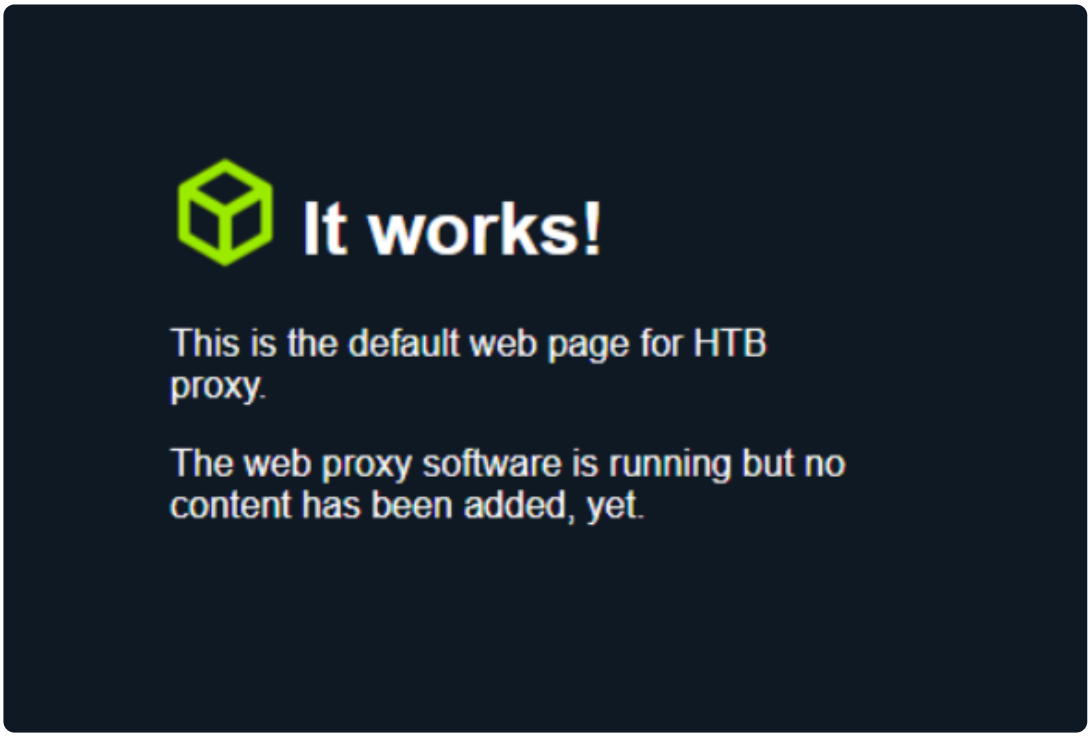
HTB Proxy

```
Your team is tasked to penetrate the internal networks of a raider base in order to acquire explosives, scanning their ip ranges revealed only one alive host running their own custom implementation of an HTTP proxy, have you got enough wit to get the job done?
```

web_htb_proxy.zip (https://2826773145-files.gitbook.io/~/files/v0/b/gitbook-x-prod.appspot.com/o/spaces%2F1uhiofTFnZvCKEvs4fJk%2Fuploads%2FfXNnZaku6PSnWviSA7IY%2Fweb_htb_proxy.zip?alt=media&token=828ff038-e0ca-4689-9cef-072ac3bd5b43)

Challenge cung cấp cho chúng ta trang web tĩnh với nội dung cho biết một reverse proxy đang chạy



Hãy đi sâu vào phân tích source code ta có thể thấy, challenge cung cấp cho ta hai đoạn code xử lý với reverse proxy được xử lý bằng golang và backend được xử lý thông qua NodeJS

```
.
├── Dockerfile
├── build_docker.sh
├── challenge
│   ├── backend
│   │   ├── index.js
│   │   └── package.json
│   └── proxy
│       ├── go.mod
│       ├── includes
│       │   └── index.html
│       ├── main.go
│       └── test.py
├── config
│   └── supervisord.conf
├── entrypoint.sh
└── flag.txt
```

Command Injection in ip-wrapper

Ở đoạn code xử lý backend, ta thấy hai routes được xử lý đó là `/getAddresses` và `/flushInterface`. Ở `/flushInterface`, có một đoạn xử lý đặc biệt `ipWrapper.addr.flush(interface)` giúp xóa tất cả các địa chỉ IP trong một `interface` cụ thể. Khi đi sâu vào lib, ta có thể thấy ở `flush`, đoạn code xử lý `exec(`ip address flush dev ${interfaceName}`, (error, stdout, stderr)`

```
function flush(interfaceName) {
    return new Promise((resolve, reject) => {
        exec(`ip address flush dev ${interfaceName}`, (error, stdout, stderr) => {
            if (stderr) {
                if(stderr.includes('Cannot find device')) {
                    reject(new Error('Cannot find device ' + interfaceName));
                } else {
                    reject(new Error('Error flushing IP addresses: ' + stderr));
                }
                return;
            }

            resolve();
        });
    });
```

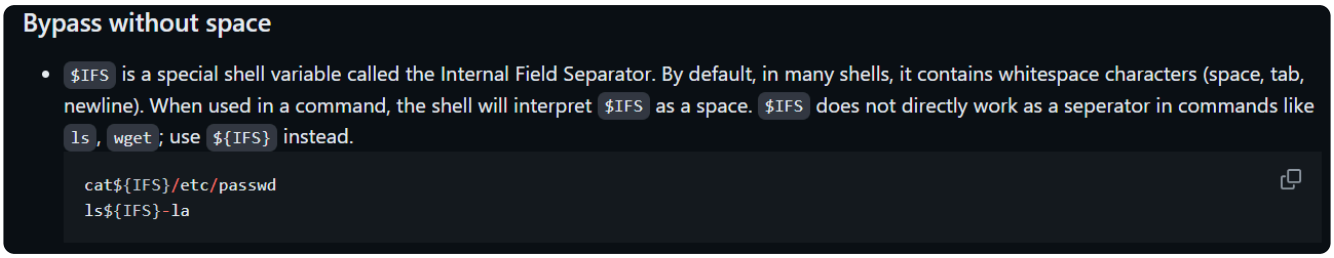
với `interfaceName` ở đây được truyền vào thông qua params `interface` nói trên.

Wait, we can trigger RCE here !!!

