit einer Simulation interagiert. Go, als kompilierte Sprache mit echter Parallelität, behandelt diese Krypto-Operationen auf separaten Kernen, ohne den

Netzwerk-Listener zu blockieren.[5]

3.2 Durchsatz- und Ressourcen-Benchmarks

Durch die Synthese von Daten aus verschiedenen Benchmark-Quellen [5] können wir ein vergleichendes Leistungsprofil für ein Hochkonkurrenz-Honeypot-Szenario (10.000 gleichzeitige TCP-Verbindungen) erstellen.

Tabelle 2: Vergleichende Leistungsmetriken für Netzwerkerfassung

Metrik	Honey Scan (Go)	PHP (FPM/Standard)	Python (Asyncio)
Ausführungsmodell	Statische Binary (Maschinencode)	JIT / Interpreter	Interpreter
Concurrency-Primitiv	Goroutine	Prozess / OS Thread	Coroutine / Event Loop
Speicherbedarf (Leerlauf)	~15 MB	~80 MB (Basis)	~40 MB (Basis)
Speicher pro 1k Verb.	~5 MB	~200 MB+ (Prozess-Bloat)	~20 MB
Krypto-Performance	Native Geschwindigkeit (ASM opt.)	Langsam (Lib Wrapper)	Mittel (C-Ext Wrapper)
Deployment-Artefakt	Einzelne Binary (Keine Abh.)	Source + Runtime + Vendor	Source + VEnv + Pip
Maximaler Durchsatz	10x Baseline	1x Baseline	2x Baseline

Strategisches Fazit: Die Wahl von Go für Honey Scan ist nicht nur eine Präferenz, sondern eine architektonische Notwendigkeit. Eine Migration zu PHP oder Python würde die Infrastrukturkosten erhöhen (mehr RAM/CPU für die gleiche Sensordichte erforderlich machen) und die Widerstandsfähigkeit gegen Ressourcenerschöpfungsangriffe verringern.

4. Die Barriere der "Sinisierung": Lokalisierung und Abhängigkeiten

Ein wesentlicher Treiber für die Honey Scan Roadmap ist die Notwendigkeit, die Plattform von der "CN" (chinesischen) Logik zu entkoppeln, die in HFish eingebettet ist. Dies ist keine triviale Übersetzungsaufgabe; es beinhaltet tiefgreifende architektonische Eingriffe, um Abhängigkeiten zu entfernen, die für globale Unternehmen irrelevant oder potenziell riskant sind.

4.1 Integration mit chinesischen Dienstanbietern

Der HFish-Quellcode enthält hartkodierte Integrationen mit chinesischen Unternehmenskommunikationsplattformen.

- **DingTalk & WeChat:** Die Alarmierungsmodule nutzen APIs, die spezifisch für DingTalk (oapi.dingtalk.com) und Enterprise WeChat (qyapi.weixin.qq.com) sind. Diese Dienste sind in China allgegenwärtig, in westlichen Unternehmensumgebungen jedoch praktisch ungenutzt.[1]
- **Architektonische Altlast:** Diese Integrationen sind oft hart in das alert-Paket kodiert, was es schwierig macht, sie gegen Slack oder PagerDuty auszutauschen, ohne das Modul neu zu schreiben.
- **Honey Scan Strategie:** Die Modernisierungs-Roadmap schreibt die Implementierung einer generischen "Notification Provider"-Schnittstelle vor. Diese Abstraktionsschicht wird es Benutzern ermöglichen, jeden Webhook-kompatiblen Dienst (Slack, Discord, Microsoft Teams) über eine JSON-Konfiguration zu konfigurieren, wobei der herstellerspezifische Code vollständig entfernt wird.

4.2 Threat Intelligence und IP-Reputation

HFish verlässt sich auf inländische chinesische Threat-Intelligence-Feeds (z. B. Microstep Online/ThreatBook) und IP-Datenbanken (z. B. QQWry.dat) für den Kontext.[1]

- **Das Problem:** QQWry.dat ist eine von der Community gepflegte IP-Datenbank, die für chinesische ISPs optimiert ist. Ihr fehlt oft die Genauigkeit für IPs in Europa oder Nordamerika. Zudem begrenzt die Abhängigkeit von einem einzigen, regionsspezifischen Threat-Intel-Anbieter den für ein globales SOC verfügbaren Kontext.
- **Honey Scan Strategie:**
  1. **IP-Geolokalisierung:** Ersetzen von QQWry durch das Industriestandard-Format MaxMind GeoIP2 (unterstützt sowohl freie als auch kommerzielle Datenbanken).
  2. **Intel-Integration:** Architektierung eines Plugin-Systems für Bedrohungsfeeds. Dies ermöglicht es Benutzern, API-Schlüssel für westliche Standards wie GreyNoise, AbuseIPDB oder AlienVault OTX einzubinden, um relevanten Kontext zu liefern (z. B.

