Masaryk University
Faculty of Informatics

Process for approval of third party libraries

Master's Thesis

Bc. Vít Šesták

Brno, Fall 2016

Masaryk University
Faculty of Informatics

Process for approval of third party libraries

Master's Thesis

Bc. Vít Šesták

Brno, Fall 2016

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Vít Šesták

Advisor: RNDr. Andriy Stetsko, Ph.D.

Abstract

Even if a proper care is taken for security of first-party code, vulnerabilities might come from third-party libraries. These libraries might be of various quality. This thesis aims to compare existing approaches for third-party libraries security and propose an approval process and suggest suitable tools for it.

Keywords

security, library, third party code, approval process, security update, known vulnerabilities

Contents

List of Figures

Abbreviations

abbrevia	tion description
AST	abstract syntax tree
BC break	backward compatibility break
CSRF	cross-site request forgery
DOM	Document Object Model
GAV	Group id, Artifact id, Version
GA	Group id, Artifact id
NVD	National Vulnerability Database
ODC	OWASP Dependency Check
OSVDB	Open-Sourced Vulnerability Database
RCE	remote code execution
XSS	cross-site scripting

1 Existing approaches

Existing approaches might either focus directly on approval of the library, or they might try to improve security of third-party code in some alternative way. In some cases, an approach that does not focus directly on approval of library might influence approval process, because libraries that fits to the approach would be preferred to other libraries.

1.1 Avoiding abandoned libraries

Multiple sources suggest using only actively maintained libraries, because those will (hopefully) get security updates. [1] [2] Hopefully, they will also be updated to keep up with current security best-practices. While actively maintained library is by no means a guarantee of those properties, an abandoned library is virtually a guarantee of the opposite.

1.2 Keeping libraries up-to-date

Even if a library is actively maintained with security in mind, there is no benefit for the project unless someone applies the updates. Reasoning for up-to-date libraries is thus similar to avoiding abandoned libraries. [1]

Requirement for keeping libraries up-to-date has some deeper impact on the whole process, see section 1.3 for impact on approval process and section 2.4 for impact on other parts of the life-cycle.

1.3 Ease of updates

Even if there is an update and we want to apply it, there might be another obstacles like backward incompatibility. Ensuring that the update will be easy is thus also important aspect of choosing a library. [2]

1.4 Using facades

Library can be used either directly, or through a facade [3]. There are some reasons for using facades:

a. Library usage is documented by the facade. [1]
b. Moving from one library to another is made easier by the facade. [1]
c. They might enforce some best practices, especially in areas with many common misconceptions or common mistakes. [4] [5] [6]
d. Enforce secure defaults. (See section 2.10.)

While [1] promotes facades strongly, it might be also controversial in some cases. Creating a facade in a limited time or with limited knowledge (or with both) might do rather some harm, e.g. it might enforce bad practices. It might be even worse with wrongly designed API of a facade, as the harm done by bad facade will "infect" all code it uses.

1.5 Using a security manager for libraries

This solution is specifically written for Java (though it might be also applied to some other platforms) and it is suggested as the most cost effective solution. [7]

Although there are many sandbox vulnerabilities [8] [9] [10] [11] [12] and one might consider Java sandbox as potentially vulnerable, this solution might be useful for two reasons.

First, if the application does not execute untrusted bytecode in the JVM sandbox, then many vulnerabilities in the library (e.g. path traversals, XML External Entity vulnerabilities, etc.) are unlikely to give the attacker enough power to exploit the JVM sandbox vulnerability. So, the sandbox effectively limits the attack surface.

Second, even if attacker is able to perform remote bytecode execution, they might be unable to find a suitable exploit for the sandbox escape.

A conceptually similar approach is usage of microservices that are separated on OS level (e.g. processes running as different user)

or on hypervisor level. The hypervisor approach is said to allow the strongest isolation.[13].

This might be a feasible way in a new project. However, it might be much harder in an existing project, as it might require some redesign.

1.6 "Owners" of libraries responsible for updates

This is an organizational advice that is used in Chromium development. [14] In this case, the responsibility is distributed under multiple people, called "owners".

The Chromium approach might be contrasted to having a dedicated security team for this purpose. Debate on advantages and disadvantages of these approaches is rather a debate on horizontal vs. vertical teams [15] focused on security. While security is not a separate module (and thus cannot be solely handled by a separate team), some security aspects might be better handled by a separate security team.

2 The approval process

This approval process is developed with configurability in mind. That is, it should be applicable for both limited security budget scenario and high security requirements scenario. Moreover, categorization (see section 2.2) allows focusing on high-impact parts of system without spending too much effort and money on low-impact parts.

2.1 Developer actions when they want to use a library

Developer has to use approved libraries only. In order to know what libraries are approved, there will be some database of libraries that user can look into. Developers will be able to find:

- exact identifier of a library (e.g. groupId:artifactId:version for Maven artifacts)
- approval or non-approval reasons
- restrictions applied to a library in this database and some explanation (in order to encourage developers to adhere those restrictions).

If the library has not been approved (i.e. it is unknown if it is OK to use the library in the version for that purpose), the developer will ask the security team to approve that library. The security team will perform either full review (see section 2.12), or new version review (see section 2.4) if some review has been done for some older version of the library. Details are shown on figure 2.2.

2.2 Library categorization

Different libraries do require different care. For example, all C/C++ libraries might potentially get the memory corrupted. This is not usually a concern in Java libraries (unless one uses either native code[16] or Unsafe API[17]). Another example might be cryptographic libraries, that are hard to implement correctly, hard to use correctly and likely security critical. As a result, cryptographic libraries require some special

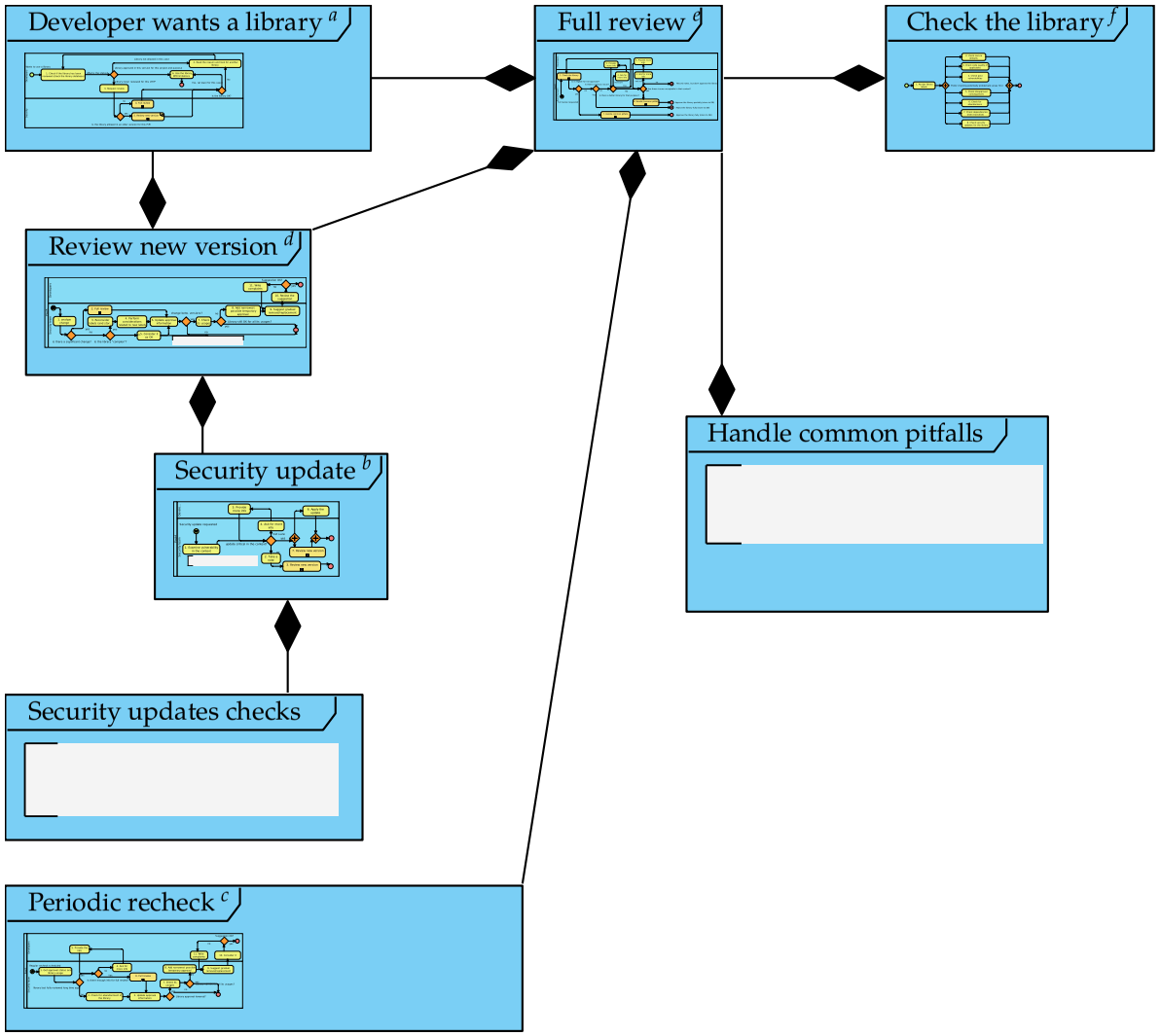

Figure 2.1: Overview

a. Specified in figure 2.2
b. Specified in figure 2.7
c. Specified in figure 2.5
d. Specified in figure 2.6
e. Specified in figure 2.3
f. Specified in figure 2.4

