Structured Project Guideline for Building a DNS Service with Domain Filtering and eBPF Integration

- DNS translates domain names into IP addresses via various record types (A, AAAA, MX, CNAME, etc.).
- Domain filtering blocks malicious domains at the DNS level, preventing access to harmful content and reducing cyberattack risks.
- eBPF enables kernel-level packet filtering and interception, significantly improving DNS filtering performance and security.
- Rust is recommended for performance and memory safety, Go for ease of use and concurrency, and C for low-level control but with higher risk.
- A modular design with clear milestones, practical experiments, and visualizations will facilitate learning and implementation.

Introduction

Building a DNS service with domain filtering capabilities and integrating eBPF for performance and security enhancements is a challenging yet rewarding project. This guideline aims to break down the complex concepts into digestible, hands-on milestones that build upon one another, forming a structured learning journey. The project will deepen your understanding of DNS protocols, network security, and systems programming while allowing you to choose between Rust, Go, or C based on performance, safety, and ease of implementation.

The project is designed to be modular, with each component (DNS resolution, domain filtering, eBPF integration) developed and tested independently before integration. Practical experiments, mini-projects, and challenges are included to reinforce concepts learned, and interactive or visual elements (e.g., logging DNS queries in real-time, simulating attacks, benchmarking performance) are suggested to make the learning experience dynamic and engaging.

High-Level Overview

Purpose

The purpose of this project is to create a custom DNS service capable of resolving domain names into IP addresses while filtering domains based on security policies. The integration of eBPF will enhance performance by filtering packets at the kernel level and provide advanced security capabilities such as intercepting malicious DNS queries before they reach user-space.

Scope

- Implement a basic DNS resolver supporting common record types (A, AAAA, MX, CNAME).
- Add domain blacklisting/whitelisting functionality with configurable policies.
- Integrate eBPF for kernel-level packet filtering and performance monitoring.
- Develop modular components for DNS parsing, filtering logic, eBPF hooks, and logging.
- Benchmark and stress-test the service to ensure performance and reliability.

Key Technologies

- **DNS Protocol**: Understanding DNS records, query types, and resolution mechanisms.
- **Domain Filtering**: Techniques for blocking malicious domains via blacklists, whitelists, and regex matching.
- **eBPF/XDP**: Kernel-level packet filtering and interception for performance and security enhancements.
- **Programming Languages**: Rust (recommended for performance and safety), Go (for ease of use and concurrency), or C (for low-level control).

Detailed Project Roadmap

Phase 1: DNS Fundamentals and Basic Resolver Implementation

Learning Goals

- Understand DNS protocol basics, including record types (A, AAAA, MX, CNAME), DNS resolution process, and DNS packet structure.
- Learn how to parse and construct DNS packets.
- Implement a minimal DNS resolver in the chosen language (Rust/Go/C).

Milestones

1. Write a program that listens for DNS queries on UDP port 53.
2. Parse incoming DNS packets to extract domain names and query types.
3. Construct and send DNS responses for A and AAAA queries with hardcoded IP addresses.
4. Test the resolver using `dig` and `nslookup` to verify correct responses.

Resources and Tools

- DNS RFCs (RFC 1034, RFC 1035) for protocol details.
- Libraries:
    - Rust: `trust-dns`, `tokio` for async I/O, `bitvec` for bit manipulation.
    - Go: `miekg/dns` package.
    - C: `libpcap` for packet capture, manual parsing of DNS packets.

- Tutorials:
    - Rust: Writing a toy DNS Server in Rust, Building a DNS Server in Rust.
    - Go: Implementing a DNS server in Go.
    - C: Building a custom simple DNS server in C.

Challenges or Experiments

- Use `dig` and `nslookup` to query your DNS server and verify responses.
- Implement decompression of DNS packets to handle compressed domain names.
- Forward DNS queries to a public resolver (e.g., Google DNS) and cache responses.

