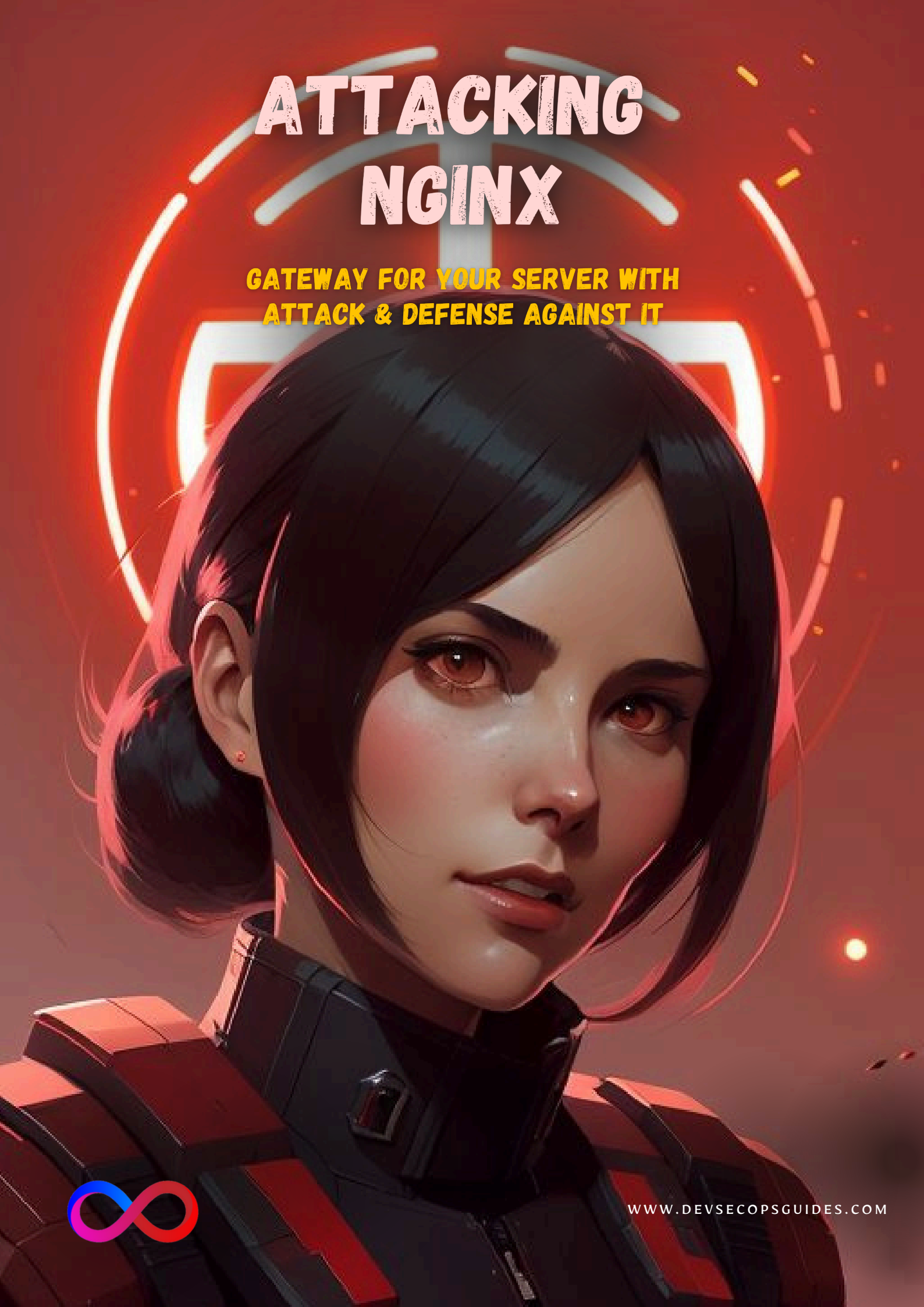
ATTACKING NGINX

GATEWAY FOR YOUR SERVER WITH ATTACK & DEFENSE AGAINST IT



WWW.DEVSECOPSGUIDES.COM

Attacking Nginx

• Aug 12, 2024 • 18 min read

Table of contents

- Examples of Vulnerable Configurations

Raw Backend Response Reading

- Example Scenario: Exposing Raw Backend Responses
- Example uWSGI Application:
- Nginx Configuration
  - Example Nginx Configuration:
- Example Invalid HTTP Request:
- Example Output for Invalid Request:
- Attack Scenario
- Mitigation Strategies

