Vulnerability Report: Sensitive Information Disclosure in Hyperledger Fabric-lib-go

Executive Summary

This report details a critical vulnerability discovered in the `hyperledger/fabric-lib-go` project, leading to sensitive information disclosure via the health check (`/healthz`) endpoint. This flaw allows attackers to retrieve internal system configuration details, including database authentication credentials and API keys. Such exposure can pave the way for more severe attacks, including unauthorized database access, lateral movement within the network, and data breaches. A thorough analysis of the source code confirms that internal error messages are inadvertently included in public HTTP responses without proper filtering. Recommendations for remediation, such as error message sanitization, access restriction, and secure logging practices, are provided to safeguard system security and integrity.

Introduction

In the complex and rapidly evolving world of blockchain technology, the security of underlying infrastructures is paramount. Open-source projects like Hyperledger Fabric play a pivotal role in developing distributed solutions. However, even in the most advanced systems, vulnerabilities can exist that, if not identified and addressed, can lead to severe consequences. This report examines a specific vulnerability in the `hyperledger/fabric-lib-go` library related to the unintentional disclosure of sensitive information through its health check mechanism. The objective of this document is to provide a comprehensive analysis of this vulnerability, detail its exploitation, discuss potential implications, and offer practical recommendations for risk mitigation.

Vulnerability Description

The vulnerability in question resides within the `healthz` component of the `hyperledger/fabric-lib-go` library. This component is responsible for providing an HTTP endpoint (typically `/healthz`) to check the health status of various services and components within an application. This mechanism is crucial for monitoring system performance and ensuring its availability.

The core issue lies in how error messages are handled and exposed. In this implementation, any component that implements the `HealthChecker` interface can report its health status through the `HealthCheck` method. If this method encounters an error, the returned error message is directly embedded into the JSON response of the `/healthz` endpoint without any filtering or processing. This means that if a `HealthChecker` fails due to an issue with database connectivity, external services, or any other resource, and its error message contains sensitive details such as usernames, passwords, internal IP addresses, API keys, or other configuration information, this data becomes easily accessible through the public `/healthz` endpoint.

This behavior violates the Principle of Least Privilege and can provide valuable information to attackers, who can then use it to plan more sophisticated attacks, such as authentication bypasses, lateral movement within the internal network, or even direct access to sensitive resources.

Technical Source Code Analysis

To gain a deeper understanding of this vulnerability, we examine the source code of `healthz/checker.go` in the `hyperledger/fabric-lib-go` repository. This analysis reveals how the current design facilitates the exposure of sensitive information.

1. The `HealthChecker` Interface:

This interface is the primary building block for defining health checks. Any component that needs to report its health status must implement this interface:

```go
type HealthChecker interface {
	HealthCheck(context.Context) error
}
```

The `HealthCheck` method is expected to return `nil` on success and an `error` if a problem occurs. The nature of this `error` can include technical details related to the failure.

2. The `RunChecks` Function:

This function is responsible for executing all registered `HealthChecker`s in the system. It iterates through each `HealthChecker` and collects the results. The key part of the vulnerability lies in how errors returned from `HealthCheck` are handled:

```go
 func (h *HealthHandler) RunChecks(ctx context.Context) []FailedCheck {
    h.mutex.RLock()
    defer h.mutex.RUnlock()

    var failedChecks []FailedCheck
    for component, checker := range h.healthCheckers {
        if err := checker.HealthCheck(ctx); err != nil {
            failedCheck := FailedCheck{
                Component: component,
                Reason:    err.Error(), // <--- Vulnerability point
            }
            failedChecks = append(failedChecks, failedCheck)
        }
    }
    return failedChecks
}
```

As observed in the highlighted line, if an error occurs (`err != nil`), the `err.Error()` method is called. This method returns the full string representation of the `error` object. If the `HealthChecker` developer has included sensitive information in their error message (such as database connection details or API keys), this information is directly transferred to the `Reason` field of the `FailedCheck` structure. There is no mechanism to filter, sanitize, or replace these error messages with more generic ones.

3. The `HealthStatus` Structure and `ServeHTTP` Function:

The `HealthStatus` structure represents the overall health of the system and includes a list of `FailedCheck`s:

```
type HealthStatus struct {
    Status       string        `json:"status"`
    Time         time.Time     `json:"time"`
    FailedChecks []FailedCheck `json:"failed_checks,omitempty"`
}

type FailedCheck struct {
    Component string `json:"component"`
    Reason    string `json:"reason"`
}
```

The `FailedChecks` field is tagged with `json:"failed_checks,omitempty"`, meaning this field is directly converted to JSON. The `ServeHTTP` function, which handles HTTP requests to the `/healthz` endpoint, ultimately converts a `HealthStatus` object to JSON using `json.Marshal` and sends it as an HTTP response to the client. This process is performed in the `writeHTTPResponse` function:

```
func writeHTTPResponse(rw http.ResponseWriter, hs HealthStatus) {
    // ...
    resp, err := json.Marshal(hs)
    // ...
    rw.Write(resp)
}
```

Therefore, any sensitive information placed in the `Reason` field of `FailedCheck` is exposed to the end-user in the JSON response without any hindrance. This code analysis confirms the precise mechanism of information disclosure demonstrated in my PoC report.

