École Polytechnique Fédérale de Lausanne

Deep trust chains may violate security expectations

by Florian Standaert

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Neuenschwander Antoine
External Expert

Antony Vennard
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 15, 2021

Everything should be made as simple as possible, but not simpler.
— Albert Einstein

Dedicated to my family. I wouldn't be here finalizing a Master Thesis without them.

Acknowledgments

A huge thanks to Prof. Mathias Payer and all the member of the hexhive lab for their welcome, their support, their guidance, and the extremely useful feedback they gave me. I really enjoyed the time I spent with them, despite the particular condition. The experience it gave me will certainly be one of the most important in my student's time to prepare me for my working life.

I thank my family that supported and encouraged me all along my studies. They let me have the best condition to study and learn and I probably wouldn't have succeeded in EPFL without them.

Lausanne, January 15, 2021 Florian Standaert

Abstract

Today software are not build from scratch, but build on top of what already exist, often using third party creation. It makes the software dependent on third party. These dependencies are distributed with the software and are part of it. These dependency relationships create ecosystem of packages that can be represented by a dependency graph.

Some previous works have looked at the graph as a whole, simply analysing the graph: how many packages are alone, how deep can a dependency be? Other works have analysed the effect of a known vulnerability in such graph. They concluded that the vulnerability exist for a long time and it can affect many packages.

This work differs in that it has a concrete question to answer and try to improve the situation: can we find packages to focus on monitoring and testing to find such vulnerabilities.

Contents

Chapter 1

Introduction

In modern industry software development has time constraints to not take too long. This make impossible to recreate everything each time so majority of the projects relies on existing code that provides some functionality. These packages end up forming an ecosystem. In this ecosystem package depends on each other which forms a dependency graph. With these dependencies what we want and consider is that they are maintained and trusted. Is it really? What happens if we are wrong.

In that kind of environment we cannot check everything and maintenance have a cost, be it money or time or other. As many previous work show there can be bug and vulnerabilities. They were interested in the effect of these bugs. How it propagates, how long until the correction. They were observer and studied a posteriori security.

In this work we want to come a priori. We are interested in optimising the prevention. Where to act in search of problems and maintenance. The ecosystem is too big to check every libraries but we want to find where or what are the packages that are the most important or have the more impact.

What we are interested are mostly three points: first what package to focus the test and review. The idea is that a package highly used that we could ask if we want to include it in the standard library would be the most important to maintain. Secondly what package we don't suspect but could have the most effect in case of a problem. If an attacker can focus on one package to have the most impact what is this package? Lastly what package can we use to monitor the ecosystem. Which package are the most likely to catch/show as many vulnerabilities as possible in the graph.

The security maintenance is done on a package level, but on an ecosystem level we are still reacting a posteriori. This can be the first step to change that: starting to analyse the ecosystem graph.

Analysis can be interesting and fun and in many domain some part of analysis and play

are necessary. Here, as a master thesis we prefer when there is some goal or some cap to avoid drifting to nowhere. To resume as simply as possible our cap it would be :can we find where to focus the review and maintenance?

Short answer is we have a way. Future work would certainly improve the result, but this work already give some result to be used.

Why is this work important? Many others have already studied dependency graphs and vulnerabilities. The importance of taking care of the dependencies' security to secure our package has already been noticed and reported with multiple examples. However, all these works are a common point. They play with the graph to see if they find something and come a posteriori when a vulnerability is found and known. This work is original in that it comes a priori and focus on a specific goal to find where to work in priority to secure the software.

Chapter 2

Background

To understand where we come from with software ecosystem let's see why it emerged. At the beginning when we created a software programmer had to include everything needed in the program. Be it their own creation or some shared code it had to be manually included and distributed as one. Everything was compiled together. As time passed the program became more evolved and complex and these searched codes became complete utility tool sets and program needed more than one of them. Manually including them became tedious and prone to create errors. People came to notice that some these tool sets are used in multiple software and that there are multiple copies of them on a limited system, disk and RAM space were really limited at the time. So they reflected and searched to improve the situation. The answer was really simple: we install the tool set, also called library, once on the system and we share it, we have shared libraries. The program won't include the library when it is installed, but it will use the one that exist on the system. It was a great idea, but there was a few problems. The most basics were how to use them reliably and how to be sure to install every needed library with a new software. The first one was handled by having the library fixing some interface to use them. The second problem was first managed manually and anyone who installed a program had to check manually for the library. That was called the DLL hell. They needed some help from a saviour, a new kind of program: the package manager. This package manager will be given information on the program and library it can install and each piece of code will tell what it needs to run properly: the package management and dependency were born.

Let's quickly review how a manager works. When you install a package it asks a repository for this package and its information. In the information there are the dependencies. The manager then check if it already has the dependency and asks for those it doesn't have and continue recursively until it has all packages. Then, it resolves in which order to install them wanting to start with the packages that have no dependency or all of them are already installed solving if necessary any loop and cycle, finally installing everything.