merge_slashes set to off

- 1. merge_slashes Directive
- Attack Scenario:
- Mitigation:
- 2. Malicious Response Headers
  - Attack Scenario:
  - Mitigation:
- 3. map Directive Default Value
  - Example:
  - Attack Scenario:
  - Mitigation:
- 4. DNS Spoofing Vulnerability
- Attack Scenario:
- Mitigation:
- 5. proxy_pass and internal Directives
- Example:

Show less

Nginx, a popular web server and reverse proxy, is a critical component in many web infrastructures, making it a prime target for attacks. Common vulnerabilities in Nginx configurations include improper handling of headers, such as `Upgrade` and `Connection`, which can lead to h2c smuggling attacks, allowing attackers to bypass security controls and access internal endpoints. Additionally, issues like insufficient path restrictions, unsafe variable use, and default settings like `merge_slashes` can expose the server to local file inclusion (LFI) attacks, HTTP request splitting, and other exploitation techniques. These vulnerabilities can be exploited to gain unauthorized access, manipulate traffic, or expose sensitive information.

To secure Nginx, it's crucial to apply best practices in configuration. This includes disabling or carefully managing headers that can be exploited, setting strict access controls on sensitive endpoints, and ensuring that directives like `merge_slashes` are configured appropriately to prevent URL-based attacks. Moreover, using features like `proxy_intercept_errors` and `proxy_hide_header` can help mask backend server errors and prevent the leakage of sensitive information. Regular audits of the Nginx

configuration, alongside keeping the software up to date, are essential steps in maintaining a robust security posture.

Missing Root Location in Nginx Configuration

When configuring an Nginx server, the `root` directive is crucial as it specifies the base directory from which the server serves files. Here's an example configuration:

```
server {
        root /etc/nginx;

        location /hello.txt {
                try_files $uri $uri/ =404;
                proxy_pass http://127.0.0.1:8080/;
        }
}
```

COPY

Explanation:

1. **Root Directive**: The `root /etc/nginx;` directive sets the base directory for all file requests. In this case, files will be served from the `/etc/nginx` directory.
2. **Location Directive**: The `location /hello.txt { ... }` block defines specific behavior for requests targeting `/hello.txt`. The `try_files` directive attempts to serve the file if it exists, and if not, returns a 404 error. The `proxy_pass` directive forwards the request to another server, here, `http://127.0.0.1:8080/`.

The Missing Root Location Issue:

The issue here arises from the lack of a `location / { ... }` block. This omission means that the root directive (`/etc/nginx`) applies globally, affecting all requests to the server, including those to the root path `/`. As a result, any request to the root path or other undefined locations could potentially access sensitive files within `/etc/nginx`.

For example, a request to `GET /nginx.conf` could serve the Nginx configuration file located at `/etc/nginx/nginx.conf`. This exposes sensitive server configurations, which could include paths, credentials, and other vital details.

Attack Scenario: Exploiting the Missing Root Location

An attacker could exploit this configuration by crafting specific HTTP requests to retrieve sensitive files:

1. **Requesting Sensitive Files**: An attacker can send a request like the following to retrieve the Nginx configuration file:

```
curl http://example.com/nginx.conf
```

2. **Accessing Other Files**: If the root directory is set to a broader location like `/etc`, attackers could access other sensitive files, such as:

```
curl http://example.com/passwd
```

This could expose the `/etc/passwd` file, which, while not containing password hashes, provides a list of user accounts on the system.

```
curl http://example.com/shadow
```

1. If accessible, this would expose the `/etc/shadow` file, which contains password hashes (although typically it should be protected by system permissions).