Success Criteria

- The resolver correctly answers A and AAAA queries for test domains.
- The resolver can parse and respond to basic DNS queries within 100ms.

Stretch Goals

- Support additional DNS record types (MX, CNAME).
- Implement iterative DNS resolution by querying authoritative name servers.

Phase 2: Domain Filtering for Security

Learning Goals

- Understand domain filtering techniques: blacklisting, whitelisting, regex matching.
- Learn how to integrate threat intelligence feeds for dynamic filtering.
- Implement domain filtering logic in the DNS service.

Milestones

1. Add a domain blacklist/whitelist configuration file or database.
2. Modify the DNS resolver to check incoming queries against the filter lists.
3. Log blocked queries and generate alerts for suspicious activity.
4. Simulate DNS-based attacks (e.g., cache poisoning) to test filtering rules.

Resources and Tools

- DNS filtering tools: `dnsdist`, `Pi-hole`, `SafeDNS` for inspiration.
- Threat intelligence APIs (e.g., VirusTotal, AlienVault OTX).
- Logging libraries:
    - Rust: `log`, `env_logger`.
    - Go: `log` package.
    - C: `syslog`.

Challenges or Experiments

- Simulate a DNS cache poisoning attack and verify that your filter blocks malicious responses.
- Benchmark filtering performance with large blacklists (10,000+ domains).
- Implement rate limiting to prevent DoS attacks.

Success Criteria

- The DNS service blocks queries to blacklisted domains and logs them.
- The service remains responsive under load (e.g., 1000 queries/sec).

Stretch Goals

- Integrate AI-driven threat detection for real-time domain filtering.
- Deploy the service in a containerized environment (Docker, Kubernetes).

Phase 3: eBPF/XDP Integration for Performance and Security

Learning Goals

- Understand eBPF/XDP basics: how it intercepts packets at the kernel level.
- Learn how to use eBPF to filter DNS traffic based on source IP and domain names.
- Integrate eBPF with your DNS service for kernel-level filtering.

Milestones

1. Write an eBPF program to intercept DNS packets and filter them based on source IP and domain names.
2. Integrate the eBPF program with your DNS service, attaching it to the network interface.
3. Benchmark performance improvements compared to userspace filtering.
4. Implement dynamic eBPF rules that can be updated at runtime.

Resources and Tools

- eBPF documentation: Linux kernel eBPF docs, Brendan Gregg's eBPF guide.
- Libraries:
    - Rust: `aya-rs`, `redbpf`.
    - Go: `cilium/ebpf`.
    - C: `libbpf`, `libxdp`.
- Tutorials:
    - Writing eBPF/XDP load-balancer in Rust.
    - A Complete Guide to eBPF with Rust.
    - dnsdist eBPF documentation.

Challenges or Experiments

- Compare userspace vs. kernel-space filtering latency and throughput.
- Use `bpftrace` or `BCC` to debug eBPF programs.
- Implement dynamic eBPF rules that expire after a set time.

Success Criteria

- eBPF filter drops 100% of blacklisted domains at the kernel level.
- Kernel-space filtering reduces latency by at least 50% compared to userspace.

Stretch Goals

- Use XDP (eXpress Data Path) for even faster packet processing.
- Integrate eBPF with Kubernetes for cloud-native DNS filtering.

Phase 4: Benchmarking, Stress-Testing, and Deployment

Learning Goals

- Learn how to benchmark DNS service performance.
- Understand stress-testing techniques to evaluate resilience.
- Deploy the DNS service locally and route traffic to it.

Milestones

1. Benchmark the DNS service using tools like `wrk` or `dhcperf`.
2. Stress-test the service with high query loads (e.g., 10,000 queries/sec).
3. Deploy the service locally using `systemd` or Docker.
4. Route DNS traffic to your service and verify it handles real-world queries.

Resources and Tools

- Benchmarking tools: `wrk`, `dhcperf`, ab.
- Monitoring tools: Prometheus, Grafana for visualizing metrics.
- Deployment: `systemd`, Docker, Kubernetes.