The management can work similarly between languages, but there still are differences. Lan-

guages have their particularities and that influences the resulting ecosystem. When the language have a solid functionality base like a standard library the ecosystem can have more standalone packages that don't need any dependencies. When the language is just a language without anything else the base installation can be more lightweight, but many little packages for the more basic functionality can appear and the ecosystem might be more connected. Different languages are created to respond to different need and it is normal to have these differences.

Graphs are a mathematical object consisting of a two sets: a set of vertices and a set of edges connecting the vertices. We usually differentiate if the edges are directed or undirected to have directed graphs or undirected graphs.

It can be used to represent many types of data and if in this project we use it to represent dependency relationship in recent year the development is more focus on social graph.

On graphs there are some metrics and functions that have been developed to extract some properties and structural information. One of those is the notion of centrality of a vertex: How important and central is a node in the graph generally or relatively to other nodes [4] [6].

Another function that was developed on graph is clustering. How are the vertex link so that they can be formed in subgroups. Here, the high-level idea is to see if the graph is regularly connected or on the contrary there are subgroups really connected with only a few connections between subgroups.

Chapter 3

Design

3.1 Goal

Today software is not created from scratch. It is built on top of the knowledge and tools created previously. One of the most important parts are the pieces of software providing different capabilities called libraries. We can create a graph of which piece of code or software depends on which other code. The goal of this project is to analyse this graph. When the security of a program is studied it is usually only done for the software. Sometimes when we find a vulnerability in a library we study how it propagate. This project aim is to have a first analysis and possibly spark the interest of studying the security at the ecosystem level.

When using a library the developers don't have time to review it. To not block all the process it is usually assumed that it is correct and if a vulnerability should be found the library would be updated and corrected. This kind of assumption can lead to devastating effect if the library is not correctly maintained, abandoned for whatever reason, lack of funds, etc. One example is openssl. It is a free open source library that was relying on donation and benevolence for maintenance but with time there were less and less and they didn't have enough to do a job corresponding to the importance of the library use. It ended up with a critical failure known as heartbleed. On a more concrete note it is not possible in practice to check and review everything. In this analyse we will have three direction in mind.

The first and principal direction is to try to find some base set of package. Is there some package that are widely used and could serve as basis. Here, we had the idea of potential standard library candidates. Can we create a list of complete enough libraries that should be well checked and maintained, and direct the community to focus the work on that at first. Some kind of basis the developers could better rely on.

For our second approach we look from an attacker point of view, their target of choice. Can we find an unseen, unexpected packages that have big impact. Here, we want the packages that

have the most impact in the ecosystem if something unexpected or unwanted happens to it. It would probably be mostly similar to the first list, but is there some new package? Some extreme example would be a piece of code used by only one or two other packages, but these packages are themselves used and continuing to create a chain that reach a huge part of the existing packages.

Lastly we attempt to find packages that use many others and could be uses as some kind of monitors. Software use libraries, but some use more than other. If we find some packages that have many dependencies we might test it to find potential vulnerabilities in this package or in the dependencies.

3.2 Getting the data

The first step in this project is to have data to work on. These datas need to be saved somewhere. We also want to consider the fact that we need to work on the data so a simple file in memory is not a good idea. The use of a database here seems obvious, but then how to structure our data we are working with a dependency graph. A simple search give the idea of a graph database. When it is important to choose carefully the database here being a research it is not as critical as some production release so made our choice quickly using the comparison on Wikipedia [1]. We settled for one that as many usable languages and is free or has a free version: neo4j.

Gathering the data would need to be done for each ecosystem we want to analyse, but here are the example we did.

For python pypi ecosystem we work with python. We first get the list of package from the public API https://pypi.org/simple. Secondly, we connect to the public API at https://pypi.org/pypi/package name/json. Then for each of these package we extract the name and the dependencies and use the neo4j driver to connect to the database and send a query to add a new node for the non existing package and create the links to the dependencies. Getting the data is as simple as that but took some time to get the database query right to be able to fill the database in a sensible time.

For node.js npm ecosystem we use node.js javascript to get the data. To get the list of package we connect to the registry of npm, here at https://replicate.npmjs.com, that give a list of log from which we extract the name of the packages. With this list we use the module npm-registry-fetch to get information on each packages and get the name and dependencies. Then, similar to python we use the npm version of neo4j driver to send the query filling the database.

Now that we can get some data we need to decide what kind of data we take. We can go with multiple idea: do we take all we can as much as we can, or do we take some common subset that we can find in all ecosystem? In the end we went with the bare minimum: only the packages. Let's see why we didn't take some other information.

One of the first ideas to augment the data we have is to take the versions that we should find in all ecosystem. Let's see two point about it. The first is the version constrains. For a given dependency it is extremely rare to use an equality, but usually it is on the form a version bigger than xx. When you take into account the possible prerelease and beta version and that that some ecosystem can have looser versionning rules it quickly turns time-consuming to get it right. The other point is that with all the version the size of the graph simply explode. We are on an 4-6 months project and the time constrain push for simplification.