Mitigating the Risk

To prevent such attacks, consider the following mitigations:

1. **Restrict Root Directory**: Set the root directory to a less sensitive location, ensuring that only the necessary files are served:

```
root /var/www/html;
```

2. **Define a Root Location**: Explicitly define a `location / { ... }` block to handle requests to the root path:

```
server {
        root /var/www/html;

        location / {
                try_files $uri $uri/ =404;
        }

        location /hello.txt {
                try_files $uri $uri/ =404;
                proxy_pass http://127.0.0.1:8080/;
        }
}
```

This ensures that requests to the root path or any undefined location are handled securely.

3. **Use Permissions and Access Controls**: Ensure that sensitive files like `/etc/nginx/nginx.conf` are not readable by the Nginx worker processes, or are protected by file system permissions.

Unsafe Path Restriction in Nginx

Nginx configuration files often include path-based restrictions to secure sensitive directories or pages. A common example is restricting access to an administrative page or directory. Here's an example configuration:

```
location = /admin {
    deny all;
}

location = /admin/ {
    deny all;
}
```

Explanation:

1. **Path Restriction with `location` Directive:**
    - The first block, `location = /admin { deny all; }`, denies access to the exact `/admin` path.
    - The second block, `location = /admin/ { deny all; }`, denies access to the `/admin/` directory and any content within it.

These configurations seem to cover the most straightforward access attempts to the `/admin` and `/admin/` paths. However, Nginx's behavior in handling URL requests leaves room for bypassing these restrictions under certain conditions.

Potential Bypass Techniques

Even though the configuration appears secure, certain tricks can bypass these path restrictions, leading to unauthorized access:

1. **Trailing Slash Mismatch**: Nginx handles URLs with and without trailing slashes differently. If a developer misconfigures the paths, attackers can exploit this

behavior.
For instance, if `/admin` is denied but `/admin.` (with a trailing dot or other characters) isn't, it might be treated as a different path:

```
curl http://example.com/admin.
```

If the server doesn’t explicitly block `/admin.`, it might bypass the `deny all;` directive and serve the page.

- **URL Encoding**: By encoding parts of the URL, an attacker might bypass simple string matching:

```
curl http://example.com/%61dmin
```

If the server doesn't properly decode the URL, `%61dmin` might be processed as a separate path, potentially bypassing the restriction.

- **Case Sensitivity**: Depending on the configuration, case sensitivity might be exploitable:

```
curl http://example.com/ADMIN
```

If Nginx is configured to treat paths case-insensitively (which is not the default but can happen in certain setups), this could bypass the restriction.

- **Adding Null Bytes**: In some server setups, appending a null byte (`%00`) might bypass restrictions by terminating the string processing early in some backend systems:

```
curl http://example.com/admin%00/
```

However, Nginx itself typically handles these correctly, but the backend application might not.

Attack Scenario: Exploiting Path Restriction Bypass

Scenario: Gaining Access to Restricted Admin Page

Given the following configuration:

```
location = /admin {
    deny all;
}

location = /admin/ {
    deny all;
}
```

An attacker tries to access the `/admin` directory to gain unauthorized access to an admin panel:

1. **Using URL Encoding**: The attacker attempts to access the admin panel using a URL-encoded version:
   `curl http://example.com/%61dmin`
   If the server does not decode the path correctly before applying the `deny` rules, this request might bypass the restriction.
2. **Adding a Trailing Dot**: The attacker tries appending a dot to the path:
   `curl http://example.com/admin.`
   This could trick the server into interpreting the path as different from `/admin`, potentially bypassing the restriction.

3. **Using Case Variation**: The attacker tries a different case:
   `curl http://example.com/ADMIN`
   If case sensitivity is not enforced, this could bypass the restriction.

