Example of Blacklist Input Validation

Student *Jeongmin Lee*
Contributor *Jonghyeok Lee, Chanhang Lee, Nurnazihah Intan, Pedro Vidal Villalba*

1 Overview

This document demonstrates the implementation of a simple server and illustrates command-injection vulnerabilities that can arise within such a server, as well as how to mitigate them using a blacklisting approach. The programming language and framework used for the exercise are Go and the Fiber web framework.

1.1 What is a command-injection vulnerability?

A command-injection vulnerability permits an attacker to execute system commands on the host by leveraging improperly handled user input. This class of vulnerability typically occurs when user-supplied data is incorporated directly into system command invocations. For example, if a server constructs and executes system commands based on user input without adequate sanitization, an attacker can supply malicious input that results in arbitrary command execution. Consequences include unauthorized system access, data exfiltration, and denial-of-service (DoS) conditions.

1.2 What is the blacklisting technique?

Blacklisting is an input-validation strategy that blocks inputs matching a predefined set of disallowed patterns (a "blacklist"). In practice, the application rejects inputs that contain specific strings or patterns known to be associated with malicious activity. Blacklisting is simple and fast to implement, but it cannot guarantee complete protection because it is difficult to anticipate and enumerate every possible malicious input. For this reason, blacklisting is often used in conjunction with whitelisting and other defensive measures to provide stronger security.

2 Server Implementation and Defense Technique Application

This section briefly describes the server implementation and compares the results of applying the defense technique with those of not applying it. The complete code will be included in the appendix. The key parts are the `vulnHandler` and `safeHandler` functions.

2.1 Vulnerable server implementation

The example below summarizes the core behavior of a simple practice server. The server uses the Fiber framework and exposes two primary endpoints:

- `/vuln?ip=` : a vulnerable handler (`vulnHandler`). It constructs a raw command string from user input and executes it via a shell. This approach is susceptible to command injection using shell metacharacters (e.g., `|` or `;`) that allow an attacker to append or chain additional commands.
- `/safe?ip=` : a defended handler (`safeHandler`). It performs checks against a blacklist of patterns (e.g., `whoami` or the pipe character `|`) to block suspicious input, and then invokes external programs using an argument-list (i.e., without going through a shell). In the experiment the IP validation pipeline is commented out; you may enable it when trying the exercise if desired.

The fundamental difference is the invocation method. `vulnHandler` concatenates user input and passes it to `/bin/sh -c`, so any extra commands embedded in the input are interpreted and executed by the shell. By contrast, `safeHandler` calls the program with an argument vector (for example, `exec.Command(\ping", \-c", \1", ip)`), thereby avoiding shell interpretation.

2.2 Local Environment Setup and Testing

1. Run the server: from the server directory,

```
cd server
go run test_server.go
```

   The server will bind to :8080.

2. Check for vulnerabilities: Inject the whoami command with a command separator into the vulnerable endpoint.

```
curl -i 'http://localhost:8080/vuln?ip=127.0.0.1%7Cwhoami'
```

Listing 1: Test example for the vulnerable endpoint

   This request is internally translated to "ping -c 1 127.0.0.1|whoami" and sent to the shell, allowing us to observe the output of whoami (the executing user name) in the server response.

3. Check the defense: Send the same payload to the defended handler.

```
curl -i 'http://localhost:8080/safe?ip=127.0.0.1%7Cwhoami'
```

Listing 2: Test example for the defended handler

   The defended handler blocks the payload through input format (regex) and blacklist checks, returning an HTTP 400 response with the message "input blocked by blacklist".

Summary of the exercise results:

- /vuln : malicious input is passed through a shell and the appended command (e.g., whoami) is executed.
- /safe : the blacklist detects command-related tokens and the request is rejected (blocked immediately).

```
potato@potatoui-MacBookAir server % curl -i -s 'http://localhost:8080/vuln?ip=127.0.0.1%7Cwhoami'
HTTP/1.1 200 OK
Date: Tue, 30 Sep 2025 07:12:28 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 7

potato

<----------------------------->
potato@potatoui-MacBookAir server % curl -i -s 'http://localhost:8080/safe?ip=127.0.0.1%7Cwhoami'
HTTP/1.1 400 Bad Request
Date: Tue, 30 Sep 2025 07:12:55 GMT
Content-Type: text/plain; charset=utf-8
Content-Length: 53

input blocked by blacklist: suspicious token detected%
```