After that we can consider information on how the package is maintained: The number of maintainer, the frequency of commits, of new revision,... It can be great data but where do you get them and how do you treat them? You might not always find the source on git. When you have many commit but no new version what does it mean? and and when the maintainer is the address of a group or a mailing list do you consider there is many maintainer? That kind of data might need a study on it own. Very interesting but not the point here.

Another information we could get is the number of download. It could be used to see the importance of a package: if it is use by many system even alone it is important. We tried to see if we could get that data. On some ecosystem there is a data download easily accessible, on other it is a bit more difficult to get, and the comment on that data suggest that it might not be existing for many ecosystems. With what we could access we checked what we had and we see that the data is at the package level. We can have a number of download of a package bigger than its dependency. How? a package can not work without its dependencies. But the number being at the package level it doesn't separate for each version, or update. It lead to a noisy data that is difficult to work with at this stage.

Now that we know what data we want, the package,its name and its dependencies, we have to get them and we already have to deal correctly with versions. When we talk about dependencies we have the mandatory dependencies but also the optional. And the different version of a code may need different dependencies. As we don't care about version of a package we decided to take the dependencies of the latest stable version. As a first analysis we want to focus on the most important part so we take only the mandatory dependencies.

Each ecosystem using a specific versionning system it is difficult to recreate it right so we try as much as possible to use the tools from the ecosystem, in our example it is the package semver in npm and a part of pip for python pypi.

When we prepared the data we found that in the ecosystem there are different rules for the names. In some like npm it is strict naming but in pypi it is case insensitive and hyphen and underscore are interchangeable. In the database the name property being strict it is necessary to normalise the names to avoid splitting package and having wrong result. More generally the data are not perfect as is and there are some cleaning and conversion to put them correctly in the database.

3.3 Analysis

Now that we have the data it is time to analyse them. What would be good to check is if every package are correctly maintained but it is not really doable so a secondary idea could be to know which package to check in priority.

As a general point all along the project every time a database query or algorithm working on the whole database must be run we must take care and optimize as possible. Here the use of neo4j is interesting because it provide (as plugins) a collection of function and algorithm necessary here. For centrality and clustering we only use provided algorithm mostly because recreating new algorithm in query is not possible considering the time it take to run on the database and the constrain of a 4 months project.

The first exploration was with centrality algorithm. Long story short it was not good. Thinking about it it can be understood. Centrality are made for network and social graphs. Here the dependency graph have some particularities that can have great influences. They have a strong orientation: the package that don't use anything are at the bottom and the ones that are final/not used by anything are at the top.It is not exactly a pyramid because it can have more package at the top but the idea is here, they build on top of each other. Another point is that a dependency is necessary and be forgotten which make side dependencies as important has the one in the shortest path from one end to the other and that is highly different from network and social graph that greatly rely on the idea on shortest path for the importance of nodes.

After that we explored some clustering. We have the Louvain algorithm, Label Propagation, Weakly Connected Components, and the Local Clustering Coefficient.

The first we discard is the local clustering coefficient. It uses the number of triangle the node is in to compute a probability that the neighbours of a node are connected. What kind of node do we search? A node that is used as base in a dependency graph, it doesn't matter if the neighbour are connected, usually they are not so much.

The weakly connected components algorithm separates the graph in its disconnected components. It can be useful in optimisation to work on each part of the graph separately but we don't need it for our graph so we can put it aside.

The Louvain and label propagation algorithms are more interesting at that point. They group the nodes with the nodes they have the most connections. At that point the interesting clusters are big so need another step. We tried to use a different clustering algorithm on the different cluster separately. Now that we have our clusters which one do we start to look at? Do we want t ocluster the most connected internally? AS we want some base in the graph we are more in search of external connection so we output the cluster with the most external link IS_DEPENDENCY_OF. We print that as a graph to visualise the results. Here what we get have some unexpected parts: starting from the most connected cluster the following decrease rapidly, and these top cluster

they all rely on one node connected. We could have expected that some would have one node that do all the connection but with the clustering we wanted to find some potential split package that together are more important and that is what we didn't found (in our ecosystem).

After that we came back to something simpler, we only evaluate the property for each node without grouping them. Let's learn to better use the KISS principle: Keep It Simple, Stupid. We consider 2 numbers:

- The number of direct connection in which the package IS_DEPENDENCY_OF.
- The size of the subgraph (number of node) in the subgraph following link IS_DEPENDENCY_OF.

The scale of these 2 values are different and cannot be mixed as is. We normalise them between 0 and 1 by simply dividing the value by the max value among the nodes.

With these 2 values the first thing we can try is to weight them. Starting from 50/50 if we move toward more on the direct links we can start to find node that are a the center of a star with many related package but a bit in there own corner. When we take more of the subgraph size we start to get less obvious node but with high impact. It is interesting and to take but the extreme of only taking the subgraph size is perfect for the unexpected little packages of huge impact. The best compromise for the pseudo standard library like packages seems to be 50/50 between the number of direct link and the size of the subgraph.