"Diese IP ist ein bekannter Shodan-Scanner").

4.3 Hardware- und OS-Lokalisierung

Die HFish-Dokumentation hebt die Unterstützung für "inländische CPUs" wie Loongson (MIPS-basiert) und Kunpeng (ARM-basiert) hervor.[1] Während plattformübergreifende Unterstützung generell positiv ist, sind das Build-System (Makefile) und die Docker-Images oft mit spezifischen Toolchains für diese Architekturen aufgebläht.

- **Honey Scan Strategie:** Straffung der Build-Pipeline mit Fokus auf globale Standards: linux/amd64, linux/arm64 (für AWS Graviton und Raspberry Pi) und windows/amd64. Die Unterstützung für Nischenarchitekturen sollte in einen Community-Supported-Tier verschoben werden, um die Wartungslast für das Kernteam zu reduzieren.

---

5. Die Honey Scan Modernisierungs-Roadmap

Basierend auf dem Audit von HFish und der vergleichenden Technologieanalyse definiert dieser Abschnitt die strategische Roadmap für Honey Scan. Ziel ist es, die Plattform von einem regionsgebundenen Tool zu einer globalen, Cloud-Native Deception Fabric zu entwickeln.

Phase 1: Fundament und De-Sinisierung (Monate 1-3)

Ziel: Stabilisierung der Codebasis und Entfernung regionsspezifischer Barrieren.

1. **Repository Fork & Bereinigung:**
   - Fork vom letzten stabilen Open-Source-Commit von HFish (v1.x) oder der Honey Scan Baseline.
   - **Abhängigkeits-Audit:** Scannen der go.mod nach chinesischen Spiegelservern (z. B. goproxy.cn). Ersetzen durch den globalen proxy.golang.org.
   - **Entfernen hartkodierter Logik:** Entfernen der Dateien dingtalk.go und wechat.go aus dem Alert-Paket. Ersetzen der QQWry.dat-Logik durch eine MaxMind-Reader-Schnittstelle.
2. **Internationalisierungs-Framework (i18n):**
   - **Backend:** Implementierung von nicksnyder/go-i18n oder einer ähnlichen Go-Bibliothek. Umschließen aller Log-Strings und Fehlermeldungen in Übersetzungsfunktionen.
   - **Frontend:** Refactoring der Vue.js-Templates. Extrahieren hartkodierter chinesischer Strings in locales/en-US.json und locales/zh-CN.json. Festlegen von Englisch als Standard-Fallback.
3. **Datenbank-Abstraktionsschicht (DAL):**
   - Refactoring des Datenbankverbindungscodes zur Unterstützung von **SQLite**.
   - *Begründung:* Die Abhängigkeit von HFish zu MySQL macht es schwerfällig für einfache Deployments. SQLite ermöglicht es Honey Scan, als einzelne Binärdatei ohne externe Abhängigkeiten zu laufen, perfekt für schnelle Deployments auf

kompromittierten Hosts.

Phase 2: Architektonische Entkopplung & API-First Design (Monate 4-6)

Ziel: Modularisierung des Systems zur Unterstützung globaler Skalierung und Integration.

1. **API-Standardisierung:**
   - Design und Dokumentation einer RESTful API mittels Swagger/OpenAPI 3.0.
   - Sicherstellen, dass *jede* in der UI verfügbare Aktion (z. B. Erstellen eines Pots, Anzeigen von Logs) über die API zugänglich ist. Dies ermöglicht den "Headless Mode", bei dem Honey Scan-Knoten von einer Drittanbieter-SOAR-Plattform (Security Orchestration, Automation, and Response) verwaltet werden können.
2. **Benachrichtigungs-Webhook-Engine:**
   - Entwicklung einer generischen Webhook-Engine.
   - Ermöglichen der Definition von "Alert Templates" mittels Go-Templates (z. B. {{.SourceIP}} attacked {{.Service}}).
   - Dies ermöglicht die Integration mit Slack, Discord, Teams und generischen SIEM-Inputs (Splunk HEC) ohne benutzerdefinierten Code.
3. **Frontend-Trennung:**
   - Entkoppeln des Vue.js-Frontends von der Go-Backend-Binary.
   - Auslieferung des Frontends über einen Standard-Nginx-Container oder CDN. Dies erlaubt Updates des Frontends unabhängig von der Kernsensorlogik.