Mitigation Strategies

To protect against these bypass techniques, it's important to strengthen your Nginx configuration:

Use Regex to Cover Variations: Use regular expressions to cover different variations of the path:

```
location ~* ^/admin(/|$) {
    deny all;
}
```

- This ensures that paths like `/admin`, `/admin/`, `/Admin`, and others are all denied.
- **Normalize Paths**: Ensure that URL paths are normalized (decoded, lowercased, etc.) before applying the rules.
- **Use Secure URI Handling**: Implement strict URI handling and filtering, ensuring that paths are handled consistently and securely across the entire stack.
- **Test for Bypass Techniques**: Regularly test your server configurations for potential bypass techniques using tools like curl and other security testing tools to identify weaknesses.

Unsafe variable use / HTTP Request Splitting

Nginx configurations must be carefully designed to avoid vulnerabilities like unsafe variable use and HTTP request splitting, which can lead to severe security issues. Below, we will explore how certain variables and regular expressions can introduce these vulnerabilities and how to mitigate them.

1. Unsafe Use of Variables: `$uri` and `$document_uri`

In Nginx, the `$uri` and `$document_uri` variables are often used to capture the request URI. However, these variables automatically decode URL-encoded characters, which can introduce vulnerabilities, especially when handling user inputs directly.

For example:

```
location / {
  return 302 https://example.com$uri;
}
```

COPY

In this configuration, the `$uri` variable is used directly in the redirection URL. If an attacker crafts a request like:

```
    http://localhost/%0d%0aDetectify:%20clrf
```

COPY

The Nginx server will decode the `%0d%0a` characters to `\r\n` (Carriage Return and Line Feed), potentially allowing the injection of a new header in the response:

```
HTTP/1.1 302 Moved Temporarily
Server: nginx/1.19.3
Content-Type: text/html
Content-Length: 145
Connection: keep-alive
Location: https://example.com/
Detectify: clrf
```

COPY

This is an example of **HTTP response splitting**, where the response is split into two, potentially allowing an attacker to inject malicious headers or even content.

2. Regex Vulnerabilities

Regex patterns used in Nginx configurations can also be vulnerable if they are not carefully constructed. For instance:

```
location ~ /docs/([^/])? { … $1 … }  # Vulnerable
```

This regex pattern does not check for spaces, which can lead to unexpected behavior if the input contains a space or a newline character.

A safer version would be:

```
location ~ /docs/([^/\s])? { … $1 …
}  # Not vulnerable (checks for spaces)
```

Alternatively:

```
location ~ /docs/(.*)? { … $1 … }  # Not
vulnerable (matches anything after /docs/)
```

Safe Configuration

To mitigate these risks, you should avoid using `$uri` and `$document_uri` directly in configurations where user input could be present. Instead, use `$request_uri`, which preserves the original, unmodified request, including any URL-encoded characters.

Example of a Safe Configuration:

```
location / {
  return 302 https://example.com$request_uri;
}
```

In this configuration, `$request_uri` preserves the URL encoding, preventing the server from accidentally interpreting characters that could lead to HTTP response splitting.

Attack Scenarios and Detection Techniques

1. CRLF Injection and HTTP Request Splitting

- **Attack Scenario**: An attacker tries to exploit HTTP request splitting by injecting CRLF characters into a request:

```
curl "http://localhost/%0d%0aX-Injected-Header:%20Test"
```

If the server is vulnerable, the response will include the injected header:

```
HTTP/1.1 302 Moved Temporarily
Server: nginx/1.19.3
Content-Type: text/html
Content-Length: 145
Connection: keep-alive
Location: https://example.com/
X-Injected-Header: Test
```

Detection Techniques: Test for misconfigurations using the following requests:

```
curl -I "https://example.com/%20X"  #
Should return any HTTP code if vulnerable
curl -I "https://example.com/%20H"  # Should return a 400 Bad Request
```

- If the first request succeeds and the second returns an error, the server is likely vulnerable.

2. Bypassing Path Restrictions Using Encoded Characters

- **Attack Scenario**: An attacker tries to bypass path restrictions by injecting encoded characters:

```
curl "http://localhost/lite/api/%0d%0aX-Injected-Header:%20Test"
```

If the Nginx server uses `$uri` in its `proxy_pass` directive, the request might be passed to the backend without proper sanitization, leading to header injection.