With searching the base of the graph done we can be interested in package that can show the bug of many packages, some kind of monitors nodes. the principle is basic: the node that touch to the most nodes is the none that have the biggest subgraph when using links DEPENDS_ON. When we check the results we see that we cannot expect package no naturally have that subgraph as big a from the bottom with IS_DEPENDENCY_OF. Another point we found is that the top node can be related and even if not there is a huge chance that the subgraphs overlap which make it not that great. WE could search at each new node the one that have the most different nodes but that is a computation to long for our time constrain will be left for future work.

We still check what we got.

Chapter 4

Implementation

Our project start with the installation of the database. As we work with dependency graphs we look at graph databases. We want a database that have a free licence and can handle as much languages as possible. with only that we already only have two candidates. ArangoDB and Neo4j. We need the database to have function and procedure on graphs already implemented for performance reason. Depending on the information we save the graph can become really large and we want a database that doesn't take days for every single query. A quick search to choose between them about which one we find the most function included or installable was used. The first database for which we found many interesting procedure is Neo4j.

There exist an open source community version for Neo4j and we used it. The functions and procedure of the database can be extended with plugins and through out the project we installed 2 plugins given by neo4j: the apoc plugins and the graph data science plugins. Now that we have the function we need to deal with the configuration and more particularly with the memory configuration. Because we deal with big graphs, i.e. npm has more than 1 million nodes, the procedure need more memory to run. Neo4j come with some administration tool and we use the command neo4j-admin memrec to get the recommended configuration. We followed it except for the memory. There are two size we configure: the heap and the pagecache. To avoid problem you want as much as possible, but you still need some memory for the system so you cannot give it all. We had 16GB of RAM and ended having a bit less than 8GB for each.This project work on a commodity PC hardware like Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz with 16GB RAM. If someone want to redo or improve this work we suggest to be careful and have a machine with lots of RAM but you might consider a more dedicated high performance machine. We also had a uniqueness constrain on the node property called name.

Once we have the database we fill it. We work with two ecosystem, npm and pypi, obviously having them separated.

Getting data on npm first check if the exist an API to connect to. npm doesn't have one. To get data from npm we need to connect to its registry. In order to reuse some existing code and

example it was decided to code this part in javascript node.js. Indeed asking the registry for informations on packages and solving the dependencies are something the package manager is already doing and we can base our code on that or directly use it as is. First we ask the registry for a list of all the package to get there name. The registry respond with a list of notification update like info from which we extract the list of name. Registries is made like that mostly because it is used to synchronise between servers, clients don't ask for the complete list but on the contrary only ask for a specific package. Node.js work mainly in an event approach and when we ask for the list we receive a sequence of notification. We save that list in a file. We use linux command sort and uniq just in case.

Once we have the list of name we can ask the registry for each package specifically and get more information like its dependencies. For each package name we ask the npm registry and because we code in node.js we can use the package called npm-registry-fetch that is used in the official package manager of npm. With the response we prepare the neo4j query that fill the database with the new data.

In the response we get from the registry we have many data. What we would expect is just to use the field dependencies but the data are not structured only at that level. A package can have multiple versions and the dependencies are related to a particular version. We use the informations on the latest stable version. Most of the package have in the field dist-tags the tag latest and we use that version. When we don't have this latest tag we need to pass through all version searching for the latest non pre release version. We use the same tool as in the package manager to compare versions. With these informations we can create the links of dependency between packages. In our graph we want to analyse the subgraph going recursively through these links but analysing only a part of the graph can not be considered at this point and we need the complete graph which let us deviate from the package manager that would need to ask recursively form the dependencies to install them. Here we know that we will ask for them at some point because we get informations on all the package and they are in the list.

The final optimised version of this query is also used for other ecosystem. As it is a development there can be some redundant features, especially because the uniqueness constrain was not in the configuration at the beginning. We need to create the nodes for the package if they don't exist and create the links. Directly sending the query to the database is the first try but the database can handle more than one query at a time and with the approach of event from npm we end up having multiple query at the same time. In itself it should not be a problem but in our query we create if the node doesn't exist. and there where time when multiple query related to a same node were concurrent it created duplicate node because they both check and not find it so they both create it. We were inserting one node at a time and not all dependencies at once.

Because of that point, not having yet the uniqueness constraint, we created in the code a queue to ensure that only one query is made at a time and we insert all dependencies of a node at once. The node.js code has the event of fetching the registry and processing the data at response but it then put them in a queue. There is an infinite loop that look in the queue and

send sequentially the insert queries ensuring the end of the previous one before continuing. When all the name have been fetched a part of the code wait for all the responses to return and put a special value at the end of the queue to terminate the program.

```
1.
var query="MERGE (p:Package {name : '"+name_pack+"'})";
var i = 0;
for (var k in dep){
      query=query+'\nMERGE (q'+i+':Package {name : "'+k+'"})\n'+
      'MERGE (p) -[:DEPENDS_ON]->(q'+i+')\n'+
      'MERGE (q'+i+') -[:IS_DEPENDENCY_OF]->(p)';
      i++;
}


2.
const query_batch="MERGE (p:Package {name : $name})\n"+
    "with p\n"+
    "unwind $listdep as depparam\n"+
    "MERGE (q:Package {name : depparam.depname})\n"+
    "MERGE (p) -[:DEPENDS_ON]->(q)\n"+
    "MERGE (q) -[:IS_DEPENDENCY_OF]->(p)";
```

At that point the program work and as it is getting data once the optimisation was not overly pushed but it was really slow and at some point we search deeper in optimisation and and found that if the exact value of the query change the database redo some operation called planning and that for executing query it is best to keep them short. We also found how to use parameters in query so we rewrote what we had to put some easy optimisation giving the currently used queries.