Thay vì truyền một `interface` bình thường, ta hoàn toàn có thể trigger RCE thông qua việc truyền command `; mv /fl* /app/proxy/includes/index.html` . Tuy nhiên ta vẫn cần phải bypass qua xử lý middleware

```
const validateInput = (req, res, next) => {
    const { interface } = req.body;

    if (
        !interface ||
        typeof interface !== "string" ||
        interface.trim() === "" ||
        interface.includes(" ")
    ) {
        return res.status(400).json({message: "A valid interface is required"});
    }

    next();
}
```

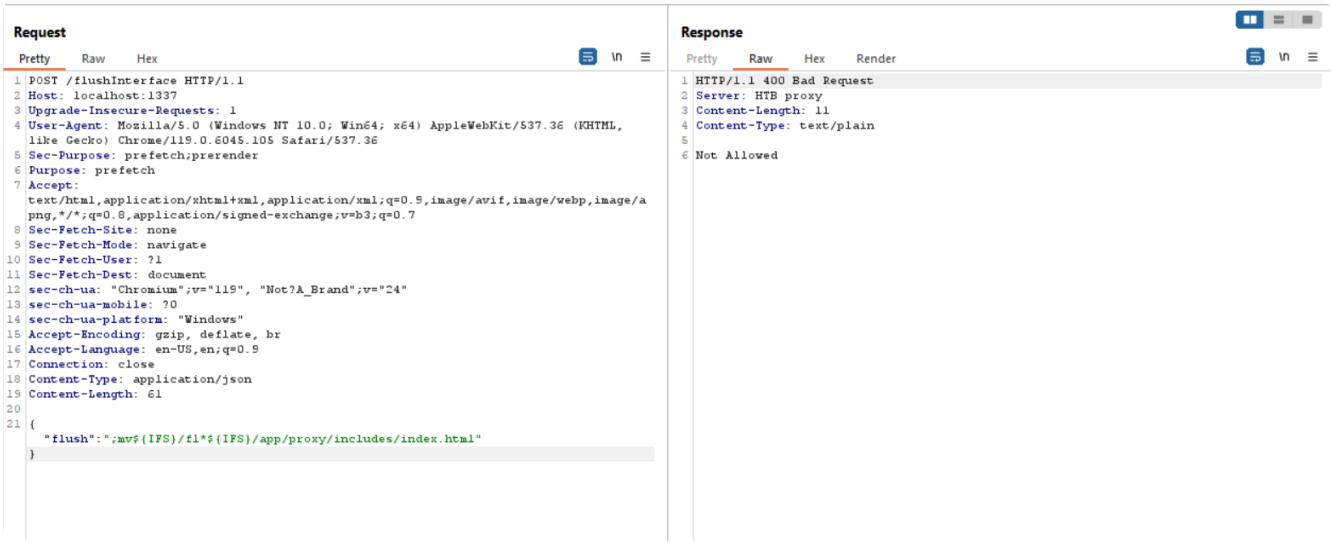
Ở đây ta thấy nó loại bỏ các khoảng trống trong `interface` truyền vào thông qua `trim()` , để bypass ta có thể truyền vào `${IFS}` . ($IFS là một biến đặc biệt trong shell nó chứa các ký tự khoảng trắng (khoảng trắng, tab, dấu xuống dòng)



Như vậy thông qua phương thức `flush` ta hoàn toàn có thể trigger RCE thông qua

`;mv${IFS}/fl*${IFS}/app/proxy/includes/index.html`

Tuy nhiên vấn đề xảy ra khi ta không thể truy cập vào endpoint `flushInterface`



Có vẻ đoạn code Golang xử lý reverse proxy đã chặn việc truy cập vào `/flushInterface`

Ta tiếp tục phân tích đoạn code xử lý reverse proxy

```
if strings.Contains(strings.ToLower(request.URL), string([]byte{102, 108, 117, 115, 104, 105, 110, 116, 101, 114, 102, 97, 99, 101})) {
    var responseText string = badReqResponse("Not Allowed")
    frontendConn.Write([]byte(responseText))
    frontendConn.Close()
    return
}
```

Đúng như mình dự đoán ở đây Golang đã chặn việc sử dụng endpoint `flushInterface` thông qua việc nó kiểm tra xem URL của yêu cầu có chứa chuỗi "flushinterface" (không phân biệt hoa thường) hay không và trả về `Not Allowed` .

SSRF to get content

Tiếp tục phân tích script xử lý golang mình tìm thấy một số đoạn code đáng chú ý:

```
if request.URL == string([]byte{47, 115, 101, 114, 118, 101, 114, 45, 115, 116, 97, 116, 1
                var serverInfo string = GetServerInfo()
                var responseText string = okResponse(serverInfo)
                frontendConn.Write([]byte(responseText))
                frontendConn.Close()
                return
        }
```

Tại endpoint `/server-status` , nó sẽ trả về `GetServerInfo()`

```
func GetServerInfo() string {
        hostname, err := os.Hostname()
        if err != nil {
                hostname = "unknown"
        }

        addrs, err := net.InterfaceAddrs()
        if err != nil {
                addrs = []net.Addr{}
        }

        var ips []string
        for _, addr := range addrs {
                if ipNet, ok := addr.(*net.IPNet); ok && !ipNet.IP.IsLoopback() {
                        if ipNet.IP.To4() != nil {
                                ips = append(ips, ipNet.IP.String())
                        }
                }
        }

        ipList := strings.Join(ips, ", ")

        info := fmt.Sprintf("Hostname: %s, Operating System: %s, Architecture: %s, CPU Cou
                hostname, runtime.GOOS, runtime.GOARCH, runtime.NumCPU(), runtime.Version()

        return info
}
```

bao gồm các thông tin như Hostname, Operating System, Architecture, ... và đặc biệt là IPs. Ta hãy chú ý đến method IsLoopBack (https://pkg.go.dev/net#IP.IsLoopback), nó sẽ check xem địa chỉ IP có phải địa chỉ LoopBack hay không. Địa chỉ LoopBack là các địa chỉ như 127.0.0.1, ::1, ... Như vậy mục đích ở đây là loại bỏ các ip localhost như 127.0.0.1 hay ::1.

Tiếp đến ta có thể thấy một số đoạn code xử lý blacklist

blacklistCheck

```
func blacklistCheck(input string) bool {
        var match bool = strings.Contains(input, string([]byte{108, 111, 99, 97, 108, 104,
                strings.Contains(input, string([]byte{48, 46, 48, 46, 48, 46, 48})) || // 0
                strings.Contains(input, string([]byte{49, 50, 55, 46})) || // 127.
                strings.Contains(input, string([]byte{49, 55, 50, 46})) || // 172.
                strings.Contains(input, string([]byte{49, 57, 50, 46})) || // 192.
                strings.Contains(input, string([]byte{49, 48, 46})) // 10.

        return match
}
```

Kiểm tra bất kì chuỗi con nào truyền vào có chứa các giá trị nhạy cảm như `0.0.0.0` , `127.` , ...

checkMaliciousBody

```
func checkMaliciousBody(body string) (bool, error) {
        patterns := []string{
                "[`;&|]",
                `\$\([^)]+\)`,
                `(?i)(union)(.*)(select)`,
                `<script.*?>.*?</script>`,
                `\r\n|\r|\n`,
                `<!DOCTYPE.*?\[.*?<!ENTITY.*?>.*?>`,
        }

        for _, pattern := range patterns {
                match, _ := regexp.MatchString(pattern, body)
                if match {
                        return true, nil
                }
        }
        return false, nil
}
```

