Dependencies:
No Software is an Island

Thesis for the degree Master of Science

Jørgen Tellnes <`jorgen@telln.es`>

October 2013

Abstract

In the past years, package managers, application frameworks and open-source libraries have made it vastly simpler and faster to get functioning software up and running, while cloud providers and external service providers have made it easier to get the application out into the hands of millions of users without large up-front costs.

While this recent technology development has made it possible for companies with limited resources to build impressive software and valuable services, the development has serious security implications which the current state of software development and systems engineering are not yet able to handle very well.

In this thesis, we will show that the security and availability of a system are largely determined by the surrounding "ecosystem" of dependencies, and that techniques to reduce the reliance on a system's dependencies—software libraries, services and infrastructures—are hugely beneficial.

The intended audience for this thesis are computer scientists, professional and amateur software developers, and system designers, but anyone with basic IT knowledge is encouraged to keep reading.

Acknowledgements

Tusen takk til Tone, min kone. Takk for at du har tolerert at jeg har hatt lange dager på universitetet, og takk for at jeg har fått lov å lesse masse tanker og idéer på deg, selv om det ikke alltid har vært riktig tid eller sted. Takk for at du har hatt troen på meg når jeg ikke selv har hatt det.

Tusen takk til min veileder, Kjell Jørgen Hole, for mange interessante samtaler, veldig mye god oppfølging, gode råd og uvurderlig hjelp. Tusen takk for at du alltid har hatt tid til å lese over eller prate, selv på en lørdag, og selv om du har hatt veldig mange andre baller i luften.

Tusen takk til Mikal, Ole, Stian og Andreas for god stemning, god kaffe, god hjelp til korrekturlesning og ærlige tilbakemeldinger.

Tusen takk til Mamma og Pappa for at dere har støttet meg under hele dette prosjektet, og bidratt med gode samtaler og middagsbesøk.

Contents

Chapter 1

Introduction

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
– Leslie Lamport [1]

Everyone depends on something outside themselves. We depend on the electricity company to deliver the power that keeps us warm and powers our gadgets, and we depend on the fire department to save us if our gadgets catch fire. There are a set of entities we all depend on to keep us safe, warm and happy. At the same time, dependence can be a burden and a risk to ones welfare. Dependence on alcohol or narcotics has ruined lives or even whole societies. The same goes for systems and code, depending on something can be beneficial, but it can also be risky or even outright dangerous.

A good programmer is a lazy programmer [2]. This makes perfect sense, as a lazy programmer doesn't waste his or her precious time reinventing the wheel or spend time on things that a framework or third-party library can do faster and better. The rise of modern package managers has accelerated this trend toward using a multitude of third-party libraries, and modern cloud-based architectures have accelerated the trend toward using multiple third-party services "in the cloud". While maintaining and reducing the dependency of other systems are an important part of managing a software project, most development teams do a shoddy job.

An important issue is that the web of entities a system depends on may be larger than what is immediately obvious, and a large number of the entities may be hard to discover even with rigorous analysis. An example illustrates the problem at hand. The availability of almost all Norwegian Internet banks depends on user authentication with BankID, which again depends on the plugin block-lists that are maintained by browser vendors. The lists are an example of non-obvious and ill-defined dependencies that can severely affect the availability of the banks at any time.

As a further example, due to a configuration error made by an Indonesian Internet Service Provider (ISP), Google went down for about a half hour in November 2012 in large parts of eastern Asia. This incident was entirely outside Google's control, and was simply caused by the dependencies inherent in the structure of the Internet.

The point is, code and systems don't exist in vacuums. They sit in the middle of a huge and ever-changing ecosystem, interacting with many other systems and other pieces of code. Keeping these dependencies to a minimum, decoupling as much as possible, and at the very least being aware of them is essential to the security and reliability of a system.

In this introduction, we define the concept of dependency and explain how dependencies occur in modern Information and Communications Technology (ICT) systems. We then discuss some prior work in detecting dependencies, show how graph theory can be used to describe and analyse dependencies, and finish by providing an outline of the rest of this thesis.

1.1 What are dependencies?

In its broadest sense, a **dependency** is a relation between two entities where one entity depends on the other for something. An example is that most people depend on the emergency services if a fire breaks out. In ICT systems, one can say that a system depends on the underlying infrastructure, or that a program has a dependency on a third-party library. The extent or importance of dependencies can vary, as well as the impact in the case of a failure in a dependency. Not all dependencies are equal.

A **security dependency** exists when the security of an entity depends on the security of another entity. The security of an office door depends on the security of the door to the janitor's office, where all the spare keys are kept. The security of a Windows desktop computer will similarly depend on the security of the domain controller[1] it is associated with [3]. The security of a service running in a public cloud depends on the security of the cloud, as an attack on the cloud infrastructure can disrupt or compromise the service.

Dependencies are necessary. A system that doesn't communicate with other systems is likely a system of limited scope and usefulness. In many cases it makes perfect business sense to outsource parts of the application

[1]In Windows enterprise domains (Active Directory), a domain controller handles passwords and authentication (among many other things) for all computers in the domain.

to third parties (e.g. SMS gateways, payment processors, cloud storage and specialist libraries) to avoid the cost or maintenance overhead, or to reduce time to market.

