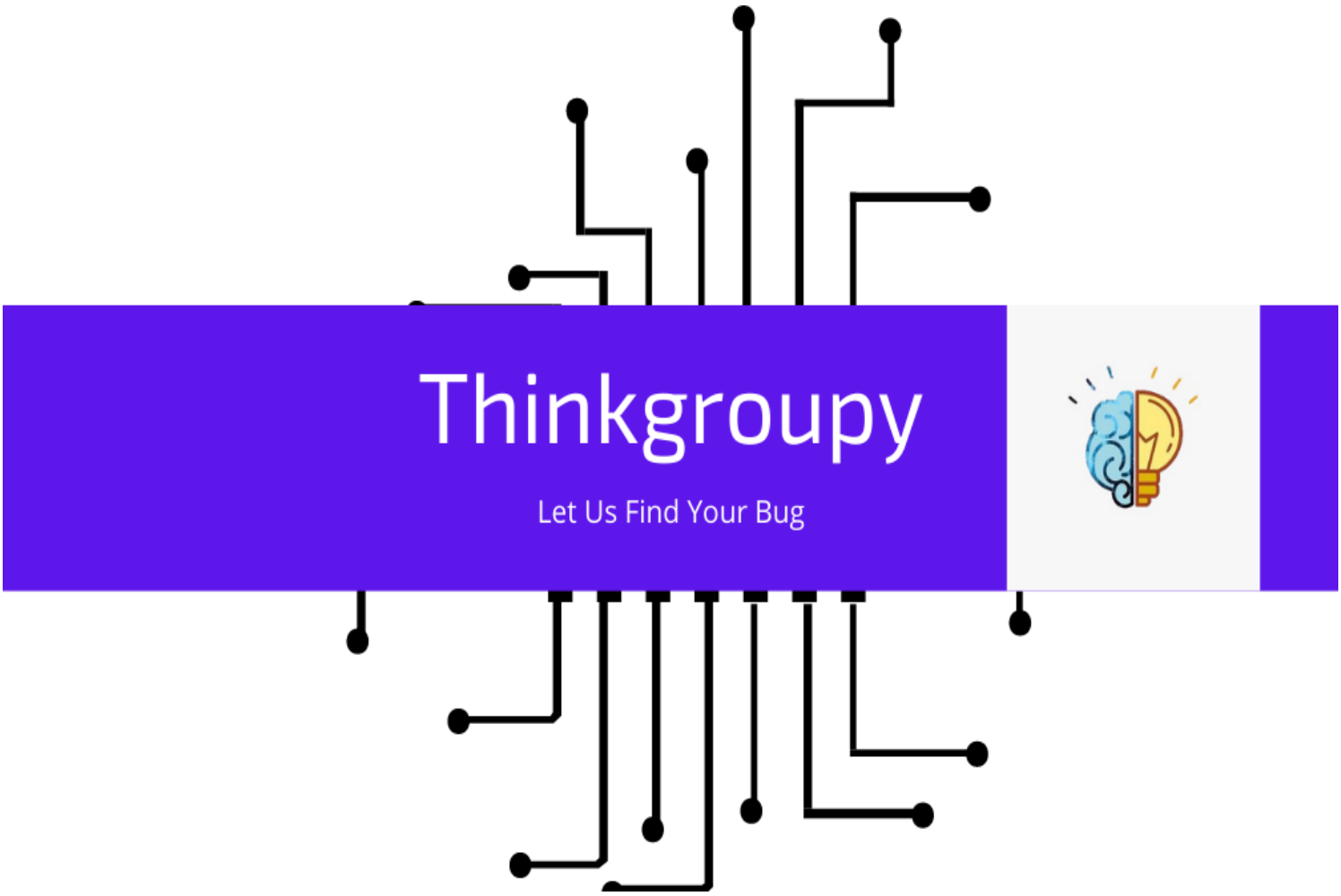

VULNERABILITY REPORT

% target_name % - % target_url %

Report generated on {%time.now%}

Exhaustive Test List (Tools name)

The following pages contain the list of vulnerabilities we tested in this scan, taking into consideration the chosen profile

---

- Reflected cross-site scripting
- Cookie without HTTP Only flag
- Open redirection
- SQL Injection
- Server-side JavaScript injection
- Ruby code injection
- Python code injection
- SQL injection (second order)

Glossary

Term	Definition
Vulnerability	A type of security weakness that might occur in applications (e.g. Broken Authentication, Information Disclosure). Some vulnerabilities take their name not from the weakness itself, but from the attack that exploits it (e.g., SQL Injection, XSS, etc.).
Findings	An instance of a Vulnerability that was found in an application.

Technical Severity

To each finding is attributed a severity which sums up its overall risk

The severity is a compound metric that encompasses the likelihood of the finding being found and exploited by an attacker, the skill required to exploit it, and the impact of such exploitation. A finding that is easy to find, easy to exploit and the exploitation has high impact, will have a greater severity.

Different findings of the same type could have a different severity: we consider multiple factors to increase or decrease it, such as if the application has an authenticated area or not.

The following table describes the different severities:

Severity	Description	Examples
HIGH	These findings may have a direct impact in the application security, either clients or service owners, for instance by granting the attacker access to sensitive information.	SQL Injection OS Command Injection
MEDIUM	Medium findings usually don't have immediate impact alone, but combined with other findings may lead to a successful compromise of the application.	Cross-site Request Forgery Unencrypted Communications
LOW	Findings where either the exploit is not trivial, the impact is low, or the finding cannot be exploited by itself.	Directory Listing Clickjacking
INFO	Findings where either the exploit is not trivial, the impact is info, orthe finding cannot be exploited by itself.	Open Port, etc.