Xem phần body có chứa các giá trị liên qua đến một số kiểu tấn công như CRLF, SQLi hay XSS, ...

checkIfLocalhost

```
func checkIfLocalhost(address string) (bool, error) {
        IPs, err := net.LookupIP(address)
        if err != nil {
                return false, err
        }

        for _, ip := range IPs {
                if ip.IsLoopback() {
                        return true, nil
                }
        }

        return false, nil
}
```

Kiểm tra xem chuỗi truyền vào có phải địa chỉ IP localhost hay không

```
var hostAddress string = hostArray[0]
var isIPv4Addr bool = isIPv4(hostAddress)
var isDomainAddr bool = isDomain(hostAddress)

if !isIPv4Addr && !isDomainAddr {
        var responseText string = badReqResponse("Invalid host")
        frontendConn.Write([]byte(responseText))
        frontendConn.Close()
        return
}

isLocal, err := checkIfLocalhost(hostAddress)
if err != nil {
        var responseText string = errorResponse("Invalid host")
        frontendConn.Write([]byte(responseText))
        frontendConn.Close()
        return
}

if isLocal {
        var responseText string = movedPermResponse("/")
        frontendConn.Write([]byte(responseText))
        frontendConn.Close()
        return
}

isMalicious, err := checkMaliciousBody(request.Body)
if err != nil || isMalicious {
        var responseText string = badReqResponse("Malicious request detected")
        prettyLog(1, "Malicious request detected")
        frontendConn.Write([]byte(responseText))
        frontendConn.Close()
        return
}
```

Kiểm tra xem địa chỉ truyền vào có phải là IPv4 hoặc một domain thông qua việc sử dụng 2 function: `isIPv4` và `isDomain`

isIPv4

```
func isIPv4(input string) bool {
        if strings.Contains(input, string([]byte{48, 120})) {
                return false
        }
        var ipv4Pattern string = `^(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(?:25[0-5]|2[(
        match, _ := regexp.MatchString(ipv4Pattern, input)
        return match && !blacklistCheck(input)
}
```

Tại `isIPv4` đoạn code sử dụng một đoạn regex lớn để check xem địa chỉ truyền vào có phải địa chỉ IP hợp lệ, thậm chí nó còn được sử dụng để tránh bypass việc encode IP thông qua việc bypass cả các kí tự chứa `0x`

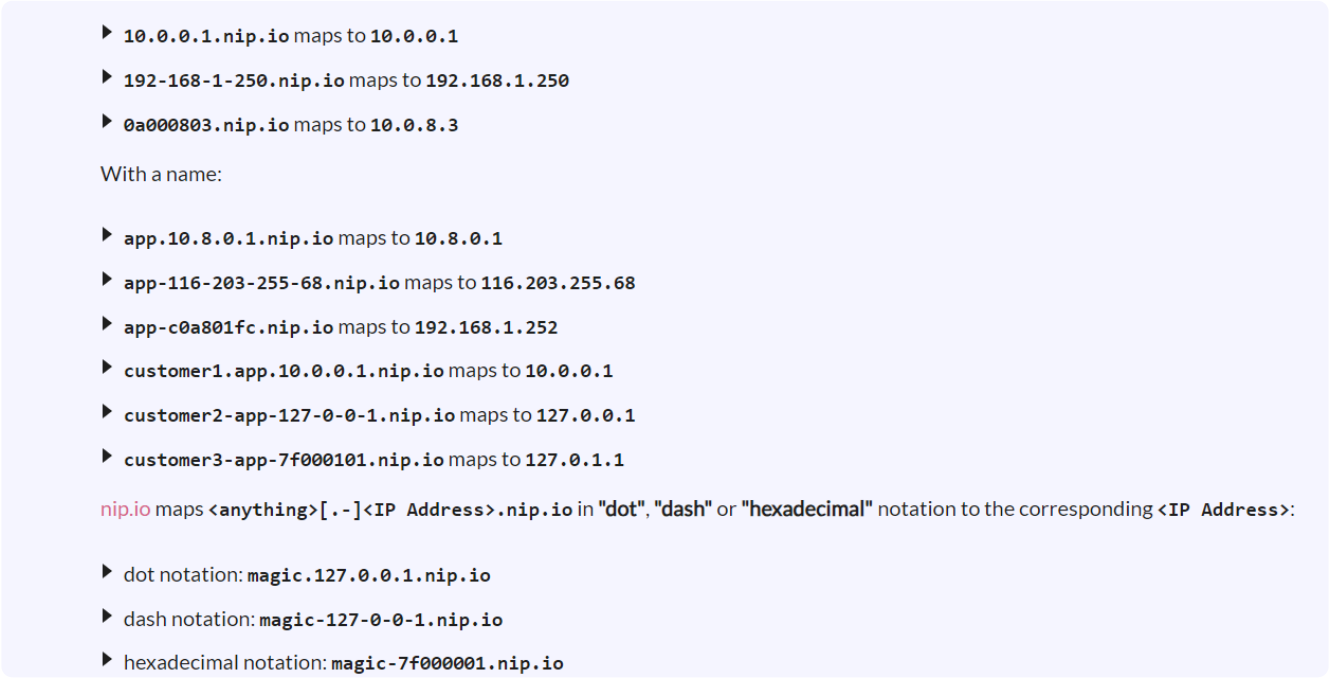
isDomain

```
func isDomain(input string) bool {
        var domainPattern string = `^[a-zA-Z0-9-]+(\.[a-zA-Z0-9-]+)*(\.[a-zA-Z]{2,})$`
        match, _ := regexp.MatchString(domainPattern, input)
        return match && !blacklistCheck(input)
}
```