- **Detection Techniques**: Test paths with encoded spaces and special characters:

```
curl -I "http://company.tld/%20HTTP/1.1%0D%0AXXXX:%20x"
curl -I "http://company.tld/%20HTTP/1.1%0D%0AHost:%20x"
```

- The first request might succeed if the server is vulnerable, while the second should cause a 400 Bad Request error.

Examples of Vulnerable Configurations

Proxy Pass with $uri:

```
location ^~ /lite/api/ {
  proxy_pass http://lite-backend$uri$is_args$args;
}
```

Vulnerable because `$uri` is directly passed to the backend, which could lead to CRLF injection.

- **Rewrite with $uri**:

```
location ~ ^/dna/payment {
  rewrite ^/dna/([^/]+) /registered/main.pl?
cmd=unifiedPayment&context=$1&native_uri=$uri break;
  proxy_pass http://$back;
}
```

Vulnerable because `$uri` is used inside a query parameter, making it susceptible to manipulation.

- **S3 Bucket Access**:

```
location /s3/ {
  proxy_pass https://company-bucket.s3.amazonaws.com$uri;
}
```

Vulnerable because `$uri` is used directly in the `proxy_pass` URL.

Raw Backend Response Reading

Nginx's ability to intercept backend responses is a powerful feature designed to enhance security and user experience by masking internal errors and sensitive information. However, under certain circumstances, particularly with invalid HTTP requests, this mechanism can fail, leading to the unintended exposure of raw backend responses. Below, we'll explore how this vulnerability occurs, provide example configurations, and discuss potential attack scenarios.

Example Scenario: Exposing Raw Backend Responses

Consider a scenario where an Nginx server is fronting a uWSGI application. The uWSGI application may occasionally return an error response that includes sensitive information, such as custom headers or internal error messages.

Example uWSGI Application:

```
def application(environ, start_response):
    start_response('500 Error', [('Content-Type', 'text/html'),
('Secret-Header', 'secret-info')])
    return [b"Secret info, should not be visible!"]
```

This application, when encountering an error, sends an HTTP 500 response along with a custom header `Secret-Header` containing sensitive information.

Nginx Configuration

Nginx can be configured to handle such situations by intercepting errors and hiding sensitive headers.

Example Nginx Configuration:

```
http {
    error_page 500 /html/error.html;
    proxy_intercept_errors on;
```

```
    proxy_hide_header Secret-Header;
}

server {
    listen 80;

    location / {
        proxy_pass http://backend;
    }

    location = /html/error.html {
        internal;
        root /usr/share/nginx/html;
    }
}
```

- `proxy_intercept_errors on;`: This directive ensures that when the backend (uWSGI in this case) returns an HTTP status code greater than 300, Nginx serves a custom error page (`/html/error.html`) instead of the backend's response. This helps to prevent the exposure of internal errors and sensitive data.
- `proxy_hide_header Secret-Header;`: This directive prevents the `Secret-Header` from being forwarded to the client, even if the backend includes it in the response.

Under normal circumstances, this setup works as intended. However, when an invalid HTTP request is sent to the server, Nginx may forward this malformed request directly to the backend without processing it properly. The backend's raw response, including any headers and content, is then sent directly to the client without Nginx's intervention.

Example Invalid HTTP Request:

COPY

```
curl -X GET "http://localhost/%0D%0A" -i
```

- **Valid Request**: If a valid request is made, and the backend returns an error, Nginx intercepts it and serves the custom error page, hiding the `Secret-Header`.
- **Invalid Request**: When an invalid request containing characters like `%0D%0A` (CRLF) is sent, Nginx might not correctly intercept the response. The backend's raw response, including the `Secret-Header`, is sent directly to the client.