Detailed Finding Descriptions

This section contains the findings in more detail, ordered by severity

n % finding_name % INFO

Vulnerable Url: %vul_url%

Description: %description%

Category Descriptions

The following pages contain descriptions of each vulnerability. For each vulnerability you will find a section explaining its impact, causes and prevention methods.
These descriptions are very generic, and whenever they are not enough to understand or fix a given finding, more information is provided for that finding in the Detailed Finding Descriptions section.

Referrer policy not defined

Description

The application does not prevent browsers from sending sensitive information to third party sites in the **referer** header.

Without a referrer policy, every time a user clicks a link that takes him to another origin (domain), the browser will add a **referer** header with the URL from which he is coming from. That URL may contain sensitive information, such as password recovery tokens or personal information, and it will be visible that other origin. For instance, if the user is at `example.com/password_recovery?unique_token=14f748d89d` and clicks a link to `example-analytics.com`, that origin will receive the complete password recovery URL in the headers and might be able to set the users password. The same happens for requests made automatically by the application, such as XHR ones.

Applications should set a secure referrer policy that prevents sensitive data from being sent to third party sites.

Fix

This problem can be fixed by sending the header **Referrer-Policy** with a secure and valid value. There are different values available, but not all are considered secure. Please note that this header only supports one directive at a time. The following list explains each one and it is ordered from the safest to the least safe:

- `no-referrer`: never send the header.
- `same-origin`: send the full URL to requests to the same origin (exact scheme + domain)
- `strict-origin`: send only the domain part of the URL, but sends nothing when downgrading to HTTP.
- `origin`: similar to **strict-origin** without downgrade restriction.
- `strict-origin-when-cross-origin`: send full URL within the same origin, but only the domain part when sending to another origin. It sends nothing when downgrading to HTTP.
- `origin-when-cross-origin`: similar to **strict-origin-when-cross-origin** without the downgrade restriction.

Insecure options: * `no-referrer-when-downgrade`: sends the full URL when the scheme does not change. It will send if both origins are, for instance, HTTP. * `unsafe-url`: always sent the full URL

A possible, safe option is `strict-origin`, so the header would look like this:

```
Referrer-Policy: strict-origin
```

It is normally easy to enable the header in the web server configuration file, but it can also be done at the application level.

Please note that the referrer header is written `referer`, with a single `r` but the referrer policy header is properly written, with `rr`: `Referrer-Policy`.

HSTS header not enforced

Description

The application does not force users to connect over an encrypted channel, i.e. over HTTPS. If the user types the site address in the browser without starting with *https*, it will connect to it over an insecure channel, even if there is a redirect to HTTPS later. Even if the user types *https*, there may be links to the site in HTTP, forcing the user to navigate insecurely. An attacker that is able to intercept traffic between the victim and the site or spoof the site's address can prevent the user from ever connecting to it over an encrypted channel. This way, the attacker is able to eavesdrop all communications between the victim and the server, including the victim's credentials, session cookie and other sensitive information.

Fix

The application should instruct web browsers to only access the application using HTTPS. To do this, enable HTTP Strict Transport Security (HSTS).

You can do so by sending the `Strict-Transport-Security` header so that browsers will always enforce a secure connection to your site, regardless of the user typing *https* in the address.

An HSTS enabled server includes the following header in an HTTPS response: `Strict-Transport-Security: max-age=15768000;includeSubdomains` Please bear in mind that only HTTPS responses should have the HSTS header, because browsers ignore this

header when sent over HTTP.

When the browser sees this, it will remember, for the given number of seconds, that the current domain should only be contacted over HTTPS. In the future, if the user types *http://* or omits the scheme, HTTPS is the default. In this example, which includes the option `includeSubdomains`, all requests to URLs in the current domain and subdomains will go over HTTPS. When you set `includeSubdomains` make sure you can serve all requests over HTTPS! It is, however, important that you add the option `includeSubdomains` whenever is possible.

Instead of changing your application, you should have the web server setting the header for you. If you are using Apache, just enable `mod_headers` and add the following line to your virtual host configuration: `Header always set Strict-Transport-Security "max-age=15768000;includeSubdomains"`

If you are using NGINX, just add this line to your host configuration: `add_header Strict-Transport-Security max-age=15768000;includeSubdomains`

Note that because HSTS is a "trust on first use" (TOFU) protocol, a user who has never accessed the application will never have seen the HSTS header, and may therefore be vulnerable to aforementioned SSL stripping attacks. To mitigate this risk, you can optionally ask the browser vendors to include your domain in a preloaded list, included in the browser, and afterwards add the 'preload' flag to the HSTS header.

Browser content sniffing allowed

Description

The application allows browsers to try to mime-sniff the content-type of the responses. This means the browser may try to guess the content-type by looking at the response content, and render it in way it was not intended to. This behavior may lead to the execution of malicious code, for instance, to explore an XSS vulnerability.

Applications should disable this behavior, forcing browsers to honor the content-type specified in the response. Without a specific content-type set browsers will default to render the content as text, turning XSS payloads innocuous.

Disabling mime-sniffing should be seen as an extra layer of defense against XSS, and not as replacement of the recommended XSS prevention techniques.

Fix

This problem can be fixed by sending the header **X-Content-Type-Options** with value **nosniff**, to force browsers to disable the content-type guessing (the sniffing).

The header should look this:

```
X-Content-Type-Options: nosniff
```

It is normally easy to enable the header in the web server configuration file, but it can also be done at application level.