**Managing vulnerabilities in 3rd party components**

Introduction

The main objective of vulnerability management is to “Never ship a package with a known, unmanaged vulnerability in the 3rd party components” (see NX-3889: CVE monitoring of shipped librariesREADY FOR WORK).

That is achieved by bringing together 3 components: build definition, security manifests and checking script.

Build definition

Build definition resides in packaging/\_\_\_build\_definition.yml and is a YAML document which, among other things, defines what dependencies are built along with JDELOG EE package and their versions. By the time of writing, it is not used to build production packages, but it surely will, so it is used as a source of truth about JDELOG external dependencies.

Security manifests

Security manifest is a YAML document named SecurityManifest.yml and describing managed 3rd party products and their vulnerabilities. It resides in a directory most closely related to the 3rd party product: in src/ for global external dependencies, like OpenSSL; in src/common/ for embedded code residing in that directory, like YAJL; in src/modules/input/pcap/ for linked WinPcap library; and so on. Its format is strict and is enforced by the checking script. If the checking script finds a manifest not following the format, it will signal a failure and exit.

Checking script

The script residing in src/utils/seccheck.pl

reads the build definition and the security manifests,

requests NVD API for a list of known vulnerabilities in managed 3rd party products,

checks if new (unmanaged) vulnerabilities did appear and

if some managed 3rd party product or vulnerability requires a review,

based on its status and its last review date.

Also checks if new dependencies have appeared in build definition (thus introducing a new 3rd party product which requires to be managed).

Script exits with success status if no problems were found, otherwise it exists with error.

Running a check

It is possible to run a script directly, as $ src/util/seccheck.pl. Current working directory does not matter, as the script will calculate the repository root by itself and will traverse every directory under that root.

If no problems are found, there will be no any output from the script, and it will exit with success status. If there are any problems, they will be reported to <STDERR> and the script will exit with failure status.

Also, a build target is added to the root Makefile, called check\_cves. So the same check may be run as $ make check\_cves. In case of successful run (without any unmanaged problems found) it will also generate AsciiDoc documentation snippet in docs/autogen/report\_cves.adoc (see “Generating documentation” below for details).

If it is needed to debug some unexpected behavior, or just additional verbosity is necessary, run the script with DEBUG environment variable defined: $ DEBUG=1 src/utils/seccheck.pl.

Dependencies

perl 5

perl-YAML

Security manifest format

Following is the description of fields and their mutual relations:

name (required) – common name of a product being described in the manifest

cpe23 – CPE 2.3 identifier of the product, if known (see below). Do not forget to quote the value as YAML has a special treatment for colons

type – either external for the dependencies which are built separately and then linked (like APR), or embedded for the embedded foreign code (like YAJL)

buildSource (required if type is external) – library identifier from the build definition. May be an array

version (required if type is embedded) – version of product that is embedded to our code

versionMap – maps version in build definition to an actual version of product. May be required if a version is specified in build definition is in a different format. May be a string (or an array of strings) specifying a version from build definition, followed by literal ->, followed by real version number. Example is WinPcap: 4\_1\_3 -> 4.1.3

reviewer (required if cpe23 is not defined) – e-mail address of a person who last reviewed this product and ensured there’s no CPE 2.3 identifier is assigned to it

reviewDate (required if cpe23 is not defined) -- the date of the last review. The next one will be enforced by checking script after two months. Dates in the future are forbidden.

comment – any comment you want to pass to the next person who will read/modify/review the manifest. Not used anywhere.

CVEs – associative array of known CVEs, if any. Key is the CVE identifier. Value is another associative array containing:

◦ status (required) – Not applicable means that this vulnerability does not affect our code in any way, and by no chance it will in the future. We’re forgetting about its existence and never need to review it. Observed means that this vulnerability does not affect our code right now but it probably will if some features are added to our code. It must be reviewed every two months. Active means that the vulnerability affects our code and the information regarding that fact must be published along with a new release. It should be reviewed every two months until it is fixed.

◦ description (required) – a description of how this vulnerability relates to our product.

◦ reviewer (required) – e-mail address of a person who last reviewed this vulnerability and set/confirmed its status.

◦ reviewDate (required) – the date of the last review. If vulnerability is either Active or Observed, the next review will be enforced after two months. Dates in the future are forbidden.

What are CPEs and how do we use them

CPE stands for “Common Platform Enumeration”. It is intended to uniquely identify a software/hardware configuration, like an exact application of an exact vendor and of an exact version, or an exact version of an exact operating system, or an exact revision of an exact hardware, and so on.

There are different versions of the specification which are not compatible with each other and which use quite different syntax. NVD supports most of them; however, checking script only supports the latest specification of version 2.3.

The syntax for a CPE 2.3 identifier is quite simple and consists of a number of colon-separated ASCII fields:

Literal “cpe”

Literal “2.3”

A single character a for “application”, or o for “operating system”, or h for “hardware”

Vendor identifier

Product identifier

Product version

The rest of fields are out of the scope of our interest right now.

Checking script only bothers about the fields 1-5. It will manage the 6th (“version”) field itself, and it doesn’t need the other ones.

The vendor identifier may be omitted. Other fields may also be omitted although it doesn’t make much sense.