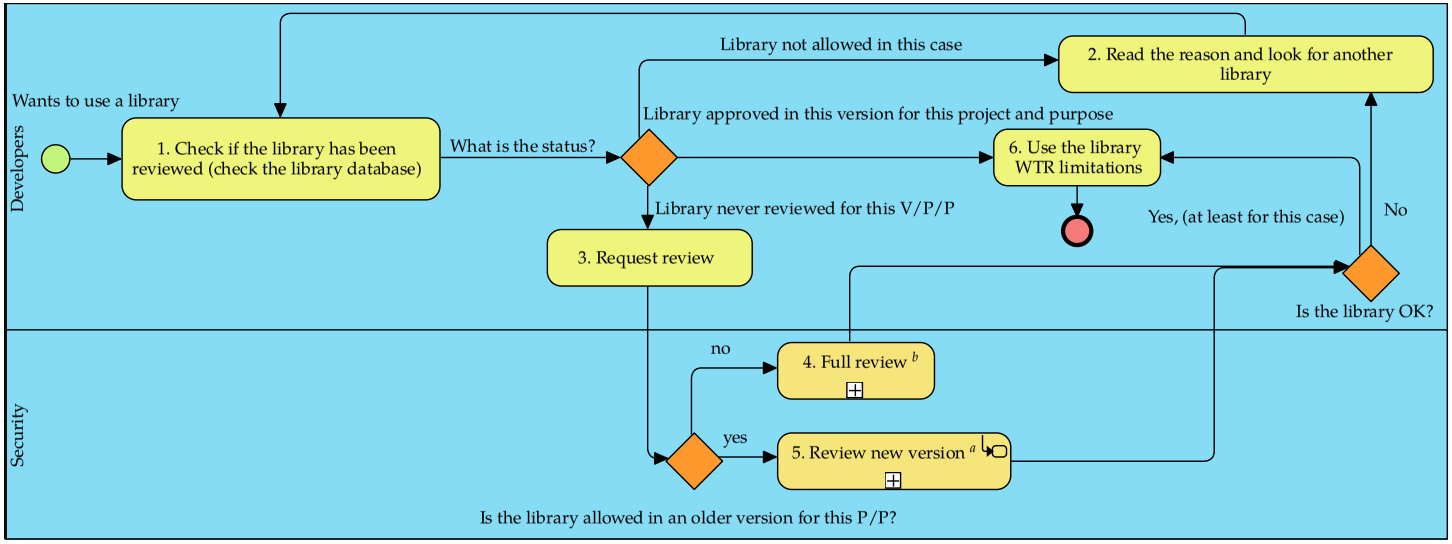

Figure 2.2: Developer wants to use a library (see section 2.1)

a. See figure 2.6
b. See figure 2.3

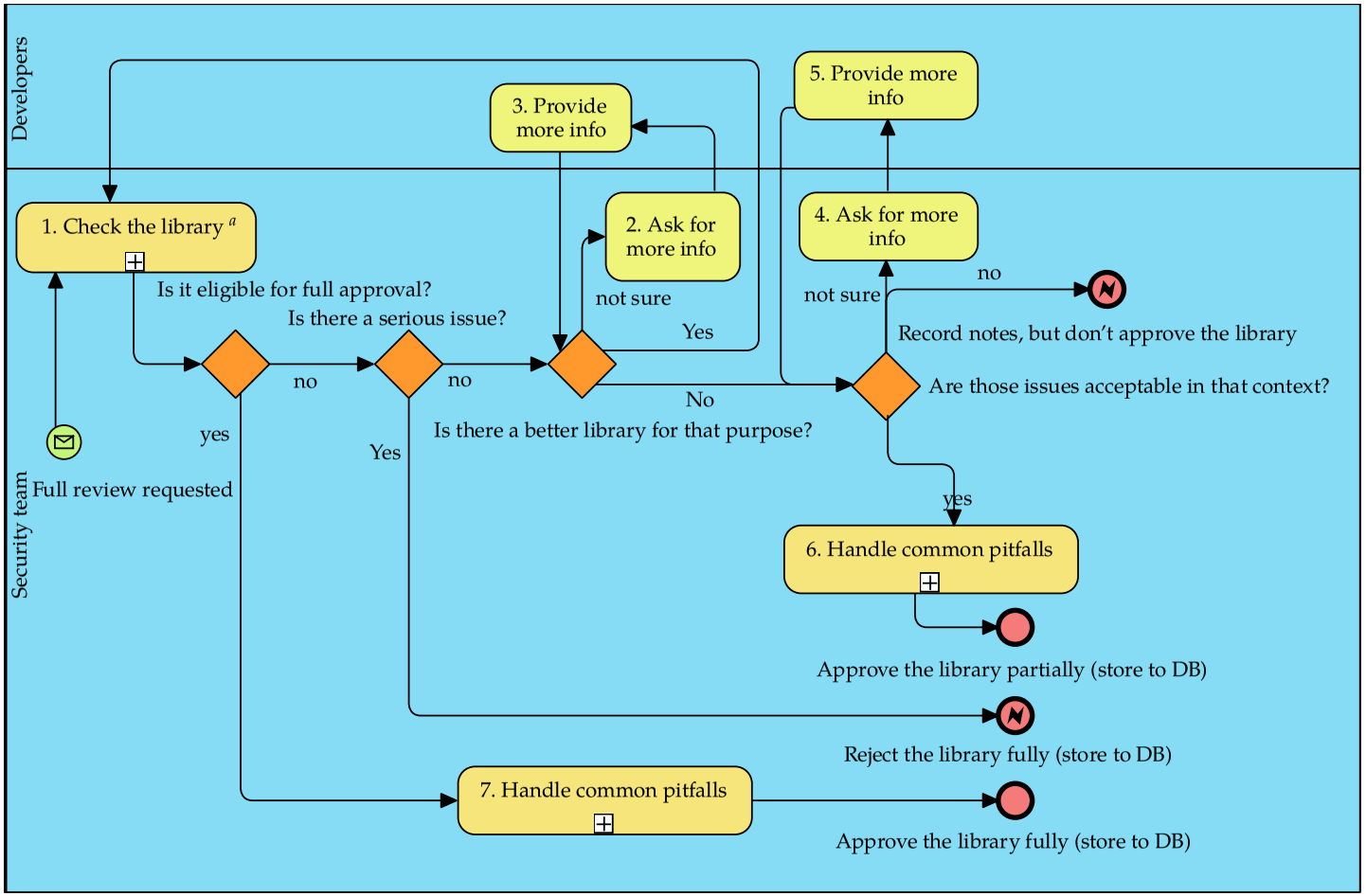

Figure 2.3: Full review (see section 2.12)

