Example of Blacklist Input Validation

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

1 개요

이 문서에서는 실제로 간단한 서버를 구현하고, 해당 서버에서 발생할 수 있는 명령어 주입 취약점과 이를 블랙리스팅 기법으로 방어하는 것을 보입니다. 실습에 사용한 언어와 프레임워크는 Go 언어와 Fiber 웹 프레임워크입니다.

1.1 명령어 주입 취약점이란?

명령어 주입(command injection) 취약점은 공격자가 시스템 명령어를 실행할 수 있도록 허용하는 보안 취약점입니다. 이 취약점은 주로 사용자 입력값을 시스템 명령어에 직접 전달할 때 발생합니다. 예를 들어, 사용자가 입력한 값을 기반으로 시스템 명령어를 생성하고 실행하는 경우, 공격자는 악의적인 입력값을 통해 임의의 명령어를 실행할 수 있습니다. 이로 인해 시스템에 대한 무단 접근, 데이터 유출, 서비스 거부(DoS) 공격 등이 발생할 수 있습니다.

1.2 블랙리스팅 기법이란?

블랙리스팅(blacklisting) 기법은 입력값 검증(input validation) 방법 중 하나로, 허용되지 않는(악의적인) 입력값을 미리 정의된 목록(블랙리스트)과 비교하여 차단하는 방식입니다. 즉, 특정한 패턴이나 문자열이 포함된 입력값을 거부하는 방식입니다. 블랙리스팅 기법은 구현이 간단하고 빠르다는 장점이 있지만, 모든 악의적인 입력값을 사전에 정의하기 어렵기 때문에 완벽한 방어를 제공하지 못할 수 있습니다. 따라서, 블랙리스팅 기법은 화이트리스팅(whitelisting) 기법과 함께 사용되는 경우가 많습니다.

2 서버 구현 및 방어 기법 적용

해당 단락에서는 간략히 서버 구현에 대해 설명하고 방어 기법을 적용한 결과와 그렇지 않은 결과를 비교합니다. 전체적인 코드에 대해서는 부록으로 첨부하겠습니다. 핵심적인 부분은 `vulnHandler`와 `safeHandler` 함수입니다.

2.1 취약한 서버 구현

아래 예시는 간단한 실습용 서버의 핵심 동작을 요약한 것입니다. 서버는 Fiber 프레임워크를 사용하며, 두 개의 주요 엔드포인트를 제공합니다:

- `/vuln?ip=<값>` : 취약한 핸들러(`vulnHandler`). 사용자 입력을 원시 문자열로 결합하여 셸을 통해 실행합니다. 이 방식은 명령어 구분자(예: `|` 또는 `;`)와 같은 메타문자를 이용한 명령어 주입에 취약합니다.
- `/safe?ip=<값>` : 방어된 핸들러(`safeHandler`). 블랙리스트 패턴(예: `whoami` 또는 파이프 문자 `|`)을 검사하여 의심스러운 입력을 차단한 뒤, 셸을 거치지 않고 인수 리스트 방식으로 외부 프로그램을 실행합니다. 실험에서는 ip 검증 파이프라인은 주석처리 해두었지만, 실험 해보실 경우 주석 해제후 실험해보셔도 좋습니다.

핵심 차이는 실행 방식에 있습니다. `vulnHandler`는 아래와 같이 사용자 입력을 직접 결합하여 exttt/bin/sh -c로 전달하기 때문에 입력에 포함된 추가 명령이 그대로 실행됩니다. 반면 `safeHandler`는 인수 리스트를 사용하여 프로그램을 호출하므로(예: `exec.Command("ping", "-c", "1", ip)`) 셸 해석을 통과하지 않습니다.

2.2 실습 방법(로컬 환경)

1. 서버 실행: 서버 디렉터리로 이동 후

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

서버가 :8080 에 바인딩됩니다.

2. 취약점 확인: 취약한 엔드포인트에 명령 구분자와 함께 whoami를 주입합니다.

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

Listing 1: 취약한 엔드포인트에 대한 테스트 예시

이 요청은 내부적으로 "ping -c 1 127.0.0.1|whoami" 와 같이 셸에 전달되며, 서버 응답에 whoami의 출력(실행 계정 이름)이 포함되는 것을 관찰할 수 있습니다.

3. 방어 확인: 동일한 페이로드를 방어 핸들러에 전송합니다.

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

Listing 2: 방어 핸들러에 대한 테스트 예시

방어 핸들러는 입력 형식(정규식)과 블랙리스트 검사에서 페이로드를 차단하여 HTTP 400 응답과 함께 "input blocked by blacklist" 메시지를 반환합니다.

실습 결과 요약:

- /vuln : 악의적 입력이 셸을 통해 실행되어 추가 명령(예: whoami)이 수행됨.
- /safe : 블랙리스트 검사에 의해 명령어와 관련된 토큰이 검출되어 요청이 거부됨(즉시 차단).

```
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: 실험 결과

분석 및 한계:

- 블랙리스트 방식은 구현이 쉽고 일부 공격을 빠르게 차단할 수 있으나, 모든 변형을 포괄하기 어렵습니다(우회 패턴, 대소문자, 인코딩 등).
- 더 안전한 접근법은 화이트리스트(허용된 입력 패턴만 통과)와 셸을 우회한 인수 기반 실행을 조합하는 것입니다. 또한 최소 권한 원칙, 출력 인코딩, 로깅 및 모니터링을 함께 적용해야 합니다.
- 실습 환경에서는 시스템 명령(예: ping)의 실행 권한 및 네트워크 정책에 따라 결과가 달라질 수 있습니다. CI나 제한된 컨테이너에서는 ICMP가 차단되어 명령이 실패할 수 있습니다.

A 부록: 전체 서버 코드

```
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: Go 언어와 Fiber 프레임워크를 사용한 서버 구현 예시