After npm we prepare the data from python pypi. Here they have an API we can easily connect. There is one page in simple html that list all the package names and for detail on package it is returned in json format. To get python data we code in python. We request the full list of package name and put it in html parser beautifulsoup to get the list of name that we save in a file. As for npm we sort and uniq the file. Python being sequential we can simply request the data for each package, process them and run the database query to insert them. The sensitive point in pypi is the processing part. We had to use the internal module of the tool pip to parse the dependencies because all data are not completely clean and redoing a correct parser is to complex and slow here. The other point to not forget is to normalise the name if we want to avoid split in package. At the end because the dependency version constrain in the data is not checked at upload we have a few error and we manually process these 50-ish packages.

The two main ecosystem are quite big and in development we don't want to always wait

for every test and if the querys must be optimised we need to have some usable runtime so we generated some reduced test set of data. To give a size idea the pypi graph have in the order of 250000 nodes and npm have more than a million nodes whereas our test data have only around 9000 nodes. To get a sensible test graph we generated it by taking parts of the two main dataset.

Once we have the data we start to analyse and test different algorithms but we quickly get to an new point to solve: how to see the results if we want to check them, how to make these results understandable? We need to visualise the the result in some way. The principle used is that the algorithm write the result as a node property and we export the graph data as json, then we use a python script to read the json file and create the visualisation. Effectively it is in two steps. We use the the program and tool graphviz to create the final image but it can work in a particular manner. Graphviz create some language called dot to describe a graph but also its representation like the colors. The python script take the json file and create the dot file. It selects which node to print, set the colors, etc. Graphviz come with multiple graph placement tools like nodes in circle or a flow from top to bottom. Because of the size of our graph we need to use the toll made for big graphs called sfdp.

Now that the necessary parts are more or less in place we can dig further in the analysis of the graph. Here from an implementation aspect there is only one point: optimising the database queries. We are searching a graph so we test different thing and it is only some database query but when to just test we need to be careful because a bad query can run really slowly. For example some bad query to get the number of node reachable for a specific node in the directed graph can end with an out of memory error after tens of minutes on the reduced test set when a good query can run on the real big graph in a matter of minutes. These are example of actual queries. The first one is the query that cannot complete and the second using implemented procedure is the fast one.

```
1.
MATCH (n)-[:IS_DEPENDENCY_OF*0..]->(dep)
with distinct dep,n
set n.node_subgraph_dependency_of=count(dep)


2.
MATCH (n:Package)
CALL apoc.path.subgraphNodes(n, {
        relationshipFilter: "IS_DEPENDENCY_OF>",
    minLevel: 0
})
YIELD node
with n,count(distinct node) as cn
set n.node_subgraph_dependency_of=cn
```

Chapter 5

Evaluation

In this chapter we prove that our work answers what we wanted to know so let's first recall the research questions:

RQ1 Can we find where to focus review and maintenance in the ecosystem.

RQ2 Can we find the target of choice for an attacker.

RQ3 Can we find the package that catch the most problem in the ecosystem.

This work have a different approach from what already exist and our evaluation is qualitative. There won't be a hard numerical score saying that it work but it doesn't lessen the usefulness of our work.

5.1 RQ1

To answer this question we tried different metrics and in the end we used a simple mean average of the normalized number of direct package depending on it and the normalized size of the subgraph. Because the scale of both number were different we normalised between 0 and 1.

For each package:

ido : the number of packages it is a direct dependency of

$idon$: ido normalised between 0 and 1

sdo : the number of packages in the subgraph where it is a dependency of

$sdon$: sdo normalised between 0 and 1

$$idon = \frac{ido}{\max ido}$$
$$sdon = \frac{sdo}{\max sdo}$$
$$RQ1score = idon + sdon$$

This score is computed for each package and we could all rank them but we want a few important package to focus and we can see that the score decrease quickly at the beginning so we thought of taking the arbitrary value of 10 first values. We finally took a little margin and settled for 15 values here.

Package	Score
requests	1.7425071670575971
numpy	1.5066819737707111
six	1.362262038073908
urllib3	0.8792139813098586
certifi	0.8558864885038311
chardet	0.8546486962225042
idna	0.8437400660205606
pandas	0.7860095131427105
click	0.7419392146091073
odoo	0.7070615448118519
python_dateutil	0.6715497217083265
pytz	0.6530819198890822
scipy	0.6523147163161102
pyyaml	0.6272479714860599
setuptools	0.5700296118722994

We take the 15 first package and check them. We present them here grouped by the type of package it is. For the python pypi ecosystem we first have the obviously expected widely known and used package requests. More oriented math, science, data, etc. we have the as much known packages numpy, pandas, scipy.