a. See figure 2.4

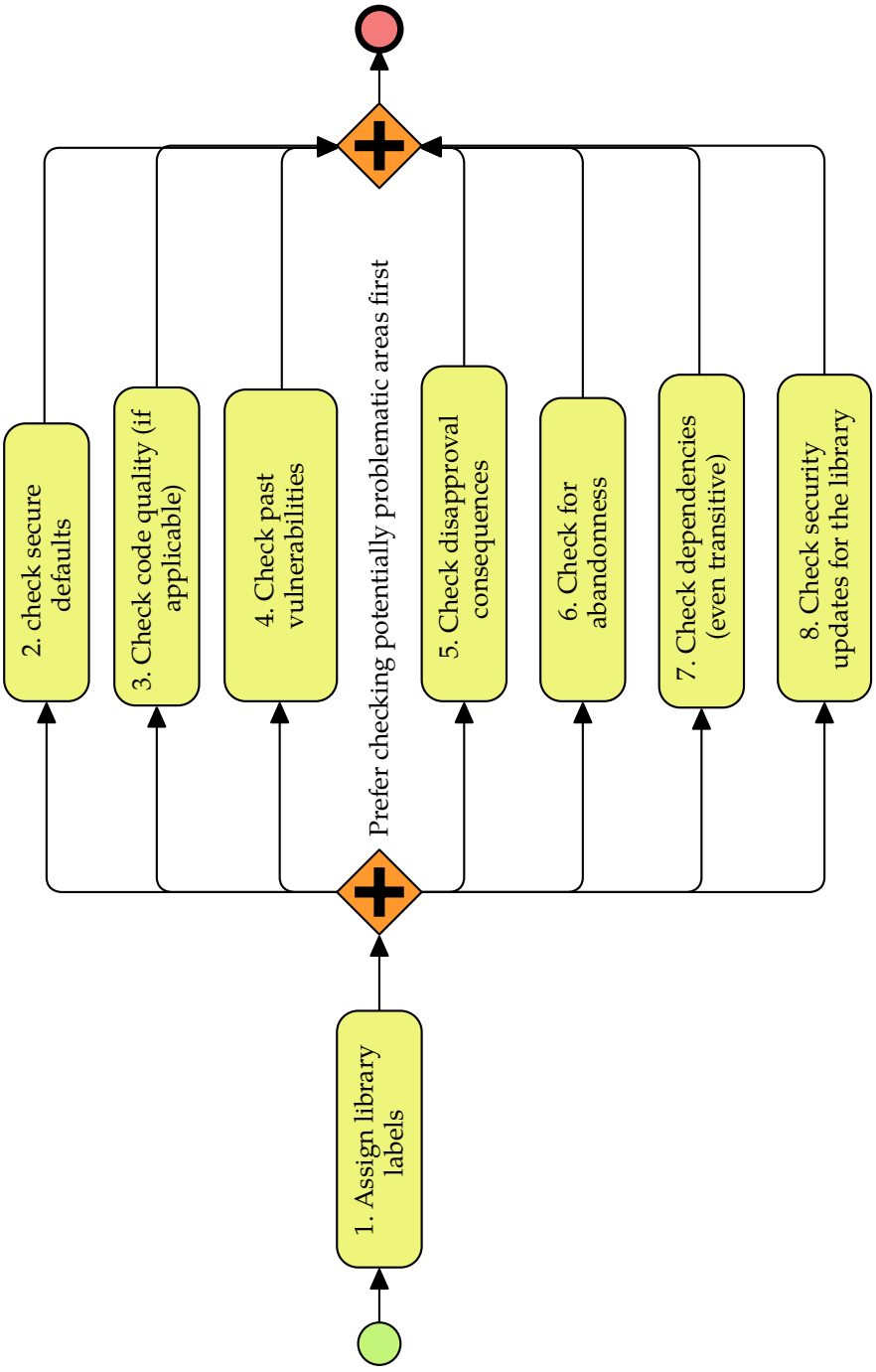

Figure 2.4: Check the library

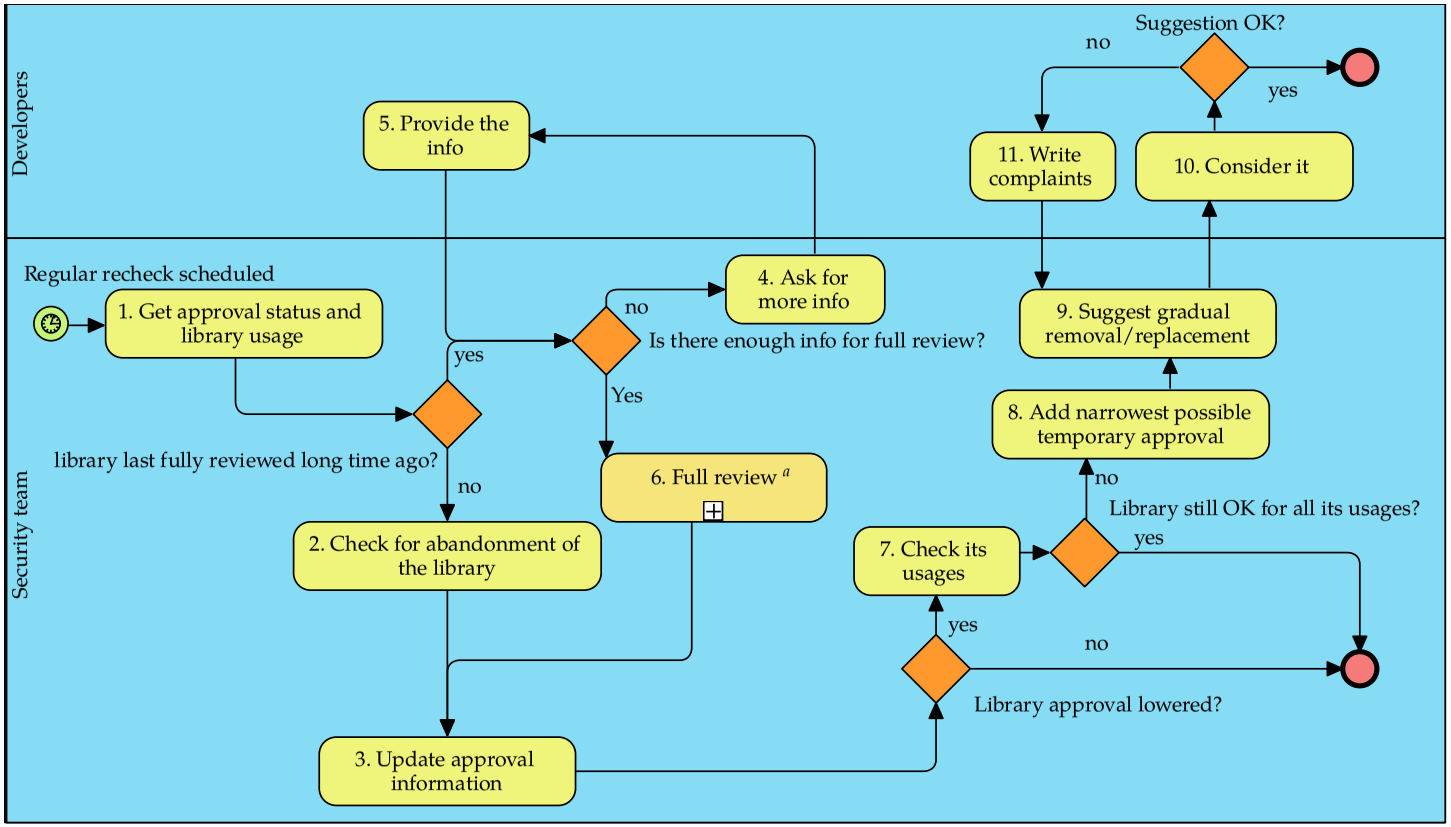

Figure 2.5: Periodic recheck (see section 2.5)

a. See figure 2.3

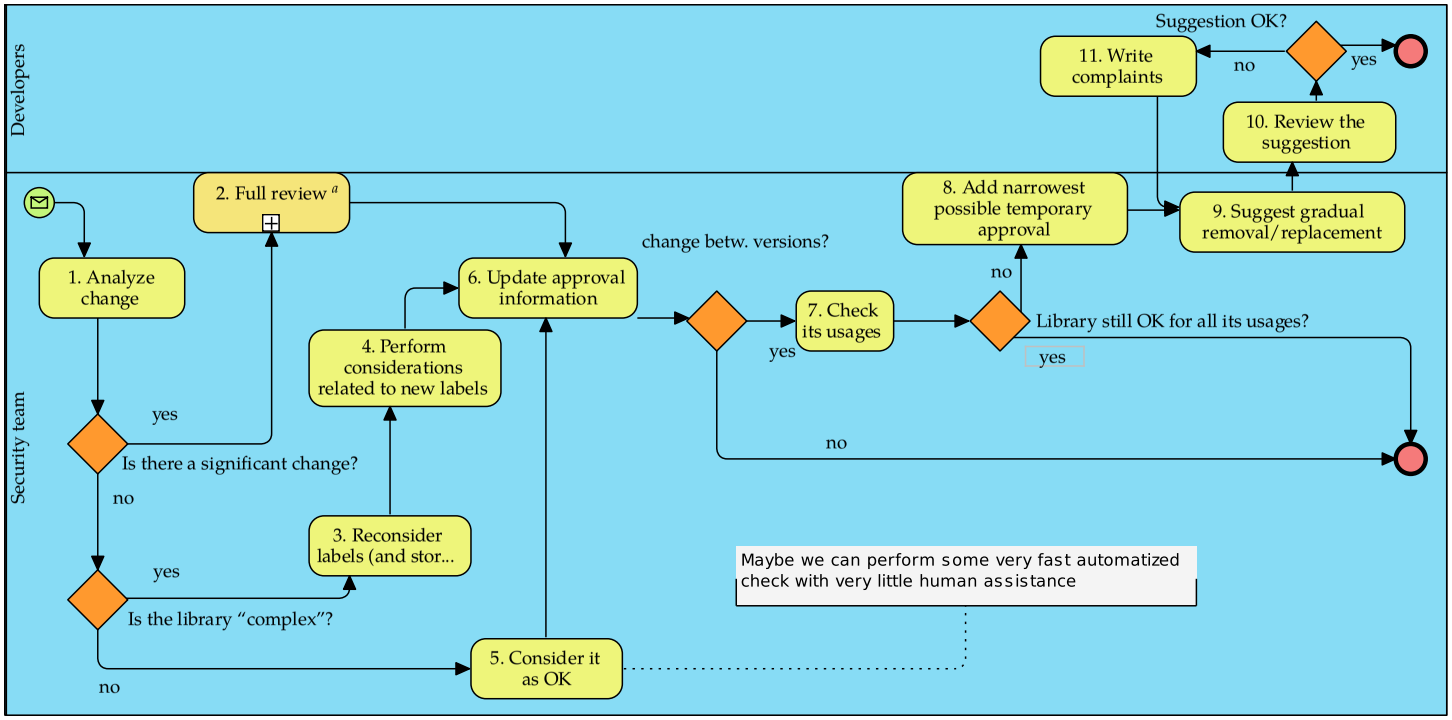

Figure 2.6: Review new version (see section 2.4)

a. See figure 2.3

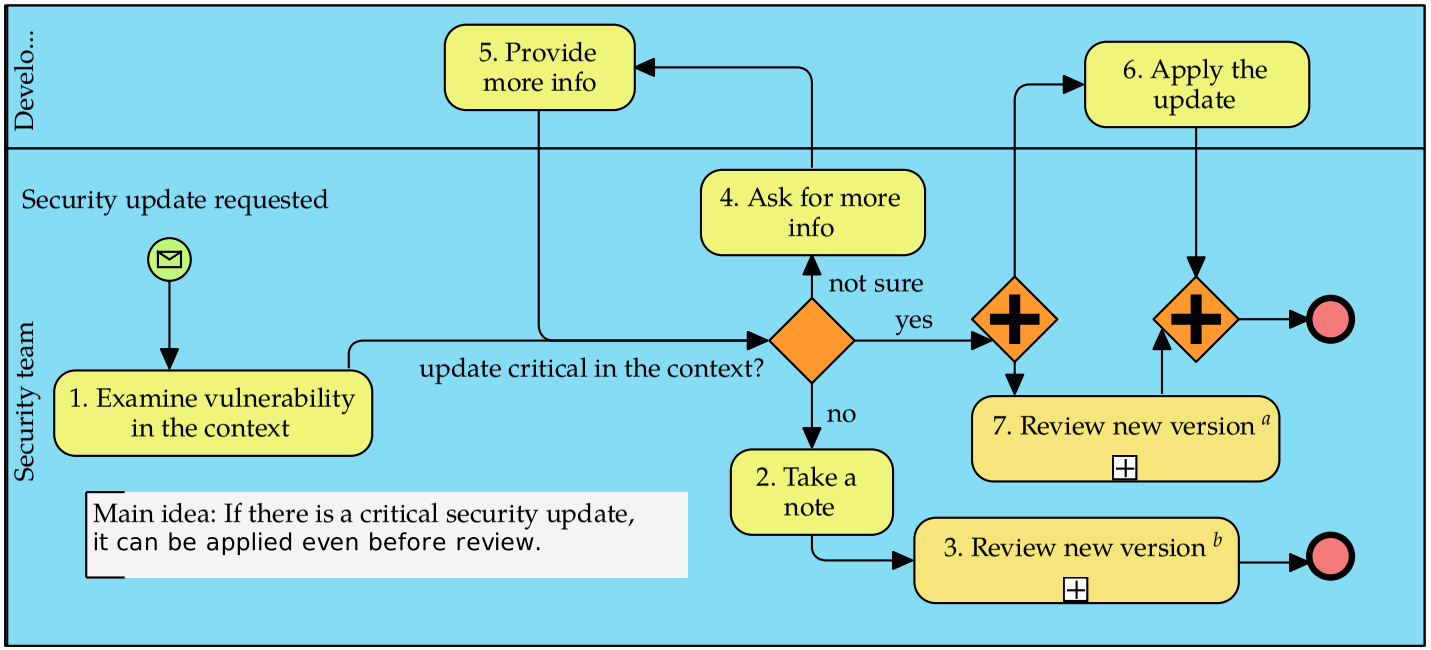

Figure 2.7: Security update (see section 2.4)

a. See figure 2.6
b. See figure 2.6

care. On the other hand, there are libraries where the worst expected vulnerability is DoS (e.g. Java-based regexp libraries) and maybe even DoS is not much likely. Although DoS is not nice for the victim, damages done by this type of attack are bounded and the attack cannot be hidden.

For those reasons, we label all the libraries. A library can have multiple labels. Those labels are related to security and may imply somewhat different approval policy.

2.2.1 Types of library categorization

Library categorization could happen in two ways. They vary in power and in cost.

a. Usage-insensitive approach
b. Usage-based approach

Usage-insensitive approach

In case of usage-insensitive approach, the risks of the library are evaluated regardless the context. This should be easy, although not as accurate as the latter approach.

Usage-based approach

In case of usage-based approach, the labels are given depending on the library usage. For example, a library that accepts input just from trusted configuration files could have some parsing flaws, but those do not matter: If the attacker is able to modify the configuration file, then there are probably some worse problems and abusing configuration parsing flaws might give the attacker little or no additional power.[1]

This approach is theoretically more powerful. However, it requires more time to examine the library in the proper context. Moreover, if the application is actively developer, this approach also requires to handle situations such as:

1. Of course, this conclusion has some undeclared assumptions and should not be taken too generally. This was done just for illustrational purposes.

- Some code changed the way it uses the library. (E.g. it started passing some untrusted input to the library, so it now relies on the library to handle malicious input well.)
- Some seemingly unrelated code has changed. For example, code that deserializes some input data might look unrelated to some other classes. However, they might cause some vulnerability together. [18] [19]
- Some new code uses the library in a new way. (Maybe some previously unused part of the library is used in the new version.)
- Another third-party library has added a change described in one of previous bullets.