Some examples of CPE identifiers used for our security checks:

cpe:2.3:a:apache:portable\_runtime – Apache Portable Runtime Library

cpe:2.3:a::zlib – Zlib (vendor is omitted as different versions have different vendors and we’re interested in them all)

cpe:2.3:a::yajl-ruby – YAJL JSON parser (which does not have its own CPE identifier, but Ruby developers are using the code and perform the security management so we can use it to follow their findings)

The full specification of CPE 2.3 may be found here. I do not encourage anyone to read it in full as it is pretty lengthy and knotty.

Finding CPE identifiers

A task of finding a CPE identifier is not as trivial as it seems to be. There are several methods, more or less deterministic, and the one I found to produce best results with the least effort is the following:

Given a common name of a product, e.g. “Apache Portable Runtime” just enter in the search field of Google that name in quotes, followed by the word “CVE” (without quotes) like that:

You need those quotes because Google likes to create synonyms on his own, like ”iconv” and ”libiconv” are the same thing for him.

If you see nothing relevant on the first page, you may be quite sure that no CPE identifier is assigned to the product. Proceed with adding a managed product without a CPE, specifying a review date. After two months the checking script will remind that this check should be repeated.

If there are some CVEs, they will likely be among the first results, pointing to http://cvedetails.com, http://cve.org or http://cve.mitre.org. Follow the link found and make sure that the CVE is related to that exact product you’re searching for.

If yes, copy the CVE identifier, concatenate it to the URL of NVD: https://nvd.nist.gov/vuln/detail/<CVE\_identifier>, and open NVD page of that CVE.

Under the section “Known Affected Software Configurations” find a CPE identifier corresponding to the product you’re searching for.

Other possible methods are:

⦁ To use NVD Official CPE Dictionary;

⦁ To use NVD CPE API;

⦁ To use NVD Vulnerabilities search page with “Advanced” search type and use auto-completion feature of the “Product” field

But I found all of them less convenient and requiring more effort.

Version treatment

Versions of external dependencies are extracted from build definition. It has two types of version declaration: a minimal version declaration, like 1.0.1, and an exact version declaration, like =1.0.1. The former instructs the build system to use version no less than declared but to try to find a newer version if possible. The latter instructs to use that exact version even if newer one is available.

Checking script ignores the difference between these declarations. Any of them means that build system is allowed to use the version declared, thus vulnerabilities must be checked against that version, not the one build system actually uses for some specific build.

For example, if build definitions declares version 2.3.0 for Expat library, which is affected by many vulnerabilities, but actual Jenkins build logs shows that version 2.4.8 is actually used instead, it is a good idea to bump that version declaration in build definition and not to manage a bunch of out-of-date vulnerabilities in security manifests (INF-1293: Hard code dependency versionsBACKLOG was filed for that purpose).

For the embedded code, the only source of truth about the version used is a developer’s declaration which must present in a security manifest for every mentioned embedded 3rd party product.

Process of review

Reviewing a product

For managed 3rd party products, reviewing process is enforced every 2 months. Having a managed 3rd party product without a CPE identifier means that previous reviewer failed to find a CPE identifier for the product which usually means there is no known CVEs affecting the product. The one who is reviewing such a product should follow the steps in “Finding CPE identifiers” section of this document. If still there is no known CPE identifier, one should update reviewDate field of the security manifest with today’s date, and to set reviewer field to one’s e-mail address to reflect the new reviewer. After that a security manifest should be committed and pushed to repository, thus leaving a track of the review process.

If CPE identifier is found, it should be put to cpe23 field of the security manifest. After that, reviewer and reviewDate fields may be removed, but that is not required. Reviewing process will not be enforced on products having cpe23 field fulfilled.

After that, one should run checking script once more to check if some CVEs appear that are affecting product version used with JDELOG. If there exists ones, they should be made managed by putting them under CVEs array in the security manifest.

Reviewing a vulnerability

Active and Observed vulnerabilities are reviewed every 2 months.

For Observed vulnerability one should check the vulnerability description and then check in our code if some new conditions appeared that makes the vulnerability to affect our product. If so, status should be changed to Active.

For Active vulnerability one should check with bug-tracking system if an issue to fix the vulnerability exists (and to file one if it doesn’t) and if it is resolved. If solution is implemented and is merged to the main branches, status should be changed to Not applicable.

After that, reviewer and reviewDate fields should be updated accordingly and the security manifest should be committed and pushed to repository.

Generating documentation

Run src/utils/seccheck.pl --gendoc <path to output file> to generate an AsciiDoc documentation snippet which describes what vulnerabilities JDELOG dependencies are affected to. The following information is included for every vulnerability: CVE identifier, dependency name (from security manifest’s name field), link to NVD, a list of builds affected, and a vulnerability description.

Please note that --gendoc is not a “dry run”. The checking script performs a normal security check as described above, and generates the documentation along the way. It is implemented that way for the following reasons:

⦁ CVEs' descriptions are not cached on our side and are requested from NVD in real time and included into the generated documentation. Those descriptions change sometimes, they’re getting re-reviewed by NVD and Mitre, and I believe to include the latest version of description is a good thing.

⦁ It may turn out that after such a re-review vulnerability does not affect that exact version of dependency we’re using any more. In that case it will not be included in the generated documentation, but that information may be obtained in real time only.

If checking script discovers any kind of problem during its execution (unmanaged problems, impossibility to connect to NVD API etc.) it will not generate the documentation. To check script’s exit status should be mandatory therefore.