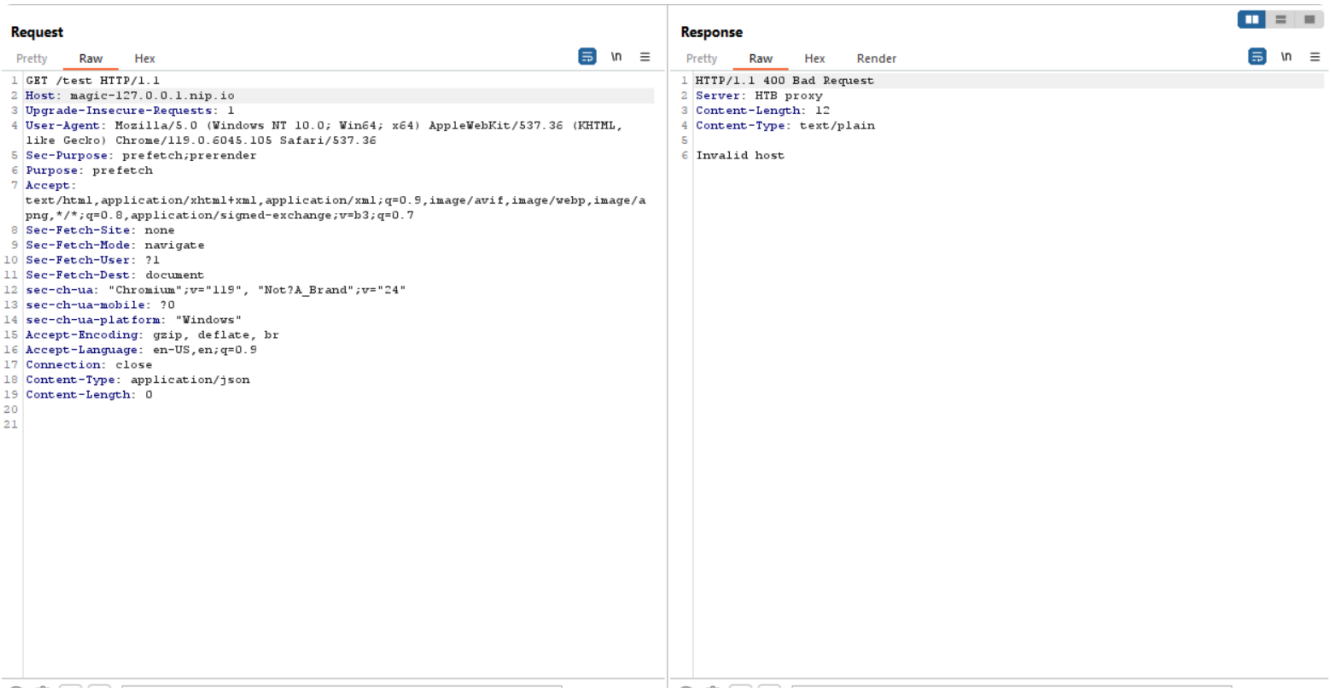
Tại `isDomain` , golang kiểm tra xem dữ liệu đầu vào có phải là một tên miền hợp lệ và không nằm trong blacklist

Ta thấy dường như những config này ngăn chặn chúng ta khỏi việc tấn công SSRF qua việc truyền vào IP localhost nhằm bypass proxy để có thể get respone từ server nodejs. Dường như tất cả các kĩ thuật thông thường nhằm trigger SSRF đều không thành công vượt qua Blacklist này. Mất một thời gian tìm hiểu thì mình biết được ngoài việc trigger SSRF thông thường ta còn có thể sử dụng một kỹ thuật đó là DNS Binding (https://unit42.paloaltonetworks.com/dns-rebinding/) thông qua việc sử dụng nip.io ().

Nip.io (http://Nip.io) là một dịch vụ DNS tự động chuyển đổi địa chỉ IP thành tên miền phụ dựa trên định dạng nhất định. Ví dụ, nếu địa chỉ IP của máy chủ là 192.0.2.1, ta có thể truy cập vào máy chủ đó bằng cách sử dụng tên miền phụ "192.0.2.1.nip.io (http://192.0.2.1.nip.io)".

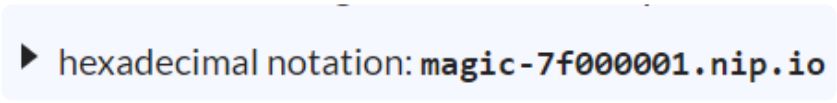


Vậy liệu ta có thể bypass SSRF blacklist thông qua việc sử dụng nip.io (http://nip.io)? Câu trả lời là chưa?

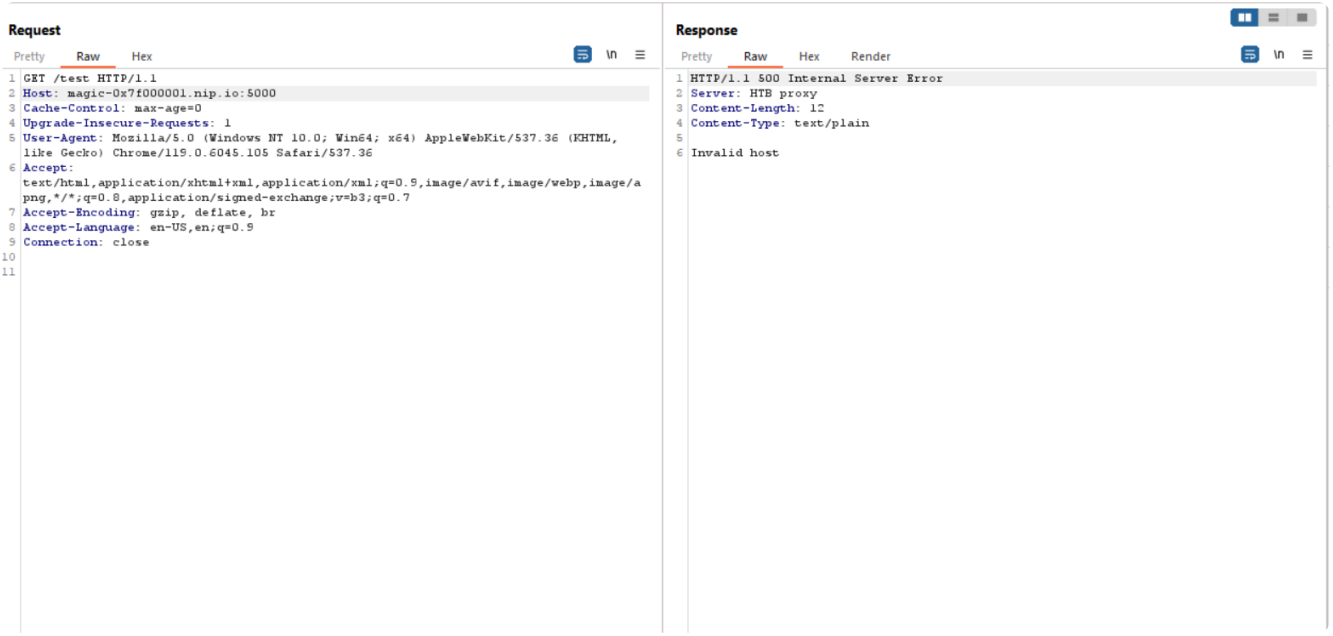


Vì giá trị Host truyền vào vẫn chứa giá trị `127.` nên là vẫn chưa thể bypass qua SSRF