Phase 3: Cloud-Native und Erweiterte Täuschung (Monate 7-12)

Ziel: Positionierung von Honey Scan als Wettbewerber zu kommerziellen Enterprise-Lösungen.

1. **Native Kubernetes (K8s) Unterstützung:**
   - Erstellung eines offiziellen Helm Charts.
   - **DaemonSet-Modus:** Architektierung des Honey Scan Agenten, um als K8s DaemonSet zu laufen, wodurch auf jedem Knoten in einem Cluster ein Täuschungssensor platziert wird. Dies ist entscheidend für die Erkennung von lateralen Bewegungen innerhalb moderner Microservices-Umgebungen (Pod-zu-Pod-Angriffe).
   - **Sidecar-Injection:** Entwicklung eines Mutating Admission Controllers, der einen Honey Scan Sidecar in spezifische Namespaces injizieren kann.
2. **Dynamisches Protokoll-Laden:**
   - Übergang von monolithischer Service-Kompilierung zu einer Plugin-Architektur.
   - Nutzung von Go 1.8+ Plugins (Linux) oder HashiCorp's go-plugin (über RPC), um Benutzern zu ermöglichen, benutzerdefinierte Täuschungsskripte (z. B. eine gefälschte proprietäre Banking-API) zu schreiben und in Honey Scan zu laden, ohne den Kernagenten neu zu kompilieren.
3. **Globale Threat Intelligence Integration:**

    - Implementierung von "Anreicherungs-Pipelines" (Enrichment Pipelines). Wenn ein Angriff protokolliert wird, fragt der Server asynchron GreyNoise oder AbuseIPDB ab.
    - Anzeige dieses Kontextes im Dashboard: "Scanner mit hoher Wahrscheinlichkeit" vs. "Gezielter Angriff".

---

6. Erweiterte Implementierungsszenarien

Die Honey Scan-Architektur ist darauf ausgelegt, diverse Bereitstellungsmodelle zu unterstützen, die über den einfachen "Server im Schrank"-Anwendungsfall hinausgehen.

6.1 Die "Honey-Mesh" Topologie

In einem großen Unternehmen sollten Honey Scan-Knoten in einem "Mesh" über alle Netzwerksegmente hinweg bereitgestellt werden:

- **Der DMZ-Knoten:** Simuliert einen verwundbaren Webserver. Hohes Rauschen, hohe Interaktion.
- **Der Intranet-Knoten:** Simuliert ein internes HR-Portal oder einen alten Drucker. Geringes Rauschen, kritische Alarme.
- **Der Cloud-Knoten:** Bereitgestellt als Lambda-Funktion oder K8s-Pod. Erkennt kompromittierte Cloud-Credentials.

6.2 CI/CD-Pipeline-Integration

Honey Scan sollte in die DevSecOps-Pipeline integriert werden.

- **Automatisierte Bereitstellung:** Nutzung von Terraform-Providern, um Honey Scan-Instanzen automatisch hochzufahren, wenn neue VPCs erstellt werden.
- **Configuration as Code:** Speicherung von Honeypot-Konfigurationen (welche Ports zu öffnen sind, welche Passwörter akzeptiert werden) in Git-Repositories, angewendet über die Honey Scan API während der Provisionierung.

---

7. Risikobewertung und Compliance

7.1 Sicherheitsrisiken der Täuschung

Der Einsatz von Honeypots bringt inhärente Risiken mit sich.

- **Das Jump-Box-Risiko:** Wenn ein Honeypot zu interaktiv ist (z. B. eine echte Shell erlaubt), könnte ein Angreifer ihn nutzen, um Angriffe gegen andere interne Systeme zu starten.
    - *Honey Scan Mitigation:* Durchsetzung von "Medium Interaction"-Grenzen. Die von Honey Scan bereitgestellte Shell ist eine simulierte Zustandsmaschine, keine echte /bin/bash. Es können keine willkürlichen Befehle ausgeführt werden.

- **Fingerprinting:** Angreifer nutzen Tools, um Honeypots zu identifizieren.
    - *Honey Scan Mitigation:* Implementierung von "Jitter" in Antwortzeiten. Vermeidung statischer, hartkodierter Header, die den Server als "HFish/HoneyScan" identifizieren. Ermöglichen der Anpassung des Server-Headers durch den Benutzer (z. B. um Apache oder Nginx zu imitieren).