As a result, this approach is powerful, but it would be very costly and impractical in real world.

Trade-offs between those two approaches

One might try to balance between those two approaches. The result will be less accurate than usage-based approach, but also can be much less costly in real world.

For example, it might be fair assumption that a chess game server will never execute any untrusted bytecode, because it would imply some very unexpected design. Such application is also unlikely to pass any untrusted input to bytecode manipulation libraries, as they would likely be used just for compiling templates or some similar purpose.

When such trade-off is done with enough care **and reconsidered on every significant application redesign**, it might be the most cost-effective approach.

2.3 Level of approval

There are some libraries that can be approved fully (all features for any project and any purpose). Some other libraries can be fully rejected, because there is a better library for the job. However, there are some cases (discussed below) where a library should not be approved fully, but might be approved partially. Partial approvals can use a combination of following restrictions.

- approved for a particular project
- some feature(s) of the library approved (e.g. approved for bcrypt, not approved for AES)
- approved for a special purpose or type of usage (e.g. good enough if user can pass no input to the library, maybe with some whitelisted cases)

Note that approving a library for a particular project does not imply approving it transitionally to be directly used in all projects that use the project as a dependency. Of course, those projects may use the library indirectly by using a library that is allowed to use the partially approved library.

Partial approval might be, however, hard to enforce. For example, one might accidentally use a feature that is not approved (maybe through a 3rd party library) or the library is added as a transitive dependency to another project. This implies some risks similar to those described in section 2.2.1. **If there is a library eligible for full approval, it should be preferred to libraries eligible for a partial approval only.**

The risk is, however, not influenced solely by technical factors, but also by probability that non-expert developer will try to use the library. For example: A hard-to-use-properly and hard-to-use cryptographic library approved just for some internal library written by cryptography experts theoretically allows direct usage of the hard-to-use-properly library. However, since the library is hard to use at all, such fail is not so likely. Advocating a partial approval of a library based on likelihood of direct usage is, however, troublesome, as it requires wayward human behavior prediction.

When approving a library partially, risks of accidental misuse should be taken in account. Those risks include the reason for not fully approving the library – a serious reason for not approving the library fully might be a reason to reject the library at all, while a minor reason for not approving the library fully might be a reason to approve the library partially.

All programmers should be able to find short reasoning why is a library approved only partially. People are hopefully more likely to adhere to the rules when it makes a sense for them. On the other hand, the reasoning about partial approval should be rather short in order

to prevent the TL;DR effect [20]. Even in complex case, the reasoning should be a basic overview. Programmers should not hesitate to ask the security team for more details.

2.4 Approval of new versions

When a new version of a library is released, the reasoning about approval might change. However, doing a re-check for every minor change of every low-impact library might be cost-inefficient and blur concentration from doing something important. There is also a social aspect: **When one often has to do some rather formal confirmation, the person will likely do the confirmation automatically and thus carelessly.**

For those reasons, approval of new versions of approved libraries might be simplified and/or delayed in some cases.

If the library is labeled as a **"complex library"**[2], then labels have to be reconsidered unless something (e.g. version numbering policy, release notes, ...) implies a minor update that does not introduce any new features. When some labels are added, then consideration related to them is required.

When there is some indication of a more **significant change** (e.g. huge code changes or another commits analysis), then the library has to be re-approved. This means repetition of the whole approval process.

When none of above applies, then a new version of a previously approved library can be used without any approval.

There is, however, one exception to all above. When a **critical update** is released, it might be applied before any re-approval process is performed. The rationale behind this is simple: Not having applied a critical update might be worse than not re-reviewing a new version of some library approved in an older version. This, however, allows just delaying the re-approval process. This does not allow complete skipping of the re-approval process. See figure 2.7

This process is also described in figure 2.6.

2. A "complex library" covers multiple areas and might cover even more areas in some future version. Standard libraries for programming languages are typical complex libraries.

2.4.1 Significant change

Concrete definition of *significant change* should be defined for each project separately and might also depend on library type. There are some signals useful for detecting significant change:

- **Version number increase:** If the library uses a well-defined versioning scheme like Semantic Versioning[21], the significance of the update can be based on this scheme. For example, major version increase in semantic versioning might be considered as a significant change.
- **Lines changed/removed to total number of lines before change:** This is just a heuristic, so subsequent human-based review might suppress it.
- **High number of commits (maybe related to project history).**

2.5 Periodic recheck

As library might get abandoned and some other measures can change even if the library version does not change, a re-check has to be done even if the version has not been changed. (Conversely, if the library gets abandoned, then the library version never changes.)

The frequency will depend on the type of library. If library belongs to multiple categories, the shortest interval of them has to be used.

The process consists of two parts:

- Check if library is not abandoned as defined in section 1.1.
- If the library was fully approved a **long time ago**, do a full re-approval.

It is also described in figure 2.5

2.6 Taking care of implications of disapproval when approving a library

Trying to approve a new version might cause disapproval of the library. Such disapproval might however require rewrite of some related

code. Doing it immediately might not be possible in some cases, as it might require some non-trivial time and there might be some other urgent tasks. Doing it gradually might be hard because of some inter-dependencies. More details on disapproval process itself are described in chapter 3.

This part focuses on choosing libraries with minimum expected risks connected to future disapproval. When approving a library, one should ensure that:

- Disapproval will not be too hard. (It **will** be probably somewhat hard, but one should ensure that it will not be harder than necessary.)
  - Fine-grained libraries are better in general.
- If disapproval is likely to be very hard, then need for disapproval should be unlikely, i.e., the harder the disapproval looks like, the more one should be convinced about the library quality.
  - Documents related to security should suggest good design, which suggests that the library is secure by design, not accidentally.

2.7 Abandoned libraries

An abandoned library will unlikely receive security updates. It also will likely be less reviewed. As a result, the library might use some outdated practices that are known to be broken, but we would not be notified of those. An attacker might intentionally look for such abandoned libraries in the product and inspect them deeply. If one had to care enough about such libraries, the care required would essentially mean maintaining own fork of the library with deep understanding of the code.

2.7.1 How to detect them

Following methods can be used as some heuristics for detecting abandoned libraries:

- Homepage is not available or has changed owner. (For example, one might check whois information or the page content.)
- Latest release is too old. (Being old is however justifiable for some simple libraries or for non-security-critical libraries.)

2.7.2 How to handle them

There are three ways to handle abandoned libraries:

a. The library can be approved for a limited set of purposes where it is expected not to have any bad security impact.
b. The library can be adopted by the company, which essentially means forking the library and taking all the needed care about that. This one is likely to be the more expensive in general, but it might be justifiable in some cases.
c. The library can be rejected (or disapproved).

2.8 Library dependencies

In order to approve a library, all its dependencies must also have been approved. This obviously means we have to approve all the transitive dependencies in defined versions.

There is, however, one exception, which is some version eviction. Say, there is a library A (version 1.0) that depends on library B (version 1.0) and C (version 1.0). Library B depends on library D (version 0.5). Library C depends on library D (version 0.6). Say that library D in version 0.5 is not approved[3], but library in version 0.6 is approved. Now, the resolution depends on the build system. If the build system evicts library D in version 0.5 and chooses library D in version 0.6, we do not need the approval of library D in version 0.5. Without such approval, the library B can be approved only under condition that it is used with library D of version 0.6, of course.

However, this complication might be not so frequent. So, it might be wise to solve the details for a particular build system if this issue ever happens.

3. The library is either rejected or maybe just has not been reviewed yet. Even if it could be approved, it is some extra work that might be unneeded in this case, so we want to avoid that.

2.9 Security updates policy

With some types of libraries, we may assume that vulnerabilities are likely to be present or likely have high impact (or both). For other libraries, these criteria can be used as subsidiary criteria.

For those libraries, we need to ensure:

1. Security updates will be available in a reasonably short time.
2. We will get information about security updates by some channel.
3. We will be able to apply those updates to our product quickly.

2.9.1 Security updates will be available in a reasonably short time

There are some relevant indicators for security updates:

- recent project activity
- past project updates (and security updates)
- commercial support or commercial interests of the author/authors

2.9.2 We will get information about security updates by some channel

There is no strict restriction on the channel type. The channel, however, has not to depend on a particular person. For example, when using e-mail newsletters, some shared e-mail address has to be used rather than some personal e-mail address. This should prevent losing such channel when the responsible person is temporary not available (holidays, illness) or when the responsible person leaves the company for whatever reason.

Depending on policy, the responsible person should be aware that watching such details might cause library name (and maybe version) disclosure. Subscribing to a newsletter gives a 3rd party some information about the software the organization uses if some e-mail address like `security@someCompany.com` is used. Some channels might even publicly disclose list of subscribers.

For any channel, it is advised to have very narrow scope of messages. Any messages that are irrelevant (not related to security or

not related to the particular library or maybe not related to the relevant version) are not welcome. If the security team gets too much such irrelevant messages, one can accidentally skip some relevant and important message.

There are some examples of channels:

- Tools for known library vulnerability scanning (see chapter 4)
- RSS
- e-mail security advisories

2.9.3 We will be able to apply those updates to our product quickly

This covers mainly backward compatibility in some ways. A security update should not break compatibility (unless a vulnerability cannot be patched without a BC break) in order to be applicable just by changing a version number in some project file. It is desired to use a version that has a backward compatible (if possible) security updates and will likely have such support in a reasonable future. When there are some doubts about that, library should be approved at most for some specific project and purpose in order to make migration to a new version reasonably easy.

In general, it is acceptable to release a new backward-incompatible version provided that backward-compatible patches are released for the older version.

2.10 Secure defaults

Insecure or risky default configuration increases the risk of wrong usage. Although this might be considered as a fault outside of the library, default values of a library arguably influences the security of the resulting product, which is the reason we should consider them. This helps to address misconfiguration, which is in OWASP 2013 TOP10. [22] Moreover, insecure defaults might uncover skilllessness of the library author (which predicts vulnerabilities) or outdated library (which might not patching future vulnerabilities).

While insecure defaults can be found on various places (cryptographic parameters, cryptographic primitives, access rights, CSRF [23]

checks, XML entities, ...) and they might depend on context and point of view, we can categorize the insecure defaults to categories:

- unskilled developer or outdated library (e.g. DES or SSL3 enabled by default, null-byte padding)
- point-of-view dependent (e.g. CSRF checks are not default, external XML entities are resolved by default)

The distinction between those two is rather simple: If the default value has no sane reasoning except ability to perform something legacy (e.g. DES or ECB), then we assume that the this is either result of unskilled developer or outdated library. Those values might have made sense earlier, but not with today security state-of-the-art. If there is a sane reason to use the default other than doing something legacy (but the default configuration is still somehow risky, e.g. default write permission or need of explicit CSRF check), then we assume that the reasoning depends on the point-of-view.