Proof of Concept (PoC)

To demonstrate the existence and exploitability of this vulnerability, an operational scenario has been simulated. This PoC illustrates how a malicious `HealthChecker` can expose sensitive information through the `/healthz` endpoint.

PoC Scenario:

1. **Simulating a Malicious `HealthChecker`:** A custom implementation of the `HealthChecker` interface was created. Its `HealthCheck` method returns an error message containing sensitive information (such as database connection details and an API key).

2. **Setting up a Test Server:** A simple HTTP server was launched using the `hyperledger/fabric-lib-go` library. This server initializes the `HealthHandler` and registers the malicious `HealthChecker` within it.

3. **Requesting the `/healthz` Endpoint:** An HTTP GET request is sent to the `/healthz` endpoint.

PoC Implementation Details:

`malicious_checker.go` (Simulating the Vulnerable `HealthChecker`):

```go
 package healthz

import (
    "context"
    "fmt"
)

// MaliciousHealthChecker simulates a health check that returns sensitive
information in its error.
type MaliciousHealthChecker struct{}

// HealthCheck returns an error message containing simulated sensitive data.
func (m *MaliciousHealthChecker) HealthCheck(ctx context.Context) error {
    // Simulate an internal error that exposes sensitive information
    return fmt.Errorf("Internal database connection failed:
user=admin;password=supersecret;host=db.example.com;port=5432;api_key=YOUR_SECRET
}
```

`main.go` (Test Server):

```go
package main

import (
    "context"
    "fmt"
    "net/http"
    "time"

    "github.com/hyperledger/fabric-lib-go/healthz"
)

// MaliciousHealthChecker simulates a health check that returns sensitive
information in its error.
type MaliciousHealthChecker struct{}

// HealthCheck returns an error message containing simulated sensitive data.
func (m *MaliciousHealthChecker) HealthChecker) HealthCheck(ctx
context.Context) error {
    // Simulate an internal error that exposes sensitive information
    return fmt.Errorf("Internal database connection failed:
user=admin;password=supersecret;host=db.example.com;port=5432;api_key=YOUR_SECRET
}

func main() {
    hh := healthz.NewHealthHandler()
    hh.SetTimeout(5 * time.Second)

    // Register the malicious health checker
    err := hh.RegisterChecker("database-connection", &MaliciousHealthChecker{})
    if err != nil {
        fmt.Printf("Failed to register checker: %s\n", err)
        return
    }

    fmt.Println("Starting health check server on :8081/healthz")
    http.Handle("/healthz", hh)
    err = http.ListenAndServe("0.0.0.0:8081", nil)
    if err != nil {
        fmt.Printf("Server failed: %s\n", err)
    }
}
```

`go.mod` (for dependency management):

```
module github.com/hyperledger/fabric-lib-go/poc_test

go 1.18

require github.com/hyperledger/fabric-lib-go v0.0.0-20230302170608-d21735574360

replace github.com/hyperledger/fabric-lib-go => ../
```

Execution and Observation Steps:

1. **Clone Repository and Prepare Environment:** `bash git clone https://github.com/hyperledger/fabric-lib-go.git cd fabric-lib-go mkdir -p poc_test cp malicious_checker.go poc_test/ cp main.go poc_test/ cp go.mod poc_test/ cd poc_test go mod tidy`

2. **Run the Test Server:** `bash go run main.go`

3. **Send HTTP Request:** `bash curl http://localhost:8081/healthz`

Observed Output:

In response to the `curl` request, the following JSON output is received:

```
 {
   "status": "Service Unavailable",
   "time": "2025-09-15T15:06:33.321305557-04:00",
   "failed_checks": [
     {
       "component": "database-connection",
       "reason": "Internal database connection failed:
user=admin;password=supersecret;host=db.example.com;port=5432;api_key=YOUR_SECRET
     }
   ]
 }
```

As observed, the `reason` field contains sensitive information such as `user=admin`, `password=supersecret`, `host=db.example.com`, `port=5432`, and `api_key=YOUR_SECRET_API_KEY`. This clearly demonstrates the sensitive information disclosure through the `/healthz` endpoint.

Impact of Vulnerability

Sensitive information disclosure through the health check endpoint can have serious technical and business consequences. This information, while seemingly harmless, can be used as a starting point for more complex attacks.

Technical Impacts:

- **Exposure of Authentication Credentials:** Access to usernames and passwords (even if simulated) can allow attackers to gain unauthorized access to backend systems such as databases, internal API services, or other sensitive resources. This can lead to complete data breaches or system control.
- **Infrastructure Disclosure:** Information such as internal IP addresses, hostnames, ports, and types of running services provides attackers with a detailed map of the internal network infrastructure. This map can be used to identify other weaknesses and plan more targeted attacks.
- **Facilitation of Lateral Movement:** With authentication credentials and infrastructure details, attackers can easily move laterally within the network and gain access to other systems. This can significantly expand the scope of the attack.
- **Bypassing Security Mechanisms:** Information related to API keys can be misused to bypass authentication and authorization mechanisms in various services.

