A5-Security Misconfiguration

Exploitability: EASY | Prevalence: COMMON | Detectability: EASY | Technical Impact: MODERATE

Description

This vulnerability allows an attacker to accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.

Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, framework, and custom code.

Developers and system administrators need to work together to ensure that the entire stack is configured properly.

Attack Mechanics

This vulnerability encompasses a broad category of attacks, but here are some ways attacker can exploit it:

1. If application server is configured to run as root, an attacker can run malicious scripts (by exploiting eval family functions) or start new child processes on server
2. Read, write, delete files on file system. Create and run binary files
3. If the server is misconfigured to leak internal implementation details via cookie names or HTTP response headers, then attacker can use this information towards building site's risk profile and finding vulnerabilities
4. If request body size is not limited, an attacker can upload large size of input payload, causing server to run out of memory, or make processor and event loop busy.

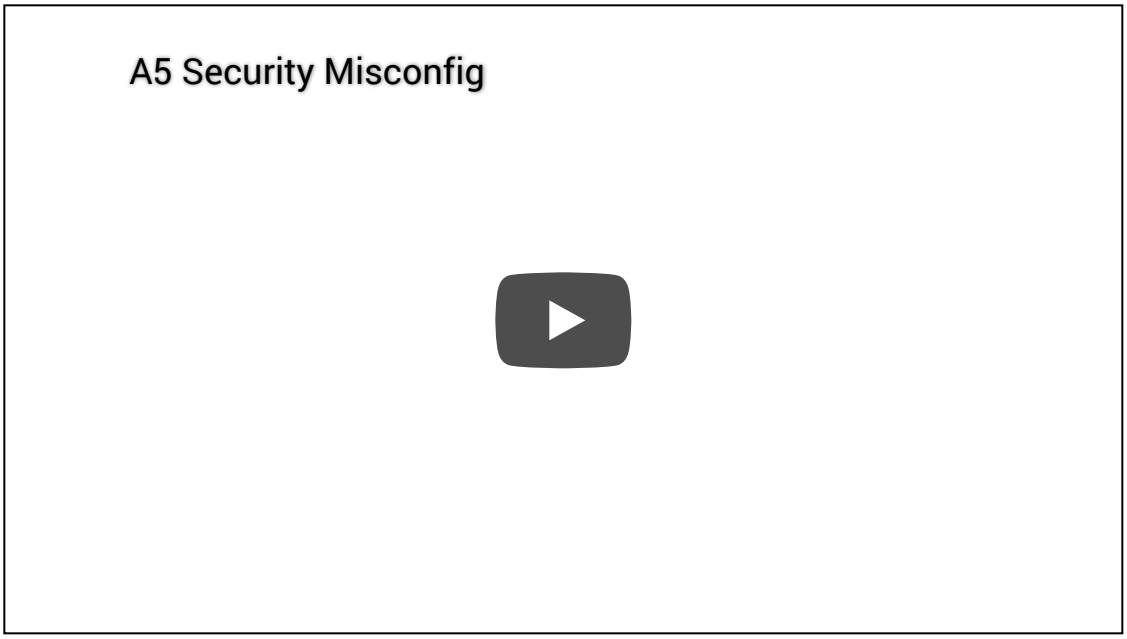
How Do I Prevent It?

Here are some node.js and express specific configuration measures:

- Use latest stable version of node.js and express (or other web framework you are using). Keep a watch on published vulnerabilities of these. The vulnerabilities for node.js and express.js can be found here (http://blog.nodejs.org/vulnerability/) and here (http://expressjs.com/advanced/security-updates.html), respectively.
- Do not run application with root privileges. It may seem necessary to run as root user to access privileged ports such as 80. However, this can achieved either by starting server as root and then downgrading the non-privileged user after listening on port 80 is established, or using a separate proxy, or using port mapping.

- Review default in HTTP Response headers to prevent internal implementation disclosure.
- Use generic session cookie names
- Limit HTTP Request Body size by setting sensible size limits on each content type specific middleware (`urlencoded, json, multipart`) instead of using aggregate `limit` middleware. Include only required middleware. For example if application doesn't need to support file uploads, do not include multipart middleware.
- If using multipart middleware, have a strategy to clean up temporary files generated by it. These files are not garbage collected by default, and an attacker can fill disk with such temporary files
- Vet npm packages used by the application
- Lock versions of all npm packages used, for example using shrinkwarp (https://www.npmjs.org/doc/cli/npm-shrinkwrap.html), to have full control over when to install a new version of the package.
- Set security specific HTTP headers

Source Code Example



The default HTTP header x-powered-by can reveal implementation details to an attacker. It can be taken out by including this code in `server.js`

```
app.disable("x-powered-by");
```

The default session cookie name for express sessions can be changed by setting key attribute while creating express session.

```
        app.use(express.session({
            secret: config.cookieSecret,
            key: "sessionId",
            cookie: {
                httpOnly: true,
                secure: true
            }
        }));
```

The security related HTTP Headers can be added using helmet middleware as below

```
        // Prevent opening page in frame or iframe to protect from clickjacking
        app.disable("x-powered-by");

        // Prevent opening page in frame or iframe to protect from clickjacking
        app.use(helmet.xframe());

        // Prevents browser from caching and storing page
        app.use(helmet.noCache());

        // Allow loading resources only from white-listed domains
        app.use(helmet.csp());

        // Allow communication only on HTTPS
        app.use(helmet.hsts());

        // Forces browser to only use the Content-Type set in the response header instead
of sniffing or guessing it
        app.use(nosniff());
```