The point-of-view-dependent insecure defaults might be discussed and maybe solved by some recommendations and education if done correctly. The security team also might provide suitable configurations, factory methods [24] and other reusable code. This might also lead to creating a facade [3] as suggested in section 1.4. Although this might suggest some kind of partial approval of the library, it would be too strict to reject all such libraries without further discussion.

Hopefully, the only grey zone are configurations that are not state-of-the-art, but are not believed to be broken. For example, using AES-128[4] (although AES-256 is believed to be stronger in all means except related-key attack [@1pass_aes]) is slightly in grey zone. The same applies for HMAC-MD5 or HMAC-SHA1. More controversial, but still a grey-zone, is using MD5/SHA1/SHA2/SHA3 for salted password hashing; It is somehow weak (the brute-force attack is much easier than with bcrypt), but still not fundamentally broken.

Insecure defaults caused by unskilled library developer should imply rejection of the library, as unskilled developer predicts future vulnerabilities. Insecure defaults caused by outdated library should

4. Using AES-128 for communication with embedded devices is pretty justifiable, as AES_128 is still considered to be reasonably secure. However, if there is enough computational power and there are no related keys, AES-256 should be preferred.

imply either switching to a new version (and trying to approve the new version) or rejection of the library if the library is abandoned.

If the insecure default is in grey zone, then this fact might be taken as subsidiary criteria for accepting/rejecting the library. In case of more controversial grey-zone defaults, it is recommended not to approve the library fully. If there is a expectation that a controversial grey-zone default will get considerably worse over time, a recheck should be scheduled. Moreover, replacing this library by another one should be made reasonably easy, unless the default setting is plausible to be changed in an upcoming version of the library.

2.11 Code quality

Code quality influences probability of human errors in logic. [25] This might lead to vulnerabilities at some kinds of libraries, most notably native libraries (memory management errors leading to memory corruption) and access control libraries.

One might want to check various metrics of code quality (tests, readability, ...) for some libraries.

- code complexity metrics [26]
- test coverage metrics [27]

2.12 Review process

When a review is requested, it consists of the following steps:

1. Assign labels to the library. (See section 2.2 and section 2.13 for more details on labels.)
2. Do checks for the labels.
    a. If the library is eligible for full approval, then just handle common pitfalls and approve the library fully.
    b. If the library is not eligible for full approval and has some serious issue, then reject the library and write a reason for rejection.

c. If the library is not eligible for full approval, but it does not have any serious issue, then partial approval might be considered.

The process is also described in figure 2.3.

2.13 Specific requirement for some labels

I have tried to cover all the OWASP 2013 Top 10 [28] weaknesses. Some of them (e.g. [29], [30] and [31]), however, have to be handled on application level rather than on library level. Those will have little or no application in this thesis.

2.13.1 Cryptographic libraries

Quality of cryptographic libraries is essential for security. However, it is hard to do cryptography right. [32] There are many pitfalls that unskilled developer is not aware of – IVs, related keys, key derivation cost (not using PBKDF2 or something similar), wrongly derived keys at all (e.g. md5-hex (half-sized entropy of key)), malleability, obsolete modes of operation etc, need of authentication, oracle attacks, MITM attacks, encryption downgrade attacks, replay attacks and so on. It is easy to make a library that is insecure or has some insecure defaults or is not fool-proof enough.

For this reason, all cryptographic libraries should have got enough 3rd party review.

Fot this type of libraries, security updates are considered to be essential. An unmaintained library or a library with long responses to security issues should be rejected.

Secure defaults are important when evaluating such type of library. There is a checklist[5]:

- **Mode of operation** is suitable by default. ECB should not be used, other modes of operation depend on requirements. [33]
- **Initialization vector is used by default.** Initialization vector is important for security of all modes of operation. [34] Library should encourage usage of initialization vectors.

5. Some items might be inapplicable for some libraries.

- **Initialization vector is encouraged to be unpredictable.** [35]
- **Initialization vector cannot be chosen by attacker.** If it could, attacker is able to make it pointless. [34]
- **Authentication** is encouraged by default, including the initialization vector. Without authentication, various malleability-related attack might be applied. [36] [37] [38]
- Usage of some **well-reviewed cipher and/or signature mechanism** (whichever applicable for that library) without known significant weaknesses should be encouraged. For example:
  - RC4 should not be encouraged. [39]
  - DES shold not be encouraged.
- Usage of well-reviewed hash functions without known significant weaknesses should be encouraged.
- If password-based encryption is used, then some strong key derivation function is encouraged. [40]

Some exceptions for secure defaults might be acceptable for some low-level cryptographic libraries. That is, not encouraging best practices might be tolerated for low-level libraries if it is essentially out of their scope. In such case, it is however questionable if there is a good reason for usage of low-level library and if the library is used by skilled enough programmers only.

Some other requirements will depend on type of cryptographic library.

Cryptographic primitives

For cryptographic primitives, reasonable low-level language is preferred for two reasons:

- Cryptographic primitives libraries usually process just some fixed-length data without complex parsing (which should help with mitigating memory corruption attacks).
- Using a low-level language helps preventing side-channel attacks.[41] Unfortunately, even this is not a complete defense, because some acoustic attacks[42] and other side channels might still exist.

High-level cryptography (e.g. cryptographic protocols)

When using a cryptographic protocol, some existing and reviewed protocol (e.g. TLS or SSH or OpenPGP) should be preferred over a custom one. If a custom protocol is used, then:

- There should be a good reason for using the custom protocol over existing solutions.
- Review of the library should include review of the protocol.

It might be useful to prepare a similar process for cryptographic protocols. Approving libraries that implement some approved protocol would be easier, as we would not have to review the protocol itself. Approved protocols can have lists of specific implementation pitfalls.

When authentication is used, encrypt-then-mac scheme is preferred to other authenticity approaches like encrypt-and-mac and mac-then-encrypt. [38]

When reasonable, existence of replay attacks [43] should be checked.

Especially if some unencrypted or weak variant is implemented, counter-measures against downgrade attack have to be reviewed.

If some compression of encrypted data is implemented, it should be reviewed for compress-ratio side channels:[44]

- It there is any compression used, then special care must be taken about this. (See HTTP/2 header compression design and separated compression contexts.[45])
- If a non-suitable compression is implemented, but not allowed by default, one should evaluate risks of accidental usage by a programmer that is not aware of security impact. If such library is approved, it is advised to approve it for a limited set of purposes.

2.13.2 XML

Libraries that handle XML are potentially vulnerable to XXE (XML External Entity) [46] attack and Billion laughs attack [47]. Both of them are based on handling of XML entities. They might be caused either by bad default configuration of the used library (i.e. insecure default) or by bad configuration at all. In the first case, thee responsibility might

lie on programmer that uses the library, so this in not the library's own flaw. However, even in this case, the library might confuse the programmer to use some insecure configuration, which can have bad consequences there. It is recommended to take care there in order and at least provide some facade that is secure-by-default.

2.13.3 AST libraries

When a library works with abstract syntax tree (e.g. DOM [48] of XML or HTML), serialization should be reviewed, especially in cases where no good serialization exists. For example, bad serialization[49] might occur for such code:

```
createComment("--> <some>XML injection</some><!-- ")
```

In this case, neither HTML nor XML offer a way to write such comment properly. The following straightforward deserialization leads to injections:

```
<!----> <some>XML injection</some><!-- -->
```

2.13.4 Native libraries

Native libraries can bring some risks like memory corruption and subsequent remote code execution (RCE). RCE can be a serious problem even if it handles some security non-critical part of the application. Moreover, Serializable classes can bring security risk even if they are just on classpath, but not used anywhere in the code, because Java deserializaton might result in any Serializable class from relevant classloader. [50].

For this reason, it is advised to use a statical analysis for memory handling.

Usage of a safe memory management tools (e.g. automatic pointers, new C++ string management libraries, etc.) is preferred to old unsafe (error-prone) C functions. Moreover, libraries compiled with memory corruption exploitation countermeasures like canaries [51] should be preferred.

Serializable classes that are using native libraries with pointers or finalize methods should be reviewed for vulnerabilities like [19]. This is not so easy to review due to inheritance. Closed inheritance

(e.g. final classes, package-private constructors etc.) is thus preferred for such classes.

2.13.5 Links

If the library accepts links, it should be checked if the library is vulnerable to XSS[52] attacks like `javascript://%0d%0dalert(1)`, which could be modified in multiple ways in order to obscure the attack and hide it for poorly written filters:

- Additional leading spaces or tabs
- Varying case in "javascript:"
- Multiline links (Bad regular expression might match any single line for a good pattern.)

Protocol blacklist approach should be avoided, as we could theoretically have new protocol schemes to disallow in future and as the blacklist approach is more likely to be implemented wrong (case sensitivity etc.).

Moreover, it should be tested for some internal link schemes (file://, smb://, ...) if applicable.

If the link is used in a redirect, it should be checked if some external link is accepted in the redirect. [31]

2.13.6 HTTP server libraries

If a library operates on HTTP server in a way that it processes requests, it should be examined what countermeasures against Cross-Site Request Forgery (CSRF) are implemented if applicable. CSRF is also an item from OWASP 2013 TOP10. [23]

2.13.7 Authentication

Bad authentication is another part of OWASP 2013 TOP10. [53] If a library implements authentication, the following should be checked:

- If the library handles password storage, passwords are stored securely (i.e., salted and hashed using some slow hash function).

- If the library generates session identifiers, they are not predicable.
- If the library generates session identifiers, they are not malleable.
- If the library manages session identifiers, reasonable countermeasures against session fixation are implemented.
- If the library manages session identifiers, they have limited time validity.

2.13.8 General server libraries

If a library provides some remote service with a requirement for fine-grained access control, it should be checked what measures have been done. [54]

3 Disapproval process

If a library used in the code is disapproved, it would be ideal to remove it and replace it by something better. This can, however, take some time. Replacing a library with another one might also require finding and approving a replacement library. Trying to make this process short might decrease the quality of review and introduce some problems.

As a result, disapproving a library might be gradual. The library might be temporarily approved for a limited set of projects and limited set of purposes. It is advised to make such temporary approvals as narrow as possible, because such libraries are not intended to be used in a new code.

A library being gradually disapproved should be regularly (e.g. every month) scanned for its usage in order to make sure that removal of usage of such library makes progress and the library is not used on any new place.

4 Scanning for known vulnerabilities

In order to minimize cost of taking care about known vulnerabilities, it is useful to automatize the process by some tool. Those tools might differ in multiple ways, including:

- The used database(s) of known vulnerabilities
  - Completeness of the database (number of vulnerabilities)
  - Freshness of the database (number of recent vulnerabilities)
  - Details about the vulnerabilities
- Cost
- Library matching algorithm
- Supported platforms
- Whether it can be used on source code or on built project or on both
- Threats of disclosing list of used software to a third party

4.1 Databases of vulnerabilities

Because some vulnerability databases might be used in multiple scanners, it is useful to compare some of them before comparing the scanners. Two general-purpose databases were chosen.