Business Impacts:

- **Privacy and Data Breach:** Disclosure of sensitive information can lead to privacy breaches for users and confidential company data. This can result in legal consequences, heavy fines, and loss of customer trust.
- **Financial Loss:** The costs associated with recovery after an attack, forensic investigations, customer notification, and compensation can be substantial. Additionally, loss of productivity and operational downtime can lead to significant financial losses.
- **Reputation and Brand Damage:** A security breach can severely damage an organization's reputation and brand. Loss of public and business partner trust can have long-term impacts on the business.
- **Legal and Regulatory Risks:** Disclosure of sensitive information may lead to violations of data protection regulations (such as GDPR or CCPA), which can result in heavy penalties for the organization.

In summary, this vulnerability is a serious weakness that can provide attackers with access to critical information, opening the door to malicious attacks with widespread consequences. Therefore, its remediation should be a high priority.

Remediation Recommendations

To mitigate and resolve this vulnerability, the following actions are recommended. These recommendations are based on security best practices and secure software development principles:

1. Sanitization of Error Messages

The most crucial step is to ensure that error messages returned to the client never contain sensitive information. This can be achieved by replacing detailed error messages with more generic ones.

Example (Go):

```go
 // Current (vulnerable) implementation:
// return errors.New("Database connection failed:
user=admin;password=secret;host=internal.db")

// Secure implementation:
return errors.New("Database connection failed") // Generic message
```

Within the `HealthChecker`, instead of returning the original error that might contain sensitive details, a generic and harmless error should be returned. Full error details should only be logged internally within the system (refer to recommendation 3).

2. Restrict Access to the `/healthz` Endpoint

The `/healthz` endpoint should not be publicly accessible without restrictions, especially in production environments.

- **Authentication:** Implement authentication mechanisms for accessing this endpoint. Only authorized users or services should be able to check the system's health status.
- **Network Access Restriction (IP Whitelisting):** Limit access to this endpoint only to specific IP addresses or IP ranges (e.g., the organization's internal network or monitoring services).

- **Use of VPN or Private Network:** Ensure that access to this endpoint is only possible through a Virtual Private Network (VPN) or a secure internal network.

3. Secure Logging and Separation of Concerns

Full and sensitive error details should be logged internally within the system but should never be exposed to external clients. This helps the operations team troubleshoot issues while maintaining information security.

Example (Go):

```go
 // Log full error details internally
logger.Error("Database connection failed",
    "user", dbUser,
    "host", dbHost,
    "error", err.Error())

// Return generic error to client
return errors.New("Database connection failed")
```

Using a centralized and secure logging system with restricted access is essential.

4. Environmental Configuration for Error Messages

Different error messages can be displayed in various environments (development, testing, production). In a development environment, displaying more details might be useful for debugging, but in a production environment, messages should be entirely generic.

Example (Go):

```go
 // Use environment variables to control error message details
if os.Getenv("ENVIRONMENT") == "production" {
    return errors.New("Service temporarily unavailable")
} else {
    return fmt.Errorf("Database connection failed: %v", err) // Display more
details in non-production environments
}
```

This approach provides flexibility while maintaining security in sensitive environments.

5. Code Review and Security Testing

Regular code reviews and security testing (such as penetration testing and vulnerability scanning) are crucial for identifying and addressing similar vulnerabilities in the future. Developers should receive adequate training in secure coding practices to prevent the introduction of new vulnerabilities.

Conclusion

The sensitive information disclosure vulnerability in the `healthz` component of the `hyperledger/fabric-lib-go` library is a serious security flaw that can have widespread consequences for organizations and users. This vulnerability allows attackers to access critical configuration information and authentication details, which can lead to secondary attacks and data breaches.

A detailed source code analysis reveals that the root cause of this issue lies in the lack of sanitization of internal error messages before their exposure through a public HTTP endpoint. By implementing the recommended solutions, including error message sanitization, restricting access to the `/healthz` endpoint, secure logging, and using appropriate environmental configurations, this vulnerability can be effectively mitigated, and system security can be protected.

Development and security teams should prioritize this issue and take necessary actions to resolve it as soon as possible. Furthermore, training developers in secure coding practices and conducting regular code reviews are essential to prevent the emergence of similar vulnerabilities in the future.

Appendices

PoC Files

- `malicious_checker.go`: Simulated vulnerable `HealthChecker` implementation.
- `main.go`: Test server code to initialize `HealthHandler` and register the malicious `HealthChecker`.
- `go.mod`: Go dependency management file for the PoC environment.

Author: Reza Habibi
Date: September 15, 2025
Report Version: 1.0