BSI IT Grundschutz (with Guile)

These are English summaries for the collection of best practices and requirements from the German federal agency of security in information technology (BSI): IT-Grundschutz, specifically the Kompendium 2023.

There are older official documents in English, but they don't map exactly to the newest German ones. Closest is the Compendium_2022.pdf.[1]

Summary:

- Always expect requirements of Information Security Management Systems (ISMS).
- Most relevant are BSI GS **CON.8 and APP3.1**. Depending on the project, there can be more. *Must be part of the tender.*

This article Guile Scheme for practical examples. First it shows the requirements, then concrete measures in the following style:

LABEL: Title

Requirements

- Measure

For some requirements, a solution with GNU Artanis is included. Many thanks to Nala Ginrut for those solutions!

This is a **Work in Progress** (WIP). The progress markers mean: RESEARCH (need so search for a good solution), IMPLEMENT (solution known, need to write example code), WRITE (tools exist, need to write it down). The current status is shown in 3.

I only discuss CON (Concepts) and APP (Applications). There are others (e.g. for OPS), but I'm a software developer and these two are what I can judge.

[1]Why would you follow these rules? Firstoff: if you want a government contract in Germany, you likely have to. Also they are pretty well-written: most of these are just common sense when you want to create reliable software. So much that if someone does not want to follow them, you can assume that using their software might put you at risk. These rules need to be enforced, because they can conflict with budget-pressure.

Contents

1 BSI GS block CON.8

1.1 CON.8.A2: Selection of a Process Model

- Choose a model how to work, for example scrum.
- Security requirements have to be integrated into that model.
- You MUST follow that model.

- Clarify the model in the tender, if possible. Describe in which step security requirements are checked.

1.2 CON.8.A3: Selection of a Development Environment

- Choose a development environment based on selection criteria.

- Document the decisions from the other sections.
- Use linters defined by language for code quality. Defined libraries in guix.scm
- Define which browsers and OS'es and DBs to support.
- To have a consistent environment between tooling,

Don't have passwords or variables in the git or Mercurial repository that hint on the DEV, TEST, PROD environment! Define them in a separate repository if needed.

1.2.1 RESEARCH Create linters for Guile

- Do the compiler messages suffice?

1.3 CON.8.A5: Secure System Design

- validate all inputs
- … on the server
- default settings must provide secure operation
- errors or crashes must not expose information that should be protected
- software must run with least possible privileges
- data that should be protected must be encrypted during transmission (⇒ https) and storage.
- authenticate users securely
- authentication passwords must be stored with a secure hashing scheme
- log security relevant events
- do not ship code/config with comments to the customer that could contain secrets.

Document the system design.
You MUST validate that security requirements are fulfilled.

- Use https by default.
- keycloak (or similar external service) protects passwords.
- log as usual, do not activate debug log level by default.
- ship bytecode/binaries/jars, not source code.
- use safe defaults
  - e.g. only bind to `127.0.0.1:8080`, not `0.0.0.0` — enable remote access *only* via SSL terminator

1.3.1 Use single-sign-on / OAUTH / JWT

Use either guile-oauth or guile-jwt while a system like keycloak handles usernames and passwords.

Alternatively use an oauth plugin to nginx or another SSL terminator.

1.3.2 Use nginx as SSL terminator

Terminate the SSL connection for two servers running on port 9000 and 9001. Also serve the static files `style.css` `favicon.ico` `robots.txt` directly via nginx.

Write the following into `/etc/nginx/sites-enabled/default`:

```
events { }
http {
    # setup for logging
    include /etc/nginx/conf.d/anonymized.conf;
    # define the two servers
    upstream sub {
        ip_hash;
        server localhost:9000;
        server localhost:9001;
    }

    server {
        server_name sub.example.com;
        # static files
        location ~ ^/(style.css|favicon.ico|robots.txt)$ {
          root /var/www/html;
        }
        # Guile server
        location / {
            proxy_pass http://sub;
        }
        listen 80 default_server;
        listen [::]:80 default_server;

        # dsgvo allowed logging
        access_log /var/log/nginx/sub-access.log anonymized;
    }
}
```

On Debian and some other distributions, there is a main config in `/etc/nginx/nginx.conf` which includes the files from `/etc/nginx/sites-enabled/default`. In these you must leave out `events { }` and the `include` and only add the body of the `http` block.

The anonymized IP filter requires a file `/etc/nginx/conf.d/anonymized.conf`:

```
map $remote_addr $ip_anonym1 {
default 0.0.0;
"~(?P<ip>(\d+)\.(\d+))\.(\d+)\.\d+" $ip;
"~(?P<ip>[^:]+:[^:]+):" $ip;
}

map $remote_addr $ip_anonym2 {
default .0.0;
"~(?P<ip>(\d+)\.(\d+)\.(\d+))\.\d+" .0.0;
"~(?P<ip>[^:]+:[^:]+):" ::;
}

map $ip_anonym1$ip_anonym2 $ip_anonymized {
default 0.0.0.0;
"~(?P<ip>.*)" $ip;
}

log_format anonymized '$ip_anonymized - $remote_user [$time_local] '
'"$request" $status $body_bytes_sent '
'"$http_referer" "$http_user_agent"';
```

Then setup Let's Encrypt secured SSL termination via certbot:

```
sudo certbot --nginx
```

To test it locally, run

```
mkdir -p /var/log/nginx # for testing you can use /tmp here and in the
                        # config to avoid write access problems
nginx -p /etc/nginx -c sites-enabled/default
```

1.3.3 RESEARCH Ship only bytecode with Guile

To ensure that, you need an autotools setup to compile the files locally and then create a local runner that adjusts the load path.