Challenges or Experiments

- Simulate a DDoS attack and verify rate limiting and eBPF filtering mitigate it.
- Use Wireshark to capture and analyze DNS traffic.
- Deploy the service in a cloud environment (AWS, GCP).

Success Criteria

- The service handles 10,000 queries/sec with < 100ms latency.

- The service remains stable under attack simulations.

Stretch Goals

- Implement DNS-over-HTTPS (DoH) or DNS-over-TLS (DoT) for encrypted queries.
- Add a web UI for managing filters and monitoring queries.

Modular Design and Directory Structure

The project should be structured modularly to facilitate independent development and testing:

```
project-root/
├── dns-resolver/          # Core DNS query handling
│   ├── src/
│   ├── tests/
│   └── README.md
├── domain-filter/         # Domain blacklist/whitelist logic
│   ├── src/
│   ├── tests/
│   └── README.md
├── ebpf-filter/           # eBPF/XDP packet filtering
│   ├── src/
│   ├── tests/
│   └── README.md
├── logging-monitoring/    # Logging and metrics
│   ├── src/
│   ├── tests/
│   └── README.md
├── config-management/     # Configuration and runtime management
│   ├── src/
│   ├── tests/
│   └── README.md
├── docs/                  # Project documentation
│   └── project-documentation.md
└── README.md
```

Each module should have clear interfaces (e.g., Rust traits, Go interfaces) to allow swapping implementations and facilitate testing.

Programming Language Recommendations

Language	Pros	Cons	Suitable For
Rust	Memory safety, performance, zero-cost abstractions, strong	Steeper learning curve, strict compiler	High-performance, secure DNS service with eBPF integration

Language	Pros	Cons	Suitable For
	concurrency model		
Go	Simplicity, ease of use, built-in concurrency, rich standard library	Garbage collection overhead, less low-level control	Rapid development, scalable DNS service with good performance
C	Low-level control, performance	No memory safety, complex manual memory management	Performance-critical applications, but not recommended for DNS service due to complexity

Recommendation: Use **Rust** if you prioritize performance and safety, especially for eBPF integration. Use **Go** if you prefer ease of use and rapid development with good performance. Avoid **C** unless you have specific low-level requirements and are experienced with manual memory management.

Timeline Estimate

Assuming part-time commitment (10-15 hours/week):

Phase	Estimated Duration	Description
1	2-3 weeks	Learn DNS basics, implement minimal resolver
2	2-3 weeks	Add domain filtering, integrate threat feeds
3	3-4 weeks	Learn eBPF, implement kernel-level filtering
4	2-3 weeks	Benchmarking, stress-testing, deployment

Troubleshooting Section

Common Pitfalls

- **DNS Timeouts**: Check socket bindings, firewall rules, and DNS packet parsing logic.
- **eBPF Verifier Errors**: Ensure eBPF programs comply with kernel verifier rules; use `bpftrace` for debugging.

- **Memory Leaks**: Use language-specific tools (e.g., Rust's `valgrind`, Go's `pprof`) to detect leaks.
- **Performance Bottlenecks**: Profile code with `perf`, `flamegraph`, or language-specific profilers.

Debugging Tools

- `dig`, `nslookup`: Test DNS queries.
- `Wireshark`: Capture and analyze DNS traffic.
- `bpftrace`, `BCC`: Debug eBPF programs.
- `valgrind`, `pprof`: Memory and performance profiling.

Conclusion

This structured project guideline provides a comprehensive roadmap for building a DNS service with domain filtering and eBPF integration. By breaking down the project into modular phases with clear learning goals, milestones, and challenges, it facilitates a hands-on learning journey that deepens understanding of DNS protocols, network security, and systems programming. Choosing the right programming language (Rust, Go, or C) depends on your priorities for performance, safety, and ease of use. The project encourages experimentation and benchmarking to reinforce concepts and ensure a robust, secure, and performant DNS service.