Mình tiếp tục thử việc encode địa chỉ ip

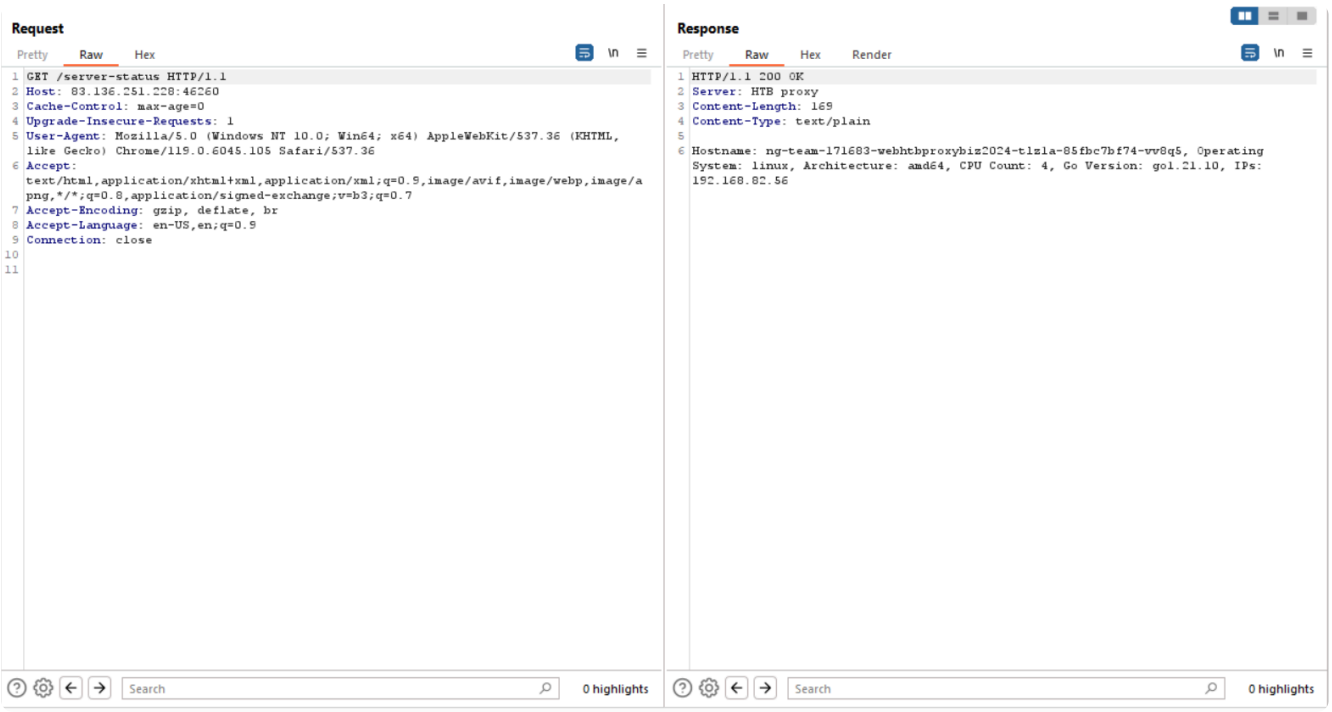


Tuy nhiên vấn đề này vẫn chưa được giải quyết do đoạn xử lý tại `checkIfLocalhost`

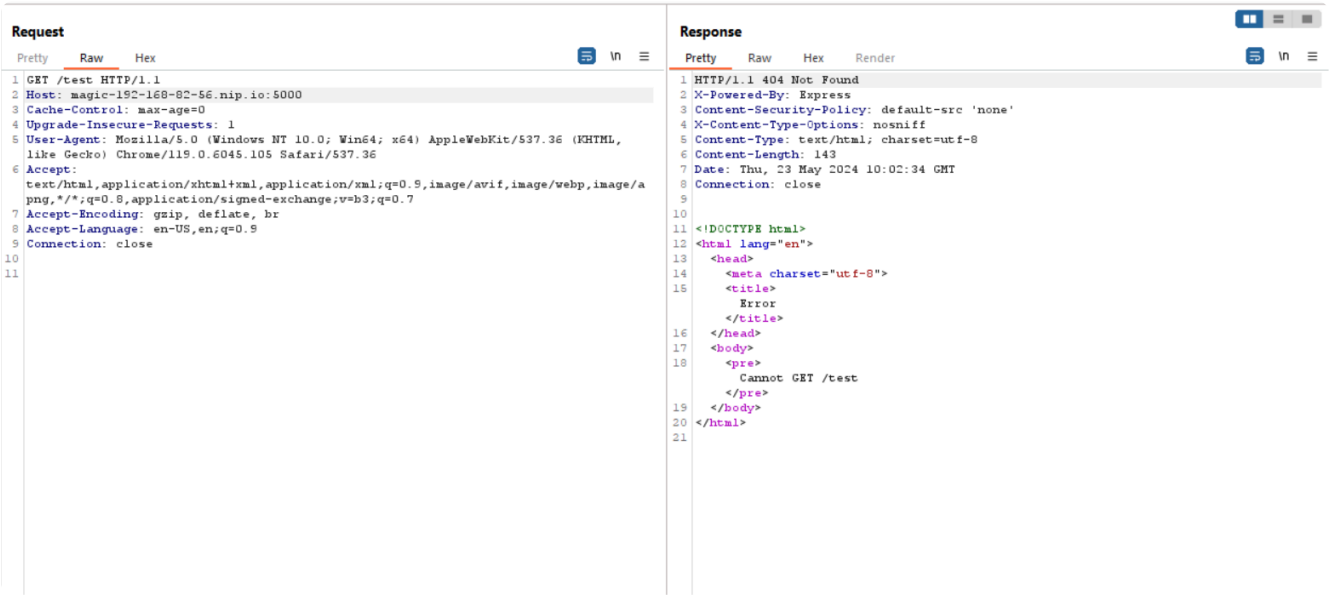


Hey Wait !!! I forgot something, miss `/server-status`

Khi truy cập vào endpoint này nó sẽ cung cấp cho chúng ta địa chỉ ip của container



`192.168.82.66` ở đây có vẻ chính là địa chị ip của phần backend xử lý được mở ở port 5000. Như đã phân tích từ đầu ta cần gọi các endpoint của Phần xử lý backend của NodeJS như `/flushInterface` để trigger RCE. Dó đó, ta có thể sử dụng ip này bypass qua blacklist check, thay vì sử dụng `192.` ta hoàn toàn có thể sử dụng được `192-` và điều này cũng được cho phép bởi nip.io (http://nip.io) `dash notation: magic-127-0-0-1.nip.io`