4.1.1 National vulnerability database

National Vulnerability Database [55] by NIST is a large database of various vulnerabilities that grows by several thousands of vulnerabilities or exposures per year last ten years. [56] So, it can be considered as a vulnerability database with considerable amount of items and stable history.

NVD is free of charge to use for any purposes. [57]

NVD aims to contain all known vulnerabilities of publicly available software. An entry about a vulnerability or exposure has some CVE number. [58] For example, see CVE-2005-1234[1]. Various details might

1. `https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2005-1234`

be available, but the level of details may depend on multiple factors. For example, a not-yet-publicly-disclosed vulnerability might have a CVE number, but its description will be obviously not very verbose.

CVE numbes are usually assigned to CPEs ("Common Platform Enumeration"). [59] A CPE is an identifier of vulnerable software. For example, the mentioned CVE-2005-1234 contains information that it affects `cpe:/a:phpbb_group:phpbb-auction:1.0m` and `cpe:/a:phpbb_group:phpbb-auction:1.2`

References to missing CPEs

NVD might refer to non-existent CPEs. For example CVE-2007-6721 [60] refers to CPE `cpe:/a:bouncycastle:bouncy-castle-crypto-package:1.32`. There is a link to nowhere[2] on the page.

4.1.2 OSVDB

OSVDB is Open Sourced Vulnerability Database. [61] The license is more restrictive: [62]

> 3. If the OSVDB is the basis of, or integrated with in any manner a commercially available product or service you MUST notify OSVDB by providing details on the usage and reach a licensing agreement prior to usage. This includes using OSVDB data in security products, security services, generating vulnerability statistics/ metrics, funded academic research, or any form of analytics used in a commercial manner (including "free" reports used for marketing). Usage requests can be submitted for approval to officers[at]opensecurityfoundation.org. Use of OSVDB in a non-profit or educational organization, in the same manners listed prior requires explicit permission from OSF, and may require licensing.

2. `https://web.nvd.nist.gov/view/cpe/search/results?searchChoice=name&cpeName=cpe%3a%2fa%3abouncycastle%3abouncy-castle-crypto-package%3a1.32&includeDeprecated=true&page_num=0`

4. Obtaining data from this website in a programmatic fashion (e.g. scraping via enumeration, web robot, crawler, etc) is prohibited. Such activity is likely to trigger security software that will permanently block your IP from accessing the site.

The cost is thus not transparent and might vary.

The vulnerability grew about one thousand vulnerabilities per year since 2008. [61] While this is considerably less than volume of NVD and while OSVDB grows rather constantly (in contrast to increasing growth of NVD), the database is still large enough to take in consideration.

Like NVD, OSVDB also contains various details about vulnerabilities, including external links and severity scores. (For example, see entry 115639[3].) OSVDB, however, does not use CPEs now, which might make matching vulnerabilities to vulnerable software harder. Usage of CPE identifiers is planned, but not yet done. [63]

4.2 Vulnerability scanners

Some vulnerability scanners them are compared there with focus to aspects mentioned above.

4.2.1 OWASP Dependency Check

OWASP Dependency Check (ODC) is a scanner suggested by OWASP.[64] It is free of charge and opensourced under Apache 2.0 license. [65] It supports scanning of "Java, .NET, Python, Ruby (gemspec), PHP (composer), and Node.js applications (and their dependent libraries)" and some source code. [66] Level of support for those platforms might vary.

It essentially tries to match a library to a CPE identifier and then list all relevant vulnerabilities assigned to the CPE identifier. [67] Analysis of the source code and behavior shows that assigning the CPE identifier, however, uses various heuristics, based on fulltext search and keywords obtained from (not exclusively) those sources:

- file name

3. `http://osvdb.org/show/osvdb/115639`

- file metadata
- platform-dependent library identifier like GAV (i.e., groupId:artifactId:version)
- frequently occurred package names in a Java library

Although using fulltext for this purpose might be considered as non-elegant and suboptimal, we probably can't do better unless we have some access to a database that contains an useful mapping, e.g.:

- library hash to CPE mapping
- library hash to vulnerability mapping
- GAV to CPE mapping
- GAV to vulnerability mapping

Is seems that all the heuristics are designed to be stable if the dependencies are not changed. However, some heuristics (e.g. `org.owasp.dependencycheck.analyzer.` are not stable when a dependency is updated to a new version. It might be a good idea to detect such situations by diffing the reports.

Analysis of source code and behavior suggests that ODC tries to analyze various types of files, even if they are in some types of archive (e.g., ZIP).

ODC works with NVD[55], but author says it should be easy to extend it for commercial databases. [67][68] The main concern about this might be the library identifier system used in the database. For this reason, I however doubt it would be so easy to extend it for OSVDB until OSVDB adds CPE identifiers or some alternative solution.

When running on Struts, it has been found to use non-existent CPEs mentioned in section 4.1.1. ODC tracks those CPEs, so it might look inconsistent when comparing to the Official CPE Dictionary [59]. This is, however, probably rather an advantage than a disadvantage, because it increases the probability of successful match.

Complexity of library identifiers

There are many ways of identifying a library (see figure 4.1):

- **GAV identifier:** groupId and artifactId and version (i.e. GAV) by which one can locate the library on a Maven repository. (Or one can pick some similar identifier used on another platform than Java.) This is where we usually start.

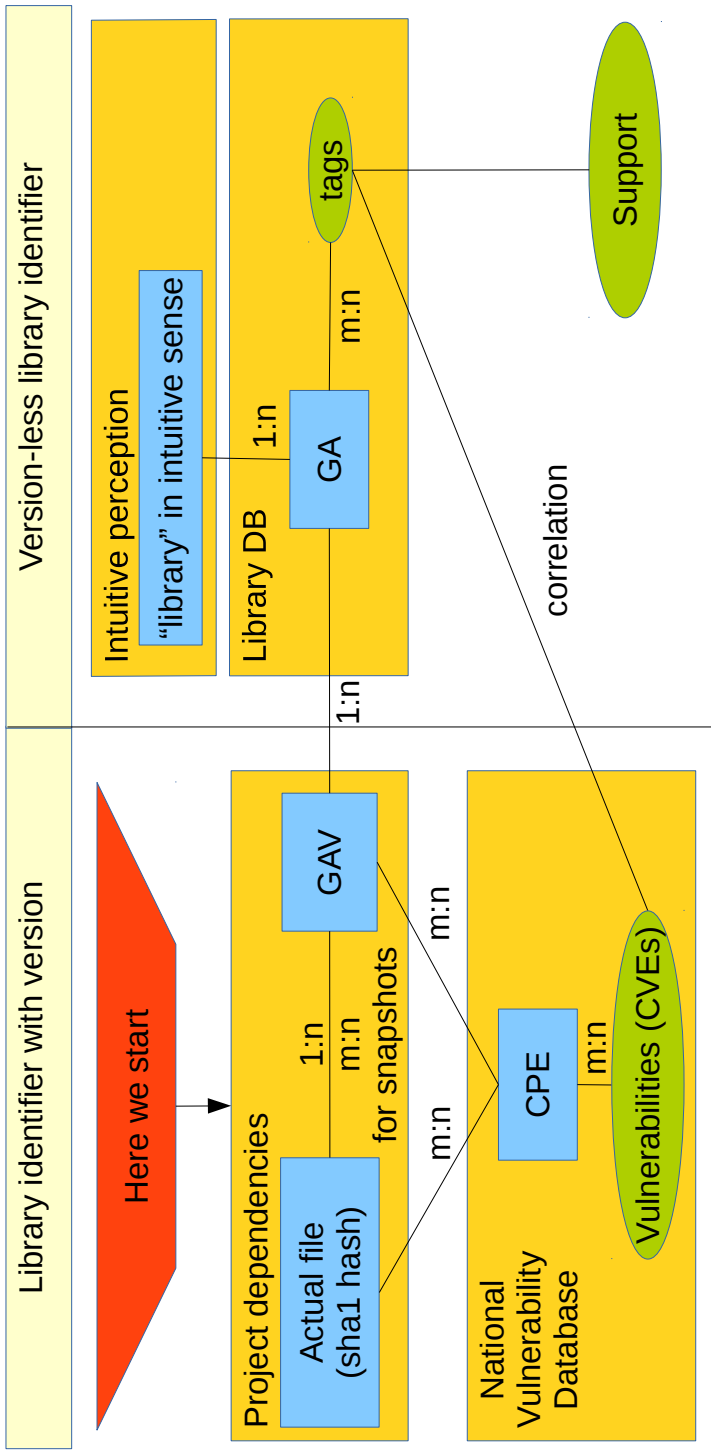

Figure 4.1: Complexities of library identifiers

- **file hash:** ODC uses SHA1 hash for lookups in Maven database. Note that there might be multiple GAV identifiers for one SHA1 hash, so there is 1:1 relation. Moreover, when we consider snapshot, we theoretically get m:n relation.
- **CPE identifier:** This one is very important for ODC, as ODC can't match vulnerabilities without that. Unfortunately, there is no exact algorithm for computation of CPE from GAV or vice versa. Moreover, multiple GAVs might be assigned to one CPE. For example, Apache Tomcat consists of many libraries, but all of them have just one CPE per version. Unfortunately, the ODC heuristic matching algorithm might also assign multiple CPEs to one GAV in some cases.
- **GA identifier:** This is just some simplification of GAV identifier, which misses the version number. There is nothing much special about this identifier for ODC, but we have to work with that, too.
- **intuitive sense:** For example, when one mentions "Hibernate", they probably mean multiple GAs at the same time[4].

Note that this complexity is not introduced by ODC. The complexity is introduced by the ecosystem that ODC uses (e.g. CPEs) and by the ecosystem ODC supports (e.g. Maven artifacts).

Leaking data

By network traffic analysis and source code analysis, we identified that SHA1 hashes of JAR files are sent to Maven central in order to identify a JAR file. If this is not acceptable, one might set a custom server URL or a Nexus server URL. While this prevents sending SHA1 hashes to an external server, it might reduce the accuracy of heuristics by providing less GAV identifiers if there are multiple matching identifiers for one JAR file.

Access to vulnerability database

Vulnerability database is queried from local cache, unless some external database location is explicitly specified. Vulnerability database is

4. `http://mvnrepository.com/search?q=hibernate`

a SQL database. Currently, MySQL and H2 are officially supported, although it would be probably easy to extend the support for some other database. User can explicitly disable or trigger database update, which allows running ODC in a Internet-disconnected environment.