Example Output for Invalid Request:

```
HTTP/1.1 500 Error
Content-Type: text/html
Secret-Header: secret-info
Content-Length: 32
Connection: keep-alive

Secret info, should not be visible!
```

Attack Scenario

1. Exploiting Raw Backend Response Exposure

An attacker can exploit this vulnerability by sending a specially crafted HTTP request that Nginx considers invalid or improperly formed.

- **Step 1**: Identify a backend endpoint that might return sensitive information in its headers or body.
- **Step 2**: Send a malformed request to the Nginx server:

```
curl -X GET "http://victim.com/%0D%0A" -i
```

- **Step 3**: If the server is vulnerable, the raw response from the backend, including any sensitive headers like `Secret-Header`, is exposed to the attacker.

Mitigation Strategies

To mitigate the risk of exposing raw backend responses, consider the following strategies:

1. **Strict Request Validation**:
    - Implement strict validation of incoming requests to ensure they adhere to proper HTTP standards. This can be done using Nginx's `if` directive or by setting up custom error handling for malformed requests.
2. **Custom Error Handling for All Scenarios**:
    - Ensure that even in cases of malformed requests, Nginx serves a generic error page instead of forwarding the request to the backend. You can do this by defining error pages for common invalid requests.

```
server {
    listen 80;

    location / {
        if ($request_uri ~* "%0D|%0A") {
            return 400 "Bad Request";
        }
        proxy_pass http://backend;
    }
}
```

3. **Limit Backend Exposure**:
    - Configure the backend to not include sensitive information in its error responses or headers. This reduces the risk even if Nginx does not properly

intercept the response.

4. **Monitoring and Logging**:
    - Monitor and log requests that result in malformed requests or responses. This can help in detecting and responding to potential attacks quickly.

merge_slashes set to off

Nginx, while being a powerful and flexible web server and reverse proxy, can be configured in ways that unintentionally introduce security vulnerabilities. Below, we'll explore several important security considerations, including the `merge_slashes` directive, malicious response headers, the `map` directive, DNS spoofing risks, and the use of the `proxy_pass` and `internal` directives. Each section includes examples, potential attack scenarios, and mitigations.

---

1. `merge_slashes` Directive

By default, Nginx's `merge_slashes` directive is set to `on`. This setting compresses multiple consecutive slashes in a URL into a single slash. While this can streamline URL processing, it can inadvertently hide vulnerabilities, such as Local File Inclusion (LFI) or Path Traversal, especially when Nginx is used as a reverse proxy.

Consider the following URL:

```
http://example.com//etc/passwd
```


With `merge_slashes` enabled, Nginx will compress this to:

```
http://example.com/etc/passwd
```

Attack Scenario:

If an application behind Nginx is vulnerable to LFI, an attacker could exploit this behavior to bypass certain security mechanisms. For instance, if an application allows access to certain files only when there's more than one slash in the URL, `merge_slashes` could allow an attacker to access restricted files.

Mitigation:

Turn off the `merge_slashes` directive in Nginx to ensure that URLs are forwarded to the backend without modification.

```
http {
    merge_slashes off;
}
```

COPY

2. Malicious Response Headers

Certain headers in the HTTP response can alter Nginx's behavior, potentially leading to security vulnerabilities. These headers include:

- `X-Accel-Redirect`: Triggers an internal redirect in Nginx.
- `X-Accel-Buffering`: Controls whether Nginx should buffer the response.
- `X-Accel-Charset`: Sets the charset for the response when using `X-Accel-Redirect`.
- `X-Accel-Expires`: Sets the expiration time for the response.
- `X-Accel-Limit-Rate`: Limits the rate of transfer for responses.

A backend server might send a response header like:

```
X-Accel-Redirect: /.env
```

Attack Scenario:

If the Nginx configuration has `root /`, this header could cause Nginx to internally redirect the request to the `.env` file, exposing sensitive environment variables to the client.

Mitigation:

Ensure that the backend does not send headers like `X-Accel-Redirect` unless explicitly intended. Additionally, carefully review the Nginx configuration to prevent unintended access.

---

3. `map` Directive Default Value

The `map` directive in Nginx is often used to map one value to another, frequently for controlling access or other logic. However, failing to specify a default value in a `map` can lead to unintended behavior.