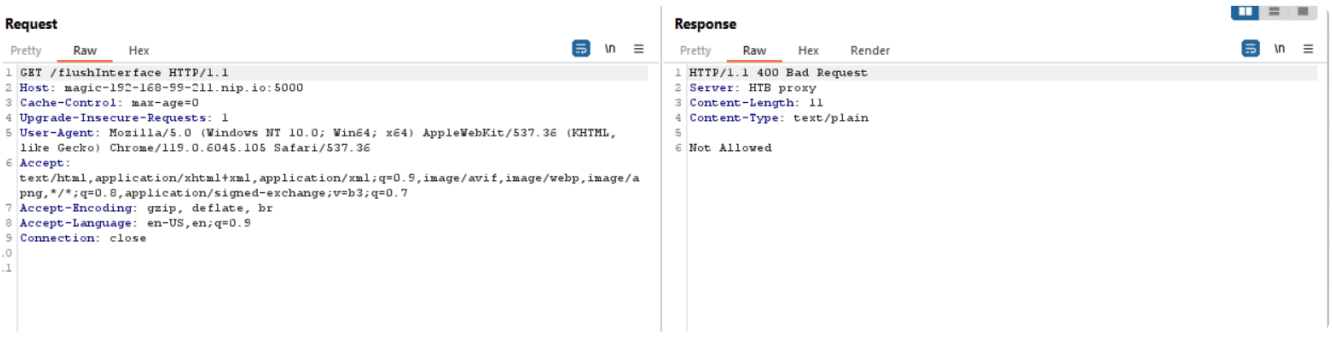
Như vậy ta đã bypass thành công được blacklist và trigger SSRF

Bypass proxy to get endpoint `/flushInterface` via HTTP Request Smuggling

Như đã phân tích từ đầu mục tiêu của chúng ta là get endpoint `/flushInterface` nhằm trigger RCE. Tuy nhiên do việc truy cập endpoint này đã bị proxy chặn từ trước đó

```
if strings.Contains(strings.ToLower(request.URL), string([]byte{102, 108, 117, 115, 104, 1
    var responseText string = badReqResponse("Not Allowed")
        frontendConn.Write([]byte(responseText))
        frontendConn.Close()
        return
}
```

Ta vẫn chưa thể thành công thực thi được RCE mặc dù đã bypass được SSRF blacklisst check

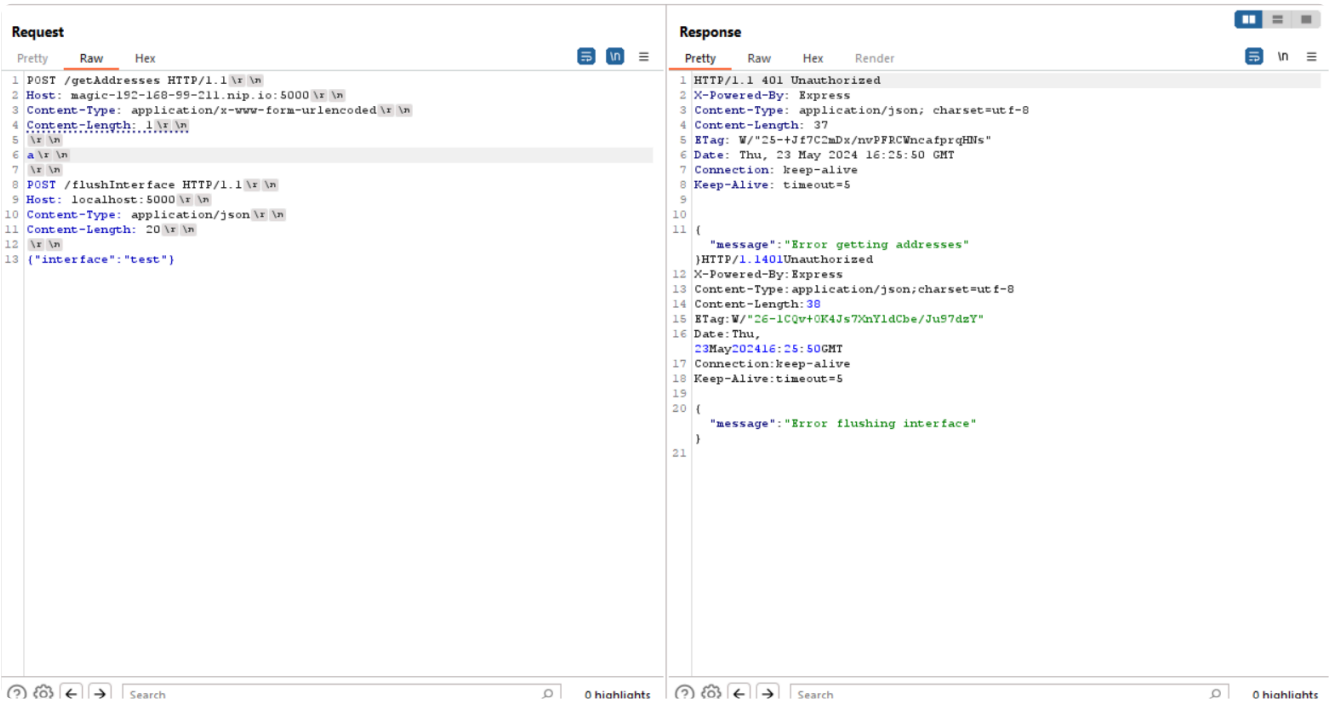


Mục tiêu giờ đây chuyển hướng sang HTTP Request Smuggling. Tại sao lại là Request Smuggling thì cũng tại đoạn golang xử lý proxy mình tìm được một vài xử lý thú vi trong đoạn code đối với request

```
func requestParser(requestBytes []byte, remoteAddr string) (*HTTPRequest, error) {
	var requestLines []string = strings.Split(string(requestBytes), "\r\n")
	var bodySplit []string = strings.Split(string(requestBytes), "\r\n\r\n")

	if len(requestLines) < 1 {
		return nil, fmt.Errorf("invalid request format")
	}

	var requestLine []string = strings.Fields(requestLines[0])
	if len(requestLine) != 3 {
		return nil, fmt.Errorf("invalid request line")
	}

	var request *HTTPRequest = &HTTPRequest{
		RemoteAddr: remoteAddr,
		Method:     requestLine[0],
		URL:        requestLine[1],
		Protocol:   requestLine[2],
		Headers:    make(map[string]string),
	}

	for _, line := range requestLines[1:] {
		if line == "" {
			break
		}

		headerParts := strings.SplitN(line, ": ", 2)
		if len(headerParts) != 2 {
			continue
		}

		request.Headers[headerParts[0]] = headerParts[1]
	}

	if request.Method == HTTPMethods.POST {
		contentLength, contentLengthExists := request.Headers["Content-Length"]
		if !contentLengthExists {
			return nil, fmt.Errorf("unknown content length for body")
		}

		contentLengthInt, err := strconv.Atoi(contentLength)
		if err != nil {
			return nil, fmt.Errorf("invalid content length")
		}

		if len(bodySplit) <= 1 {
			return nil, fmt.Errorf("invalid content length")
		}
		var bodyContent string = bodySplit[1]
		if len(bodyContent) != contentLengthInt {
			return nil, fmt.Errorf("invalid content length")
		}
```

```
                request.Body = bodyContent[0:contentLengthInt]
                return request, nil
        }

        if len(bodySplit) > 1 && bodySplit[1] != "" {
                return nil, fmt.Errorf("can't include body for non-POST requests")
        }

        return request, nil
}
```