Bundled libraries

While libraries containing all their dependencies (sometimes called "überJARs" in Java world) might be useful in some cases, they are making inspection of all the transitive dependencies harder. While ODC can detect some bundled libraries (e.g. by package names or by POM manifests if included), this is somehow suboptimal. Moreover, if ODC finds a bundled dependency, it might look like a false positive at first look.

Libraries without a CPE

Some libraries don't have a CPE. So, when ODC can't assign any CPE, it might mean either there is no CPE for the library (and hopefully no publicly known vulnerability) or ODC has failed to assign the CPE. There is no easy way to know that is the case.

False positives

False positives are implied by the heuristics used mainly for detecting CPE identifier. We can't get rid of all of them, until a better mechanism (e.g. CPE identifiers in POMs) is developed and widely used. Especially the latter would be rather a long run.

False positives can be suppressed in ODC in two ways. One can suppress assignment of a specific CPE to a specific library (identified by SHA1 hash). If a more fine-grained control is needed, one can suppress assignment of a single vulnerability to a specific library (identified by SHA1 hash). Both of them make sense when handling false positives, depending on details.

Handling a CPE collision For example, there is NLog [69] library for logging in .NET. ODC assigns it `cpe:/a:nlog:nlog`. This might look correct, but this CPE has been used for some Perl project and

there is some vulnerability[5] for this CPE that is not related to the logging library.

If one suppressed matching the CPE, one could get some false negatives in future, as vulnerabilities of the NLog library might be published under the same CPE, according to a discussion post by Jeremy Long (the author of the tool). [70]

In this case, we should not suppress those CPEs. We should just suppress CVEs for them.

Obviously mismatched CPEs CPE might be also mismatched obviously. While the case is similar to the previous one, the CPE name makes it obvious that it does not cover the library. In this case, we can suppress the CPE for the SHA1 hash instead of suppressing every single vulnerability for the SHA1 hash separately.

Missing context

Vulnerability description usually contains a score. However, even if the score is 10 out of 10, it does not imply that the whole system is vulnerable. In general, there are multiple other aspects that might change the severity for a particular case:

- The library might be isolated from untrusted input.
- The untrusted input might be somehow limited.
- The library might run in sandboxed environment.
- Preconditions of the vulnerability are unrealistic for the context.

None of them can be detected by ODC and automatic evaluation of those aspects is very hard, if it is not impossible.

Frontends

ODC has multiple frontends:

- command-line [71]
- Maven plugin[72]

5. `https://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&cves=on&cpe_version=cpe%3A%2Fa%3Anlog%3Anlog`

- Ant plugin[73]
- Gradle plugin[74]
- Jenkins plugin[75]

While the commandline plugin scans filesystem and archives, the Maven and Gradle plugins try to extract the dependency list from the project definition. Unfortunately, the Gradle plugin has been found to include test dependencies in the result. Both Gradle and Maven support splitting the output per subproject. The Maven plugin allows setting much more than the Gradle one.

Running on Maven project vs. running on built application

Maven generally provides more information, so it should usually give more information, which makes the match more likely. However, it is not true that running on Maven projects is always more accurate. There is at least one example when it is not more accurate:
GAVs: `javax.servlet:jstl:1.2 / jstl:jstl:1.2`
Matched CPE: `cpe:/a:apache:standard_taglibs:1.2.1`
Reason: Both of those libs are analyzed similarly (including Maven central search and JAR analysis), but the following evidence is (for an unknown reason) not added in Maven search:

source	name	value
jar	package name	taglibs

Error conditions

When some error happens (e.g. no network connection, full drive and so on…), it usually issues some warning (or maybe even ignores the failure!), so **reading all the warnings and comparing output to previous versions seems to be very useful**.

Output

Output can be written to multiple formats, including HTML (for direct reading) and XML (for further processing). It is however a good idea

to read the log output, as the ODC is very fault tolerant. The log output is, however, huge, so it is impractical to read all the output after every scan.

Especially when having multiple projects, some aggregated view of results is useful. A web tool for aggregating results of such scan has been implemented as a part of this thesis and attached as an electronic attachment. It is also available on Github[6]. Features include:

- looking for error messages in log files
- prioritizing vulnerabilities and vulnerable libraries by severity and number of affected projects
- providing list of projects affected by a particular vulnerability
- providing list of projects containing a particular library
- grouping projects by team
- upgrade advisor: checking if a newer version of a library has some known vulnerability
- library tagging

4.2.2 Veracode scanner

Veracode provides a vulnerability scanner for third-party libraries. [76] Like ODC, it uses NVD as a vulnerability database. It runs as a service with no publicly available information about licensing.

Matching algorithm might differ from ODC, but it is hard to compare, as details are not available.

Veracode does not explain advantages to ODC.

4.2.3 SafeNuGet

SafeNuGet is a vulnerability scanner for .NET, which allows strict blocking of installation of vulnerable libraries. [77] It is recommended by OWASP. [64] It uses its own small and not much frequently updated database.[78] For this reason, it does not seem to be worth of further analysis.

6. `https://github.com/ysoftdevs/odc-analyzer`

4.2.4 Victims

Victims is a vulnerability scanner for Java and Python code. [79] It uses its own database for matching, which grows at most by several dozens of vulnerabilities per year. Despite being small, the database seems to be still actively maintained. An entry in the database contains a link to CVE, so the database is effectively linked to NVD.

Compared to ODC, its matching power is much lower (because of using a tiny database), but it should be more accurate (thanks to the matching database).

Because of the small database, it is probably not worth using as the only vulnerability scanner tool. It might be, however, worth using as an auxiliary scanner along some other scanner. An alternative way of utilizing Victims might be writing a plugin for some other scanner (e.g. ODC) in order to utilize Victims's vulnerability database.

4.2.5 Nexus

Nexus Auditor by Sonatype is a vulnerability scanner for Maven/Java, npm and NuGet. [80] Nexus Lifecycle by Sonatype is a scanner integrated with repository and other tools. [81] It is unclear if these two tools are related to each other, but they probably are just a different bundle of features by the same vendor.

Nexus Lifecycle supports both NVD and OSVDB, which might be an advantage over ODC. [82] However, those two database intersect, so benefit of using both of them is limited.

Price is unclear.[83]

5 Conclusion

Existing approaches include several good ideas for managing third-party libraries security. Complex existing approval process was not found.

An approval process was proposed. The process can be scaled to available resources and tuned to focus on libraries with high security impact. The process includes two kinds of periodic re-checks. First kind is automatic scan for known vulnerabilities, which is important, because checking for known vulnerabilities is low-cost for both developers and attackers. The second kind is manual (might be partially automatized, though) and is focused on checking for abandoned libraries.

For checking for known vulnerabilities, multiple tools are compared. OWASP Dependency Check seems to be the best free tool for this. There are some commercial tools (namely Veracode and Nexus Lifecycle) that look usable, but the price is unclear. There are also some free tools that have insufficient database of vulnerabilities, and thus are practically unusable.

OWASP Dependency Check is studied deeper and a tool for aggregation and further processing is implemented.

Bibliography

[1] Ray Sinnema. *Seven Tips For Using Third-Party Libraries*. Secure Software Development. URL: http://securesoftwaredev.com/2013/01/14/seven-tips-for-using-third-party-libraries/ (visited on 10/20/2015).

[2] Andrew Wagner. *Is Third-Party Software Worth It?* The State of Security. URL: http://www.tripwire.com/state-of-security/security-data-protection/is-third-party-software-worth-it/ (visited on 10/22/2015).

[3] *Facade pattern*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 681404985. Sept. 17, 2015. URL: https://en.wikipedia.org/w/index.php?title=Facade_pattern&oldid=681404985 (visited on 01/10/2016).

[4] *cryptography - Lessons learned and misconceptions regarding encryption and cryptology - Information Security Stack Exchange*. URL: http://security.stackexchange.com/questions/2202/lessons-learned-and-misconceptions-regarding-encryption-and-cryptology (visited on 12/06/2015).

[5] *Four Encryption Mistakes to Avoid » Advanced Software Products Group*. URL: http://aspg.com/four-encryption-mistake-to-avoid/#.VmQHWmQj-Cg (visited on 12/06/2015).

[6] David Wagner. *4.2.pitfalls.pdf*. URL: http://www-inst.cs.berkeley.edu/~cs161/sp14/slides/4.2.pitfalls.pdf (visited on 12/06/2015).

[7] *J2EE third party libraries insecurity - OWASP*. URL: https://www.owasp.org/index.php/J2EE_third_party_libraries_insecurity (visited on 10/22/2015).

[8] *Full Disclosure: [SE-2012-01] Critical security issue affecting Java SE 5/6/7*. URL: http://seclists.org/fulldisclosure/2012/Sep/170 (visited on 12/06/2015).

[9] *New Java exploit on the loose following recent security update*. SC Magazine. URL: http://www.scmagazine.com/news/new-java-exploit-on-the-loose-following-recent-security-update/article/290196/ (visited on 12/06/2015).

[10] Security Explorations. *SE-2014-02-ORACLE.pdf*. URL: http://www.security-explorations.com/materials/SE-2014-02-ORACLE.pdf (visited on 12/06/2015).

[11] redwolfe_98 April 28. *Java Sandbox Bypass Discovered that Breaks Latest Update*. Threatpost | The first stop for security news. URL: https://threatpost.com/java-sandbox-bypass-discovered-that-breaks-latest-update/99868/ (visited on 12/06/2015).

[12] *Experts Find Sandbox Bypass Vulnerability in Java 7 Update 25*. softpedia. URL: http://news.softpedia.com/news/Experts-Find-Sandbox-Bypass-Vulnerability-in-Java-7-Update-25-369044.shtml (visited on 12/06/2015).

[13] *Users' FAQ*. URL: https://www.qubes-os.org/doc/user-faq/#why-does-qubes-use-xen-instead-of-kvm-or-some-other-hypervisor (visited on 12/06/2015).

[14] *Adding third_party Libraries - The Chromium Projects*. URL: https://www.chromium.org/developers/adding-3rd-party-libraries (visited on 10/22/2015).

[15] Mark Samuel. *Are You Creating Horizontal or Vertical Teams?* IMPAQ | Business Execution Systems. URL: http://www.impaqcorp.com/are-you-creating-horizontal-or-vertical-teams/ (visited on 12/06/2015).

[16] *JNI APIs and Developer Guides*. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/jni/ (visited on 12/06/2015).

[17] *Understanding sun.misc.Unsafe - DZone Java*. dzone.com. URL: https://dzone.com/articles/understanding-sunmiscunsafe (visited on 12/06/2015).

[18] *Apache Commons statement to widespread Java object de-serialisation vulnerability : The Apache Software Foundation Blog*. URL: https://blogs.apache.org/foundation/entry/apache_commons_statement_to_widespread (visited on 12/06/2015).