Then we have the packages six and chardet. chardet is a character encoding detection which is not surprising to find considering the necessity to be ready for international character. And with six they are in the compatibility group. They provide compatibility between python 2 and 3 which is no surprise to be greatly used.

The next category is the connection and internet packages. It should not be a surprise to anyone to find that category in the most used packages. We have urllib3 a http client, certifi which handle certificates and idna for domain names.

After that we have a category that can be forgotten but hardly denied its entry the command line interface with the click package.

Next we see the packages python_dateutil and pytz. They are time, timezone and date related packages. Saying that there are datetime components in many programs is not really a big news and their places here make sense.

Then we check the pyyaml package. It is in data serialisation category, using yaml. For many people knowing software development the category data serialisation is understandable but for those that don't know python they might ask why not finding json. That is related to the particularity of python and show that each ecosystem has its particularities. We find yaml but not json in the package because json is already in standard library and doesn't need an external package.

One more package is setuptools that do package management. For people that are not familiar with package management it can be weird to a that as a dependency, we might think we just want to use it but it does many thing and many package can use some part of it.

The last package is more peculiar. It is odoo. The program odoo is an Enterprise Resource Planning (ERP) and Customer Relationship Management (CRM) software. We can wonder why we find it here and when we check we find that the package is a basis for a huge number of other odoo package i.e. odoo13_addon_mail_activity_done. They are almost all odoo related module with different package for each odoo version. It is definitly unexpected that a subgraph or cluster like that is big enough to appear an we could see two possibility: the first is that this metric is not perfect, no metric is perfect. The other possibility is that on the contrary we need to handle that package because is more important than expected and wanted and this metric is doing the job. We have in appendix the last update date for each package and considering the update of this package we consider that this package need to be addressed or at least check by the maintainer if it is still needed or safe.

Now we check that result of this metric for npm. We also have 15 package that we manually review.

Package	Score
lodash	1.8877173091458805
chalk	1.326659947797331
debug	1.1487382560641788
inherits	1.025719306941847
safe-buffer	1.0011997355192759
tslib	0.9762310147321757
commander	0.9172445392716881
ms	0.8702511593593335
request	0.8582113420471602
glob	0.8383033874688901
color-convert	0.8094562423309863
color-name	0.8086018259813353
semver	0.7854389685804115
uuid	0.7820448059833877
ansi-styles	0.7676803569966091

We start with the package lodash. It is an utility library for javascript. We directly see the difference in ecosystem and who it influence the important packages. Javascript and node.js with npm don't have a standard library and alternative are in package form and the most used. Other package of utility are debug for debugging utility.

Next we have chalk. Here we touch to the output with some terminal output styling. Another package we got is ansi-styles. Here it might be difficult to separate the output related package and the styling package so we also got color-convert and color-name. In javascript and node.js the output is extremely basic and we see the need to improve that with packages.

After that we have a group we could call language improvement. We have inherits, safe-buffer and tslib. tslib bring typescript in node.js. safe-buffer is made to replace when needed the default node.js buffer api with a more save version as its name imply. inherits is a browser friendly compatibility package. Here we see again that an ecosystem has particularities. This node.js npm ecosystem is influenced by the web from where javascript come and the node.js is use on server side web.

The next package is commander and we arrive in the command line interface package.

Then the ms package that stand for millisecond and is a time related package.

Next an http client package request.

These 3 package are not that surprising and they have equivalent in python. It is reassuring to see that we find common part between ecosystems.

The next group could be called advanced utilities. We have the glob and uuid packages. glob do pattern match with shell pattern and uuid create strong unique id.

The last package is semver. It handle semantic version. It is part of package management, a group we also had in python.

To summarise the first research question we want a list of package that are widely used and important. The list of package correspond to what we want and show package that we didn't think of but correspond to what we want. Our method not only work but is useful which makes it a good method.

5.2 RQ2

For the second research question we wanted to find some attacker choice for a target. As for the other question we tested different value but settled with simply the number of node in the subgraph following who depend on the package. We again must check manually the results.

We check python first. As could be expected there are a number of package that are the same as in question RQ1 in a different order. We can list them: six, idna, chardet, certifi, urllib3, requests, numpy, pytz, python_dateutil, setuptools .

For the other packages we have first scandir that is supposed to improve the performance of existing function in standard library but has already been included in recent python standard library version. The first explanation for it still existing is that in python older version still exist and are used hence the need of that kind of package.

Similarly the other packages pathlib2, contextlib2, zipp and configparser we have are all backport in old python version from the newer python version of different the capabilities and improvements.

In node.js npm we also have the common package with RQ1. The list being inherits, safe-buffer, lodash, ms, debug, color-name, color-convert, ansi-styles and chalk.

Compared to python in npm there is no big need for backport so we don't find that category but instead we find another one particular. The packages we get are has-flag, supports-color, balanced-match, wrappy, once and brace-expansion. They are some little utility and we could ask why that kind of little thing appear. Here we don't want package widely use directly, we want package that have impact. It can be used by few package directly but these packages are in turn used more widely. To show the legitimity of that category we can refer to the real case of the package left-pad that broke almost all install when it disappeared.