Example:

```
http {
    map $uri $mappocallow {
        /map-poc/private 0;
        /map-poc/secret 0;
        /map-poc/public 1;
    }
}

server {
```

```
    location /map-poc {
        if ($mappocallow = 0) { return 403; }
        return 200 "Hello. It is a private area: $mappocallow";
    }
}
```

Attack Scenario:

In this configuration, accessing an undefined URI within `/map-poc` (e.g., `/map-poc/undefined`) could bypass the security check because `$mappocallow` would be unset, potentially leading to unauthorized access.

Mitigation:

Always specify a default value in the `map` directive:

COPY

```
http {
    map $uri $mappocallow {
        default 0;
        /map-poc/private 0;
        /map-poc/secret 0;
        /map-poc/public 1;
    }
}
```

4. DNS Spoofing Vulnerability

If Nginx is configured to use an external DNS server, there's a risk of DNS spoofing if an attacker can intercept and manipulate DNS responses.

COPY

```
resolver 8.8.8.8;
```

Attack Scenario:

If an attacker gains control over the DNS server or can spoof DNS responses, they could redirect Nginx to a malicious server.

Mitigation:

To mitigate this risk, configure Nginx to use a trusted, internal DNS server (e.g., `127.0.0.1`):

```
resolver 127.0.0.1;
```

Additionally, ensure DNSSEC is used to validate DNS responses.

---

5. `proxy_pass` and `internal` Directives

The `proxy_pass` directive is used to forward requests to another server, while the `internal` directive ensures that certain locations are only accessible within Nginx.

Example:

```
location /internal/ {
    internal;
    proxy_pass http://backend/internal/;
}
```

Attack Scenario:

If the `internal` directive is not properly configured, it could allow external access to sensitive internal locations, potentially exposing sensitive data or allowing

unauthorized operations.

Mitigation:

Ensure that `internal` locations are correctly configured and not accessible from outside:

```
location /internal/ {
    internal;
    proxy_pass http://backend/internal/;
}

location /public/ {
    proxy_pass http://backend/public/;
}
```

Also, use proper access controls and authentication mechanisms to restrict access to sensitive locations.

Attack Scenario

- **Test** `merge_slashes` behavior:
  `curl "http://example.com//etc/passwd"`
- **Test for malicious headers**:
  `curl -I "http://example.com" -H "X-Accel-Redirect: /.env"`
- **Test** `map` directive for unauthorized access:
  `curl "http://example.com/map-poc/undefined"`
- **Test DNS spoofing (simulation)**:
  - This requires a controlled environment, such as modifying DNS records in a test setup.
- **Test** `internal` directive:

`curl " http://example.com/internal/private-data "`

proxy_set_header Upgrade & Connection

In Nginx, the `proxy_set_header` directive is often used to customize the headers that are sent to a proxied server. However, improper configuration of headers like `Upgrade` and `Connection` can introduce vulnerabilities, such as the h2c (HTTP/2 cleartext) Smuggling attack. This vulnerability can be exploited to establish a direct connection with the backend server, bypassing Nginx's security checks and gaining unauthorized access to protected or internal endpoints.

Vulnerable Configuration:

Consider the following Nginx configuration:

COPY

```
server {
    listen       443 ssl;
    server_name  localhost;

    ssl_certificate       /usr/local/nginx/conf/cert.pem;
    ssl_certificate_key   /usr/local/nginx/conf/privkey.pem;

    location / {
        proxy_pass http://backend:9999;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $http_connection;
    }

    location /flag {
        deny all;
    }
}
```

1. `proxy_set_header Upgrade $http_upgrade;`:
    - This line forwards the `Upgrade` header from the client to the backend server. The `Upgrade` header is typically used in WebSocket connections to switch protocols from HTTP/1.1 to WebSocket or from HTTP/1.1 to HTTP/2.
2. `proxy_set_header Connection $http_connection;`:
    - This line forwards the `Connection` header, which controls whether the network connection stays open after the current transaction finishes. When combined with `Upgrade`, it can lead to the backend server establishing a different type of connection (e.g., HTTP/2).