[19] *New Android Serialization Vulnerability Gives Underprivileged Apps Super Status*. Security Intelligence. URL: https://securityintelligence.com/one-class-to-rule-them-all-new-android-serialization-vulnerability-gives-underprivileged-apps-super-status/ (visited on 12/06/2015).

[20] *TL;DR*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 697296566. Dec. 29, 2015. URL: https://en.wikipedia.org/w/index.php?title=TL;DR&oldid=697296566 (visited on 01/10/2016).

[21] *Semantic Versioning 2.0.0*. URL: http://semver.org/ (visited on 12/06/2015).

[22] *Top 10 2013-A5-Security Misconfiguration - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A5-Security_Misconfiguration (visited on 12/07/2015).

[23] *Top 10 2013-A8-Cross-Site Request Forgery (CSRF) - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A8-Cross-Site_Request_Forgery_(CSRF) (visited on 12/07/2015).

[24] *Factory (object-oriented programming)*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 698726541. Jan. 7, 2016. URL: https://en.wikipedia.org/w/index.php?title=Factory_(object-oriented_programming)&oldid=698726541 (visited on 01/10/2016).

[25] *Anatomy of a "goto fail" – Apple's SSL bug explained, plus an unofficial patch for OS X! | Naked Security*. URL: https://nakedsecurity.sophos.com/2014/02/24/anatomy-of-a-goto-fail-apples-ssl-bug-explained-plus-an-unofficial-patch/ (visited on 12/06/2015).

[26] *Cyclomatic complexity*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 692443977. Nov. 25, 2015. URL: https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=692443977 (visited on 12/06/2015).

[27] *Code coverage*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 688072346. Oct. 29, 2015. URL: https://en.wikipedia.org/w/index.php?title=Code_coverage&oldid=688072346 (visited on 12/06/2015).

[28] *Top 10 2013-Top 10 - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-Top_10 (visited on 12/07/2015).

[29] *Top 10 2013-A4-Insecure Direct Object References - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A4-Insecure_Direct_Object_References (visited on 12/07/2015).

[30] *Top 10 2013-A9-Using Components with Known Vulnerabilities - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities (visited on 12/07/2015).

[31] *Top 10 2013-A10-Unvalidated Redirects and Forwards - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A10-Unvalidated_Redirects_and_Forwards (visited on 12/07/2015).

[32] *Top 10 2013-A6-Sensitive Data Exposure - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure (visited on 12/07/2015).

[33] Phillip Rogaway. *Evaluation of Some Blockcipher Modes of Operation*. URL: http://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf (visited on 12/07/2015).

[34] *Crypto Noobs #1: Initialization Vectors*. URL: http://www.cryptofails.com/post/70059609995/crypto-noobs-1-initialization-vectors (visited on 12/07/2015).

[35] *Should CBC Mode Initialization Vector Be Secret - Defuse Security*. URL: https://defuse.ca/cbcmodeiv.htm (visited on 12/07/2015).

[36] *Malleability (cryptography) - Wikipedia, the free encyclopedia*. URL: https://en.wikipedia.org/wiki/Malleability_(cryptography) (visited on 12/07/2015).

[37] *Padding oracle attacks: in depth » SkullSecurity*. URL: https://blog.skullsecurity.org/2013/padding-oracle-attacks-in-depth (visited on 12/07/2015).

[38] *Authenticated encryption*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 684496206. Oct. 7, 2015. URL: https://en.wikipedia.org/w/index.php?title=Authenticated_encryption&oldid=684496206 (visited on 12/07/2015).

[39] *RC4 NOMORE*. URL: https://www.rc4nomore.com/ (visited on 12/07/2015).

[40] *Key derivation function*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 691386055. Nov. 19, 2015. URL: https://en.wikipedia.org/w/index.php?title=Key_derivation_function&oldid=691386055 (visited on 12/07/2015).

[41] *cryptanalysis - How practical are side-channel attacks and how much of a concern are they? - Cryptography Stack Exchange*. URL: https://crypto.stackexchange.com/questions/3775/how-practical-are-side-channel-attacks-and-how-much-of-a-concern-are-they/3778#3778 (visited on 12/07/2015).

[42] *Acoustic cryptanalysis*. URL: https://www.tau.ac.il/~tromer/acoustic/ (visited on 12/07/2015).

[43] *Replay attack*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 684723878. Oct. 8, 2015. URL: https://en.wikipedia.org/w/index.php?title=Replay_attack&oldid=684723878 (visited on 12/07/2015).

[44] *BREACH ATTACK*. URL: http://breachattack.com/ (visited on 12/07/2015).

[45] In: URL: https://http2.github.io/http2-spec/compression.html.

[46] *XML External Entity (XXE) Processing - OWASP*. URL: https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing (visited on 12/06/2015).

[47] *Billion laughs*. In: *Wikipedia, the free encyclopedia*. Page Version ID: 676441905. Aug. 17, 2015. URL: https://en.wikipedia.org/w/index.php?title=Billion_laughs&oldid=676441905 (visited on 12/06/2015).

[48] *W3C Document Object Model*. URL: http://www.w3.org/DOM/ (visited on 12/07/2015).

[49] *Top 10 2013-A1-Injection - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A1-Injection (visited on 12/07/2015).

[50] djorm. *Java Deserialization Flaws: Part 1, Binary Deserialization*. Red Hat Security. URL: https://securityblog.redhat.com/2013/11/20/java-deserialization-flaws-part-1-binary-deserialization/ (visited on 01/10/2016).

[51] *Epilogues, Canaries, and Buffer Overflows - Gustavo Duarte*. URL: http://duartes.org/gustavo/blog/post/epilogues-canaries-buffer-overflows/ (visited on 12/07/2015).

[52] *Top 10 2013-A3-Cross-Site Scripting (XSS) - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A3-Cross-Site_Scripting_(XSS) (visited on 12/07/2015).

[53] *Top 10 2013-A2-Broken Authentication and Session Management - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A2-Broken_Authentication_and_Session_Management (visited on 12/07/2015).

[54] *Top 10 2013-A7-Missing Function Level Access Control - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A7-Missing_Function_Level_Access_Control (visited on 12/07/2015).

[55] *NVD - Home*. URL: https://nvd.nist.gov/ (visited on 10/30/2015).
[56] *NVD - Statistics Results*. URL: https://web.nvd.nist.gov/view/vuln/statistics-results?cves=on (visited on 12/13/2015).
[57] *NVD - FAQ*. URL: https://nvd.nist.gov/faq (visited on 12/13/2015).
[58] *CVE - About CVE Identifiers*. URL: https://cve.mitre.org/cve/identifiers/index.html (visited on 12/13/2015).
[59] *NVD - CPE Dictionary*. URL: https://nvd.nist.gov/cpe.cfm (visited on 12/13/2015).
[60] *NVD - Detail*. URL: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2007-6721 (visited on 12/13/2015).
[61] *OSVDB: Open Sourced Vulnerability Database*. URL: http://osvdb.org/ (visited on 12/13/2015).
[62] *OSVDB: Open Sourced Vulnerability Database*. URL: http://osvdb.org/osvdb_license (visited on 12/13/2015).
[63] security curmudgeon. *[OSVDB-discuss] Using CPE for Product information*. E-mail. Tue Feb 28 02:14:17 CST 2012. URL: http://lists.osvdb.org/pipermail/osvdb-discuss/2012-February/000066.html (visited on 12/13/2015).
[64] *Top 10 2013-A9-Using Components with Known Vulnerabilities - OWASP*. URL: https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities (visited on 10/30/2015).
[65] *jeremylong/DependencyCheck*. GitHub. URL: https://github.com/jeremylong/DependencyCheck (visited on 12/13/2015).
[66] *dependency-check – About*. URL: https://jeremylong.github.io/DependencyCheck/ (visited on 12/13/2015).
[67] *OWASP Dependency Check - OWASP*. URL: https://www.owasp.org/index.php/OWASP_Dependency_Check (visited on 12/13/2015).
[68] *support for other languages - Google Groups*. URL: https://groups.google.com/forum/#!msg/dependency-check/_gBaiHjRdVM/0caNmbGdCAAJ (visited on 10/30/2015).
[69] *NLog*. URL: http://nlog-project.org/ (visited on 12/14/2015).
[70] *Interpreting CPEs - Google Groups*. URL: https://groups.google.com/forum/#!msg/dependency-check/VRPKRN2-_28/CrGMDY8gCwwJ (visited on 12/14/2015).

[71] *dependency-check-cli – About*. URL: https://jeremylong.github.io/DependencyCheck/dependency-check-cli/index.html (visited on 12/13/2015).

[72] *dependency-check-maven – Usage*. URL: https://jeremylong.github.io/DependencyCheck/dependency-check-maven/index.html (visited on 12/13/2015).

[73] *dependency-check-ant – About*. URL: https://jeremylong.github.io/DependencyCheck/dependency-check-ant/index.html (visited on 12/13/2015).

[74] *dependency-check-gradle – Usage*. URL: https://jeremylong.github.io/DependencyCheck/dependency-check-gradle/index.html (visited on 12/13/2015).

[75] *dependency-check – Dependency-Check Jenkins Plugin*. URL: https://jeremylong.github.io/DependencyCheck/dependency-check-jenkins/index.html (visited on 12/13/2015).

[76] *What's in Your Software? | Veracode*. URL: https://info.veracode.com/webinar-sans-whats-in-your-software.html (visited on 12/14/2015).

[77] *OWASP/SafeNuGet*. GitHub. URL: https://github.com/OWASP/SafeNuGet (visited on 12/14/2015).

[78] *History for feed/unsafepackages.xml - OWASP/SafeNuGet · GitHub*. GitHub. URL: https://github.com/OWASP/SafeNuGet/commits/master/feed/unsafepackages.xml (visited on 12/05/2015).

[79] *victims: evd*. URL: https://victi.ms/ (visited on 12/14/2015).

[80] *Open source risk visibility - Nexus Auditor - Sonatype.com*. Sonatype. URL: http://www.sonatype.com/nexus/solution-overview/nexus-auditor (visited on 12/14/2015).

[81] *OSS Governance across your software supply chain - Nexus Lifecycle - Sonatype.com*. Sonatype. URL: http://www.sonatype.com/nexus/solution-overview/nexus-lifecycle (visited on 12/14/2015).

[82] *Sonatype Nexus Documentation - Sonatype.com*. URL: https://books.sonatype.com/nexus-book/reference/rhc-details.html# (visited on 12/14/2015).

[83] *Nexus Repository Managers - Try, Compare & Buy - Purchase - Sonatype.com*. Sonatype. URL: http://www.sonatype.com/nexus/try-compare-buy/buy (visited on 12/14/2015).