Tại đây, golang phân tích phần thân bằng cách chỉ cần tách request thành một mảng nơi có ký tự `\r\n\r\n` . Đối với một yêu cầu HTTP thông thường, điều này là hợp lý vì phần thân thường nằm sau `\r\n\r\n` . Tuy nhiên, ta có thể thấy một điều kì lạ là nếu ta tiếp sử dụng `\r\n\r\n` để gửi request thứ hai (smuggling) và bằng cách cố định việc truyền `Content-Length: 1` và phần body với length tương đương, lúc này `request parser` sẽ coi phần thân lúc này chỉ là byte đầu với length là 1, điều này có nghĩa là khi `checkMaliciousBody` kiểm tra nó sẽ chỉ xem xét duy nhất byte này mà không tiến hành check requests thứ hai



Điều này xảy ra do việc sử dụng `\r\n\r\n` nhằm phân tách phần body và HTTP Header `var bodySplit []string = strings.Split(string(requestBytes), "\r\n\r\n")`
Ta có thể thấy mảng bodySplit được tạo ra thông qua việc tách chuỗi thành các chuỗi con

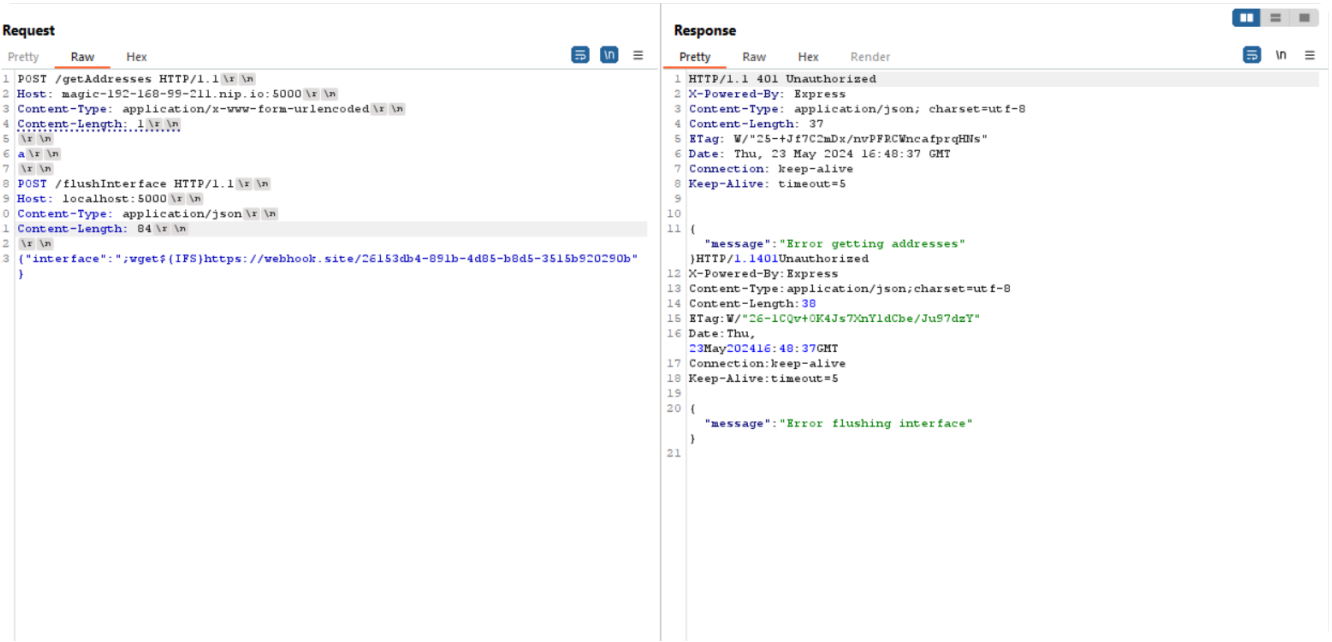
dựa trên kí tự phân tách `\r\n\r\n` Tuy nhiên phần được coi là body chỉ là phần thứ hai `var bodyContent string = bodySplit[1]` và các phân tiếp sau không được xét đến. Khi đó golang chỉ coi `a` là body và check Content-Length có thỏa mãn hay không

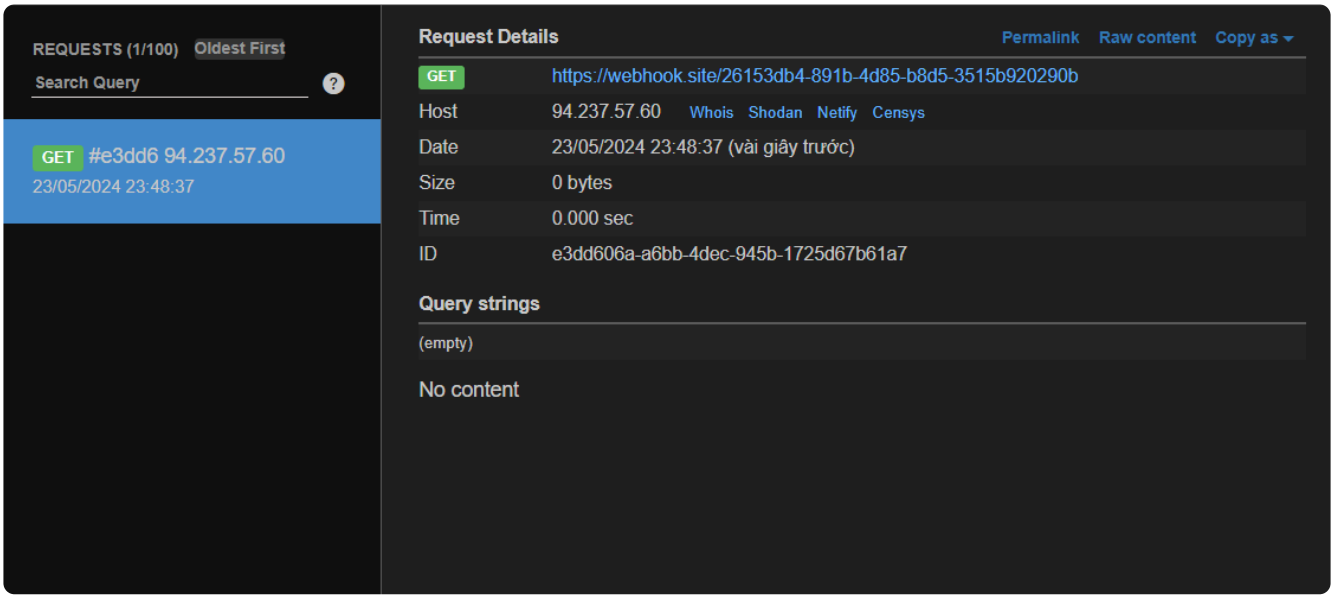
```
var bodyContent string = bodySplit[1]
    if len(bodyContent) != contentLengthInt {
        return nil, fmt.Errorf("invalid content length")
}
```