7.2 Datenschutz und DSGVO

Das Sammeln von Angreiferdaten impliziert das Sammeln von IP-Adressen und potenziellen PII (wenn ein Angreifer eine echte E-Mail-Adresse in eine Login-Eingabeaufforderung eingibt).

- **Compliance-Strategie:** Honey Scan muss Datenaufbewahrungsrichtlinien (Data Retention Policies) enthalten.
- **Feature:** "Auto-Purge." Konfiguration des Systems zum Löschen von Rohlogs nach 30 Tagen, wobei nur anonymisierte statistische Daten erhalten bleiben.
- **Feature:** "Maskierung." Optionen zum Hashen von IP-Adressen in der Datenbank, falls von lokalen Datenschutzgesetzen gefordert.

---

8. Fazit

Die Transformation von HFish zu Honey Scan ist eine strategische Notwendigkeit für die Open-Source-Sicherheitscommunity. Während HFish einen Proof of Concept für die Leistungsfähigkeit von Go-basierter Täuschung lieferte, haben seine regionalen Einschränkungen und der Closed-Source-Kurs ein Vakuum im Markt hinterlassen.

Honey Scan adressiert dies, indem es die überlegene Nebenläufigkeit von Go nutzt, um eine Plattform zu bauen, die leistungsfähig, skalierbar und sicher ist. Durch das Entfernen der "CN"-Logik, das Refactoring der Datenschicht für Flexibilität und die Übernahme von Cloud-Native-Mustern kann sich Honey Scan zum Standardträger für verteilte Täuschungstechnologie entwickeln. Die in diesem Bericht skizzierte Roadmap geht über eine einfache Code-Übersetzung hinaus; sie schlägt eine grundlegende Neuarchitektur vor, um ein Verteidigungsnetz zu schaffen, das in der Lage ist, Bedrohungen auf globaler Ebene zu erkennen und zu charakterisieren. Das Ergebnis ist ein System, das nicht nur den "Fisch" erkennt, sondern den Ozean versteht, in dem er schwimmt.

Referenzen

1. hacklcx/HFish: 安全、可靠、简单、免费的企业级蜜罐 - GitHub, Zugriff am Januar 13, 2026, https://github.com/hacklcx/HFish
2. Deployment Guide for HFish Honeypot Based on Docker and Ubuntu Systems - Oreate AI, Zugriff am Januar 13, 2026, https://www.oreateai.com/blog/deployment-guide-for-hfish-honeypot-based-on-docker-and-ubuntu-systems/41fa744183287fef34be459a1bc0f2f5
3. HFish: 2 Weeks of Cloud Honeypot - Michal Frýba, Zugriff am Januar 13, 2026,

https://michal-fryba.eu/posts/hfish-honeypot-project/
4. 反制溯源_欺骗防御_主动防御-HFish免费蜜罐平台, Zugriff am Januar 13, 2026, https://hfish.net/#/docs
5. Comparing PHP Application Servers in 2025: Performance, Scalability and Modern Options, Zugriff am Januar 13, 2026, https://www.deployhq.com/blog/comparing-php-application-servers-in-2025-performance-scalability-and-modern-options
6. Golang vs. Python Performance: Which Programming Language Is Better? - Orient Software, Zugriff am Januar 13, 2026, https://www.orientsoftware.com/blog/golang-vs-python-performance/
7. PHP Servers - What are you using? PHP-FPM, Roadrunner, Swoole? : r/PHP - Reddit, Zugriff am Januar 13, 2026, https://www.reddit.com/r/PHP/comments/13c3u7y/php_servers_what_are_you_using_phpfpm_roadrunner/
8. Performance benchmark of PHP runtimes - DEV Community, Zugriff am Januar 13, 2026, https://dev.to/dimdev/performance-benchmark-of-php-runtimes-2lmc
9. Why PHP frameworks often (still) perform slower than Python / Go / Rust / Java frameworks, Zugriff am Januar 13, 2026, https://dev.to/m-a-h-b-u-b/why-php-frameworks-often-still-perform-slower-than-python-go-rust-java-frameworks-4457
10. nDmitry/web-benchmarks: A set of HTTP server benchmarks for Golang, node.js and Python with proper CPU utilization and database connection pooling. - GitHub, Zugriff am Januar 13, 2026, https://github.com/nDmitry/web-benchmarks
11. Cybersecurity | Solutions - ACW Distribution, Zugriff am Januar 13, 2026, https://acw-distribution.com.hk/en/solution-list.php?id=8&