Vulnerability:

- If the backend server (http://backend:9999) supports HTTP/2 over cleartext (h2c), an attacker can exploit this by sending a specially crafted HTTP request that smuggles HTTP/2 frames into an HTTP/1.1 connection.
- Even if Nginx specifies a path in the `proxy_pass` directive (e.g., `proxy_pass http://backend:9999/socket.io`), the connection will be established with the entire backend server (`http://backend:9999`). This means the attacker can access any other internal path, such as `/flag`, bypassing Nginx's security rules.

Attack Scenario:

1. **Crafted Request**:
    - An attacker sends an HTTP/1.1 request to the Nginx server with the `Upgrade: h2c` header.
    - The Nginx server, due to the configuration, forwards this request to the backend server.
2. **Backend Connection**:
    - The backend interprets the `Upgrade: h2c` header and switches the connection to HTTP/2 cleartext mode.

- The attacker now has direct access to the backend server and can issue additional HTTP/2 requests that bypass Nginx entirely.

3. **Accessing Protected Endpoint**:
   - The attacker can then request protected endpoints, such as `/flag`, which would otherwise be denied by Nginx.

Example of an Attack:

To exploit this vulnerability, an attacker could use a tool like `curl` or a custom script to send the following request:

```
curl -k -v -X GET https://localhost/ -H "Upgrade:
h2c" -H "Connection: Upgrade, HTTP2-Settings"
```

If the backend supports HTTP/2 in cleartext mode, this request could smuggle a connection upgrade to HTTP/2 and allow the attacker to bypass Nginx's path restrictions.

Mitigation:

To protect against this vulnerability, avoid forwarding the `Upgrade` and `Connection` headers unless they are absolutely necessary and you are confident in the backend server's ability to handle them securely.

Secure Configuration:

```
server {
    listen       443 ssl;
    server_name  localhost;

    ssl_certificate      /usr/local/nginx/conf/cert.pem;
```

```
    ssl_certificate_key   /usr/local/nginx/conf/privkey.pem;

    location / {
        proxy_pass http://backend:9999;
        proxy_http_version 1.1;
        # Remove or restrict the following headers unless necessary:
        # proxy_set_header Upgrade $http_upgrade;
        # proxy_set_header Connection $http_connection;
    }

    location /flag {
        deny all;
    }
}
```

In scenarios where WebSockets or HTTP/2 upgrades are necessary, ensure that the backend server is properly secured and does not allow the misuse of these headers to access unauthorized resources.

Additional Attack Scenarios and Commands:

- **Test for h2c Smuggling**:

```
curl -k -v -X GET https://localhost/ -H "Upgrade:
h2c" -H "Connection: Upgrade, HTTP2-Settings"
```

Check for WebSocket upgrade vulnerability:

```
curl -k -v -X GET https://localhost/ -
H "Upgrade: websocket" -H "Connection: Upgrade"
```

Access a protected resource by bypassing Nginx:

```
curl -k -v -X GET https://localhost/flag -H "Upgrade:
h2c" -H "Connection: Upgrade, HTTP2-Settings"
```

COPY

Resources

- https://book.hacktricks.xyz/network-services-pentesting/pentesting-web/nginx#proxy_set_header-upgrade-and-connection

nginx | nginx ingress | Nginx configuration guide | Devops | DevSecOps

Written by

RR **Reza Rashidi**

Add your bio

Published on

DevSecOpsGuides

Add blog description

MORE ARTICLES

RR **Reza Rashidi**

RR **Reza Rashidi**

Attacking OpenStack

Attacking OpenStack, an open-source cloud computing platform, involves exploiting vulnerabilities in...

Attacking CI/CD

In CI/CD (Continuous Integration/Continuous Deployment) environments, several methods and attacks ca...

RR **Reza Rashidi**

Attacking Pipeline

DevOps pipelines, which integrate and automate the processes of software development and IT operatio...