Summarising the second research question we want a list of package that directly of indirectly impact the biggest number of packages. The list we get indeed correspond to that and the partial overlap with the list from RQ1 is normal and a good sign. Having some of the package that are important from RQ1 to have a impact on many other package and showing up in this list

is normal. On the other hand having some package that are new are the one that makes this research question significant.

5.3 RQ3

The last research question take an opposite view. We are dealing with a dependency graph, with dependencies. This kind of ecosystem has the particularity that if the dependency fail the package that depend on it will fail. The idea is can we use this fact and find package that will fail with the biggest number of dependency and maybe use them as some kind of monitor or the system.

The metric for that is simply the packages that have the biggest subgraph when we follow the dependencies.

Contrary to to the bottom of the graph there is not obvious category of package that could be found. We noticed an interesting but undesirable fact on that front: some of the good monitors are link between them and span the same or a very similar subgraph. In addition the size of the subgraph when we take the package that we depend on is small by order of magnitude compared to the subgraph when we follow which package depend on us.

An easy example is with NPM. If we take one of the top package of RQ1 or RQ2 the spanned subgraph it is a dependency of is in the order of 200000-300000 packages. On the contrary the top package for monitoring span a subgraph it depends on in the order of 1000 packages.

Preparing security on the ecosystem level is a good idea and should be done but on the practical side we can review and maintain the important base and the final check must be done by each package reviewing it own dependency graph.

Chapter 6

Related Work

In this part we review the work done by our predecessors on related subjects.

Studying graph in general is not new. Freeman in 1977 already did some research on graph with the study of measures of centrality [4]. At the time the study was on human communication, what we today call social graph. The intuition is that they want to have a way to rank the different person of communication center according to there importance. When a center is used to relay information it has a responsibility to do it correctly or on the contrary has power to manipulate and influence the information on the network. In this paper they pose the concept and the method to compute the first version of what is called betweenness centrality. The measure they developed has the novelty at the time that it worked on unconnected graph.

Scott White and Padhraic Smyth have been interested in the relative importance between entity in networks [6]. They don't study the graph or an entity but the entities in the graph.

Others like Sharma, Grover and Kumar have worked on components interaction and dependency [5]. They don't attach to a particular sort of component and it can be used with library component in a software or on system component like server and database. These graph are different from the dependency graph that we work with but are still related. They proposed a method to represent dependency among components. It is a work for developing tools for use in the study and not a study on dependency graph yet. It was a necessary previous step.

Some groups showed interest in studying the ecosystem impact of some fact like a security vulnerability [3]. This work is closer to our work as it study the ecosystem in case of a vulnerability. The difference with our work is that they study known vulnerabilities with its effect and reaction a posteriori where we try to come in the graph a priori and point where to search for vulnerabilities and correct them. This paper is a good argument to justify our work. We often if not always underestimate the impact of a vulnerability in an ecosystem. A few of there findings are that it takes a long time to fix a vulnerability once discovered but it also take a long time to discover the vulnerability.

An other paper that is closely related to us is [2]. They also try to see what a dependency graph resemble but just study it on a statistical point where we try to use it and have concrete question to answer. It is still interesting and related in the fact that they show the difference between ecosystem and these difference can have some visibility in our results. Their study was on CRAN PyPi and NPM. Directly from [2]:

> While only 31.5% of all PyPI packages have dependencies, this proportion jumps to more than 58.7% for NPM and to more than 69.8% for CRAN. Similarly, while more than 62.3% of all PyPI packages are isolated, only 34.6% (resp. 22.2%) of the packages on NPM (resp. CRAN) have neither dependencies nor reverse dependencies.

> We observe that most packages have fewer than 500 dependent packages. While CRAN (resp. PyPI) only has 34 (resp. 55) packages with more than 500 dependent packages, we observe that there are 1,636 NPM packages with more than 500 dependent packages. This includes 420 packages having between 500 and 1,000 dependent packages, 696 packages between 1,000 and 5,000 dependent packages and 520 packages having more than 5,000 dependent packages.

Chapter 7

Conclusion

In this work we started by developing a system to collect then analyse data about dependency in software ecosystems. For that we rely on a graph database and script to connect to API of each ecosystem.

We analysed ecosystems of dependency to find some of the important package that could be a first focus in maintenance and review. We also pointed out some package that might be a choice for attacker and that we should not forget to check for security.

We tested our system on real ecosystem NPM and PyPi and the given result let us check that our system work correctly and is able to give sensible result. Said result are packages of security interest in the ecosystem.

This work give a first method to do that kind of evaluation and if in the future it will be improved it already have a working starting point.

It is but a start and the subject of taking the ecosystem as the object of maintenance and security review is a long way to go but important. These ecosystems are a not completely new surface of attack but with the growth of such ecosystem so are there importance for an attacker and it is up to the security community to consider the problem before the full bloom of the attacks. In a way or another research in that direction will continue because of the emergence of ever more such ecosystem and package manager, be it with new languages that takes it into account at creation time or the development of manager on older languages out of necessity.