Như mình đã phân tích ta có thể bypass qua điều này bằng cách thao túng Content-Length truyền vào. Lúc này `checkMaliciousBody` kiểm tra, nó chỉ coi `a` là body, lúc này là có thể bypass qua `checkMaliciousBody` bằng cách truyền phần body và Content-Length hợp lệ. Do chưa gặp kí tự kết thúc

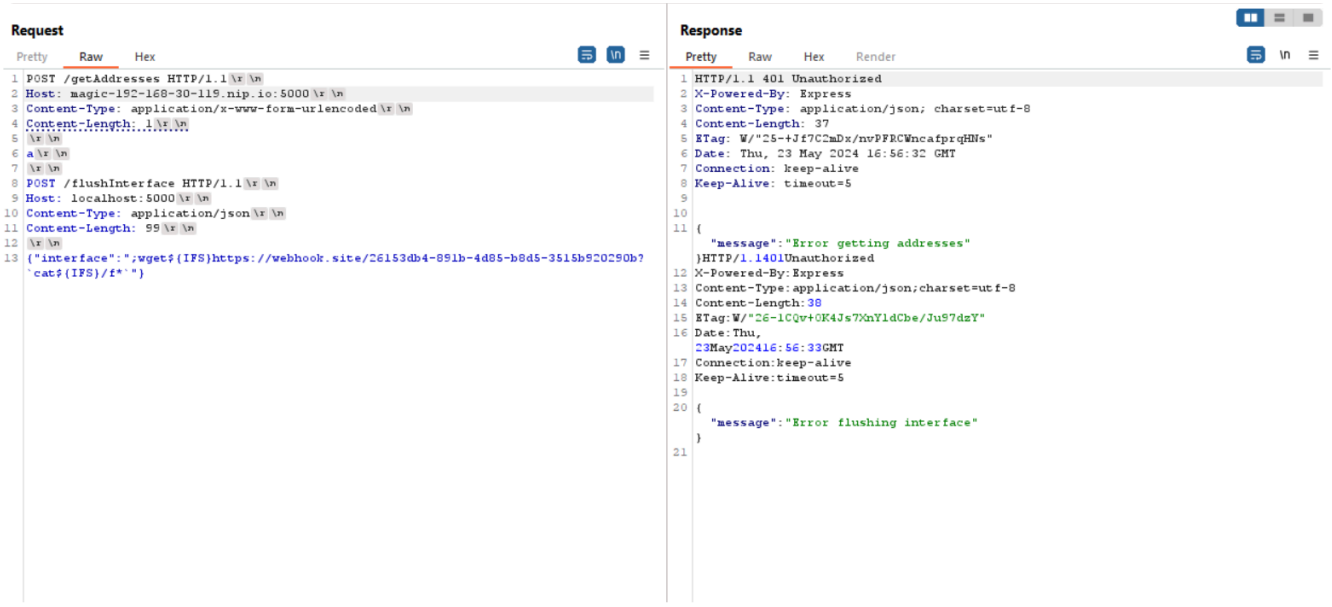
```
if line == ""
{
    break
}
```

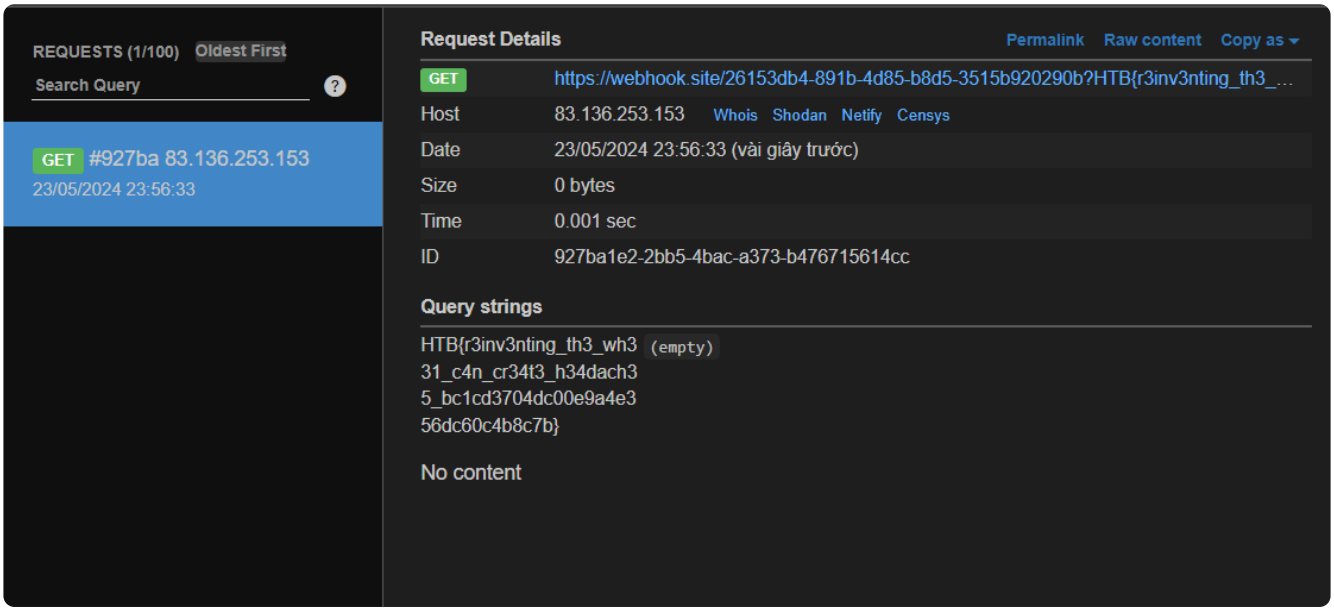
Lúc này golang tiếp tục check requests thứ hai, tại đây ta có thể trigger RCE mà không bị loại bỏ bởi blacklist





Exploit Chain: DNS re-binding => HTTP smuggling on custom HTTP reverse-proxy => command injection on ip-wrapper library.





Flag: `HTB{r3inv3nting_th3_wh331_c4n_cr34t3_h34dach35_bc1cd3704dc00e9a4e356dc60c4b8c7b}`

Skills Learned

Sử dụng kỹ thuật DNS re-binding để bypass localhost checks.

Sử dung HTTP smuggling thông qua việc tận dụng lỗ hổng trong http parsers.

RCE bypass space