Listing 3: Experimental results

3 Analysis and Limitations

- The blacklist approach is easy to implement and can quickly block some classes of attacks, but it is difficult to cover every variant (e.g., bypass patterns, case variations, encodings).
- A more robust strategy combines whitelisting (allowing only explicitly permitted input patterns) with argument-based execution that avoids the shell. In addition, the principle of least privilege, proper output encoding, and comprehensive logging and monitoring should be applied.
- In experimental settings, the outcome may vary depending on the execution privileges for system commands (e.g., ping) and the network policy in effect. In CI environments or restricted containers, ICMP may be blocked and the command may fail.

A Appendix: Full server code

```
package main

import (
  "fmt"
  "io"
  "log"
  "net/http"
  "os/exec"
  "regexp"
  "testing"

  "github.com/gofiber/fiber/v2"
)

// This file provides a tiny HTTP server using Fiber to experiment with
// command-injection defenses. Routes:
//  - GET /vuln?ip=<value> : vulnerable example that invokes the shell (unsafe)
//  - GET /safe?ip=<value> : safe example that validates input and runs command

func vulnHandler(c *fiber.Ctx) error {
  ip := c.Query("ip")
  if ip == "" {
    return c.Status(400).SendString("missing ip parameter")
  }
  // Vulnerable: constructs a shell command using user input
  cmdStr := "ping -c 1 " + ip
  out, err := exec.Command("/bin/sh", "-c", cmdStr).CombinedOutput()
  if err != nil {
    return c.Status(500).SendString(fmt.Sprintf("command error: %v, output: %s", err, out))
  }
  return c.Status(200).Send(out)
}

var ipv4Regexp = regexp.MustCompile(`^([0-9]{1,3}\.){3}[0-9]{1,3}$`)

func safeHandler(c *fiber.Ctx) error {
  ip := c.Query("ip")
  if ip == "" {
    return c.Status(400).SendString("missing ip parameter")
  }
  // Input validation: simple IPv4 regex (note: not full validation of octet range)
  if !ipv4Regexp.MatchString(ip) {
    return c.Status(400).SendString("invalid ip format")
  }
  // Blacklist check: block obvious injection patterns
  if contains := regexp.MustCompile(`(?i)\bwhoami\b|\|`).MatchString(ip); contains {
    return c.Status(400).SendString("input blocked by blacklist: suspicious token detected")
  }
  // Safe: run ping without a shell by passing args directly
  out, err := exec.Command("ping", "-c", "1", ip).CombinedOutput()
  if err != nil {
    return c.Status(500).SendString(fmt.Sprintf("command error: %v, output: %s", err, out))
  }
  return c.Status(200).Send(out)
}

func startServer() *fiber.App {
  app := fiber.New()
  app.Get("/vuln", vulnHandler)
  app.Get("/safe", safeHandler)
  go func() {
    log.Printf("starting fiber server on :8080")
    if err := app.Listen(":8080"); err != nil {
      log.Fatalf("fiber server failed: %v", err)
    }
  }()
  return app
}

func TestServerEndpoints(t *testing.T) {
```

```
  _ = startServer()

  // Simple smoke test on /safe (should return 200 or command error if ping not permitted)
  resp, err := http.Get("http://localhost:8080/safe?ip=127.0.0.1")
  if err != nil {
    t.Fatalf("failed to GET /safe: %v", err)
  }
  defer resp.Body.Close()
  body, _ := io.ReadAll(resp.Body)
  t.Logf("/safe status=%d body=%s", resp.StatusCode, string(body))

  // Test /vuln endpoint as well
  resp2, err := http.Get("http://localhost:8080/vuln?ip=127.0.0.1")
  if err != nil {
    t.Fatalf("failed to GET /vuln: %v", err)
  }
  defer resp2.Body.Close()
  b2, _ := io.ReadAll(resp2.Body)
  t.Logf("/vuln status=%d body=%s", resp2.StatusCode, string(b2))
}

func main() {
  app := startServer()
  defer app.Shutdown()
  select {}
}
```

Listing 4: Example server implementation using Go and the Fiber framework