1.4 CON.8.A6: Use of External Libraries from Trusted Sources

Use libraries from trustworthy sources. Check their integrity.

- Check hashes or signatures of libraries. This happens by default when you get dependencies via guix shell in a `guix.scm`.

1.5 CON.8.A7: Conducting Software Tests During Development

Use tests during development and check for coding errors continuously.
- test the requirements
- test failure states
- test all APIs we document
- check edge cases of input values (i.e. check for division by zero)
- don't hack on production systems!

- Build your package with Continuous Integration (CI) for centralized tests.
- Have Unit Tests, Integration Tests, code reviews.
  - Add simple unit test defintion like guile-doctests. Use SRFI-64 for larger testsuites.
  - Review changes in merge requests to catch problems that can't be caught in automated tests. For example in Forgejo (or Github) or Heptapod pull/merge requests.
  - Define and document `checklists` for regular manual checks of areas that are relevant to the **security requirements**.
- Don't hack on production.

1.6 CON.8.A8: Provision of Patches, Updates, and Changes

You MUST guarantee that security critical patches and updates are provided quickly; also when libraries have security updates. Provide checksums or signatures for release files.

- Automate creating releases. Maybe get `make distcheck` working.
- At the very least integrate something like `sha256sum "${file}" > "${file}".sha256`
- Setup your package as channel to ensure all devs have the same packages via `guix shell`. You can use an inferior channel to pin packages to specific commits.
- Automate updating libraries. Maybe adapt guix refresh for your dependency definition in `guix.scm`.
- Create a CycloneDX SBOM, then setup OWASP Dependency-Track or Mend Renovate.

1.6.1 IMPLEMENT Create SBOM from Guix

- Adjust guix graph: add `--type=sbom`

1.7 CON.8.A10: Source Code Version Management

- Use a version tracking system. Create backups of it.

- Version tracking with Git or Mercurial.
- Have tags and release branches as defined for the project in 1.1.
- Have off-site backups of the source repository.

1.8 CON.8.A20: Checking External Components

- check libraries. Either of:
  - use established checks, or
  - check for vulnerabilities ourselves
- Check all external components with checksums or certificates (signatures).
- Avoid old versions of libraries.
- Prove sufficient checking.

- Check hashes of libraries. Happens automatically when using guix shell; see 1.4.
- Maybe use Renovate in ci/cd.

2 BSI GS block APP.3

2.1 APP.3.1.A1: Clients must authenticate

- auth-retries must be limited

2.1.1 When using keycloak and oauth/jwt, configure keycloak for limited retries

Enable Brute Force Detection for Max Login Failures, locking the account for at least a few minutes.

Maybe also see the option `http-max-queued-requests` in Configuring Keycloak for production.

2.1.2 IMPLEMENT create helpers for authorized handler definition

```
;; default handler definition
define-authorized method : handler request body
  ;; ...
  values ...
;; unauthorized
define-unauthorized method : handler request body
  ;; ...
  values ...
```

2.2 APP.3.1.A4: Uploads must be limited as much as possible

If Uploads they are allowed:

- limit to permitted clients
- use restrictive access rights
- ensure that files are only saved in allowed places

2.2.1 RESEARCH define an upload helper with configurable upload limits

- always require authorization for uploads
- have configurable upload locations

2.2.2 Artanis: configurable upload control **Artanis**

- configurable using the artanis.conf template
- For details see (upload . . .) definition in the code

2.3 APP.3.1.A7: Prevent unauthorized automated use.

- Allow intended automated use like RSS feeds.

- Ensure that any other use requires a manual login.

2.3.1 IMPLEMENT make authorized access the default

Use explizit unauthorized handler from 2.1. Make unauthorized less convenient than authorized to avoid nudging people to unsafe patterns.

2.3.2 RESEARCH provide rate limits configuration that rate limits by default

(to be written)

2.4 APP.3.1.A14: Protect confidential data

- MUST use serverside crypto to prevent unauthorized access.
- MUST use salted hash for passwords.
- MUST prevent forbidden access to source files.

- use trustworthy server-side salted hash method for passwords
- Javascript files are sources: minimize which Javascript files are accessible without login
  - in addition to 1.3: not having sources in production
- login forms should import only the Javascript they really need

2.4.1 RESEARCH Use salted hash for password login

(to be written)

2.4.2 Alternatively avoid handling passwords and use oauth or jwt for authorization

See 1.3.

2.4.3 RESEARCH Provide static files in authorized and un-authorized flavor

2.4.4 Artanis: integrate oauth or jwt or use the #:auth handler Artanis

For oauth or jwt, see 1.3. For the auth-handler see the Authentication Section of the Artanis manual.

3 Still to be done

RESEARCH	IMPLEMENT	WRITE
Create linters for	Create SBOM from Gu	
Ship only bytecode	create helpers for	
define an upload he	make authorized acc	
provide rate limits		
Use salted hash for		
Provide static file		

List of Links