Bibliography

[1] In: URL: https://en.wikipedia.org/wiki/Graph_database#List_of_graph_databases.

[2] Alexandre Decan, Tom Mens, and Maëlick Claes. "On the topology of package dependency networks: a comparison of three programming language ecosystems". In: *Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, November 28 - December 2, 2016*. Ed. by Rami Bahsoon and Rainer Weinreich. ACM, 2016, p. 21. URL: http://dl.acm.org/citation.cfm?id=3003382.

[3] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the impact of security vulnerabilities in the npm package dependency network". In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. Ed. by Andy Zaidman, Yasutaka Kamei, and Emily Hill. ACM, 2018, pp. 181–191. DOI: 10.1145/3196398.3196401. URL: https://doi.org/10.1145/3196398.3196401.

[4] L.C. Freeman. "A Set of Measures of Centrality Based on Betweenness". In: *Sociometry* 40 (1977), pp. 35–41. DOI: 10.2307/3033543. URL: https://doi.org/10.2307/3033543.

[5] Arun Sharma, P. S. Grover, and Rajesh Kumar. "Dependency analysis for component-based software systems". In: *ACM SIGSOFT Softw. Eng. Notes* 34.4 (2009), pp. 1–6. DOI: 10.1145/1543405.1543424. URL: https://doi.org/10.1145/1543405.1543424.

[6] Scott White and Padhraic Smyth. "Algorithms for estimating relative importance in networks". In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Washington, DC, USA, August 24 - 27, 2003*. Ed. by Lise Getoor, Ted E. Senator, Pedro M. Domingos, and Christos Faloutsos. ACM, 2003, pp. 266–275. DOI: 10.1145/956750.956782. URL: https://doi.org/10.1145/956750.956782.

Appendix A

result

A.1 Python package result

A.1.1 RQ1

package	last update
requests	Jun 17, 2020
numpy	Nov 2, 2020
six	May 21, 2020
urllib3	Oct 19, 2020
certifi	Nov 8, 2020
chardet	Jun 8, 2017
idna	Jun 27, 2020
pandas	Oct 30, 2020
click	Apr 27, 2020
odoo	Jun 28, 2014
python_dateutil	Nov 3, 2019
pytz	Nov 2, 2020
scipy	Nov 5, 2020
pyyaml	Mar 18, 2020
setuptools	Oct 17, 2020

A.1.2 RQ2

package	last update
six	May 21, 2020
idna	Jun 27, 2020
chardet	Jun 8, 2017
certifi	Nov 8, 2020
urllib3	Oct 19, 2020
requests	Jun 17, 2020
numpy	Nov 2, 2020
pytz	Nov 2, 2020
python_dateutil	Nov 3, 2019
setuptools	Oct 17, 2020
scandir	Mar 9, 2019
pathlib2	Sep 27, 2019
contextlib2	Oct 10, 2019
zipp	Oct 25, 2020
configparser	Oct 9, 2020

A.1.3 RQ3

package	last update	last prerelease update
animal_classification	Dec 31, 2018	
punctuation_stylometry	Apr 19, 2020	
brasil.gov.temas	Feb 22, 2019	
mesmerize	Sep 30, 2020	
aiidalab	Jul 30, 2019	Nov 6, 2020
brasil.gov.tiles	Mar 13, 2020	
emrt.necd.theme	May 13, 2019	
quantum_computer	Jul 24, 2020	
products.poi	Feb 24, 2020	
unikold.connector		Apr 23, 2019

A.2 NPM package result

A.2.1 RQ1

package	last update
lodash	2020-08-13
chalk	2020-06-09
debug	2020-05-19
inherits	2019-06-19
safe-buffer	2020-05-10
tslib	2020-10-09
commander	2020-10-25
ms	2019-06-06
request	2020-02-11
glob	2019-11-06
color-convert	2019-08-19
color-name	2018-09-21
semver	2020-04-14
uuid	2020-10-04
ansi-styles	2020-10-04

A.2.2 RQ2

package	last update
inherits	2019-06-19
safe-buffer	2020-05-10
lodash	2020-08-13
ms	2019-06-06
debug	2020-05-19
color-name	2018-09-21
color-convert	2019-08-19
ansi-styles	2020-10-04
has-flag	2019-04-06
supports-color	2020-08-28
chalk	2020-06-09
balanced-match	2017-06-12
wrappy	2016-05-17
once	2016-09-06
brace-expansion	2020-10-05

A.2.3 RQ3

package	last update
npm-all-packages	2020-04-28
1000-packages	2019-09-28
bloater	2020-04-30
potionseller	2020-06-13
no-one-left-behind	2018-02-10
sindresorhus.js	2018-11-08
ucic-collection-web	2020-05-22
big-bertha	2016-10-18
amb	2015-05-18
moekit	2015-07-03
merino	2017-03-20
mspm	2014-03-13
ctos	2014-05-16
c2s	2014-05-13
component-transformer	2014-05-10