Problems occur when the system is tightly coupled [4] to a dependency, or when there is no control over what dependencies the system has. This situation can quickly land the system in "Dependency hell" [5], from which there is no easy way out. Especially since the dependencies of a system often have a set of dependencies that they themselves rely on, adding indirect dependencies to the original system.

Faults in complex systems are often attributed to a root cause, but Cook [6] asserts that attributing failures to a single root cause is fundamentally wrong. A failure in a complex system is almost always caused by multiple faults that only create a failure when combined.

Mismanaged dependencies, overly large number of dependencies—or simply a lack of awareness of a systems' dependencies—increases the risk that dependencies outside the scope of the system will be contributing to a system failure.

The term "dependency" is used in many different settings. There are dependencies in program code; between different libraries, which will be studied closer in Chapter 2; between classes, functions or even—in the context of compiler optimisation—between individual CPU instructions. In Chapter 3, we will look closer at dependencies in the infrastructure around program systems. This thesis will mainly focus on library dependencies and infrastructure dependencies.

1.2 Discovering and mapping dependencies

It is important to properly understand the "ecosystem" that a system lives in, to know as much as possible about its dependencies. How many are there, where and what are they? What uptime does the Service Level Agreement (SLA[2]) for an external system guarantee? Does the system use an outdated library or API with known vulnerabilities? In many systems, the answers to these questions aren't known, either from lack of documentation or from the sheer complexity of the system and its surroundings.

There have been some prior works on automatically discovering dependencies in a network, but most works have either been of the empiric kind (listening to network traffic to infer dependencies) [7, 8] or just attempting

[2] A contract that defines the service level (uptime, response time and other guarantees) for an external system or service, and available recourse for a breach of the agreed-upon service level.

to make a framework for describing and modelling dependencies [9] without attempting to tackle the problem of discovering them.

Empirical observation of network traffic can discover and map most dependencies during normal operation of a system, but this activity is not sufficient to discover exceptional dependencies that only emerge in failure scenarios. Failures in complex systems depend on multiple failures in collusion [6], so knowledge of exceptional dependencies are important to fully discover potential failures.

To discover all dependencies, an empirical method must exercise *all* possible system states as well as all possible states of all external systems. This is impossible, as there may be dependencies that only manifest at a specific time and date, at a specific location, or when a specific set of failures occur in a specific way. Empirical methods are not viable for detecting exceptional dependencies, but are still useful for discovering dependencies in a system's normal operation, and can be useful when trying to build an understanding of normal system operation that later can be expanded upon.

Automated static analysis of source code is able to detect most dependencies that are defined in the source code. Dependencies outside the scope of the source code, such as fallback systems, databases and other infrastructural elements, and dependencies only associated by correlation cannot be detected through static source code analysis. Not all dependencies can be found in the code, and not all dependencies can be observed from a system in the normal running state. Combining these techniques with a deep knowledge of the system and its surroundings seems to be the most viable way to discover most dependencies.

1.3 Describing dependencies with graphs

Graphs are mathematical structures used to describe entities (nodes) and the pairwise relationships between them (edges, represented by lines between nodes). When the direction of the edges means something, the graph is directed, and the direction of the edges is signified by arrows. If the direction of the edges is unimportant, or the edges are implied to be bidirectional, the graph is said to be undirected. See Figure 1.1 for an example of a graph showing the relationships between a group of friends, where an edge represents the relation "is friends with". Note that the graph is undirected, as a one-directional friendship is somewhat meaningless.

The same kind of graph can be constructed for dependencies, resulting in a **dependency graph**, where an edge signifies "depends on." Dependency graphs are directed, due to the nature of the dependency relation. In the

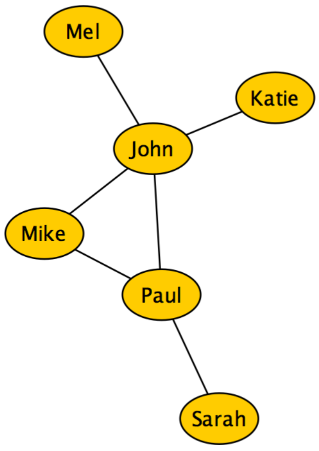


Figure 1.1: *A small group of friends, visualised with a friendship graph. The graph is made with the graph drawing software yEd [10].*

event of a co-dependency, the edge will be bidirectional.

A graph can be measured and analysed in ways that systems, infrastructures or software cannot. We can measure and visualise the graphs contained within the larger context, stripping away unnecessary detail and noise, and transforming them from being imaginary structures to an actual tangible, measurable form.

A dependency graph can be traversed (by e.g. Kruskal's algorithm [11]) to find a Minimal Spanning Tree where any unnecessary edges are removed or to discover if there are any circular dependencies in the graph. A circular dependency is when a node ends up depending on itself, through a number of intermediaries.

Shortest-path algorithms like Dijkstra's algorithm [12] can be used to find the shortest dependency chain from a node to another. The average degree (number of edges going in/out of a node) is useful to see how dense the graph is. Other useful metrics include the graph diameter, which is the "width" of the graph,[3] and betweenness centrality, which is a measure of how many of the shortest paths in a graph pass through a specific node.

These metrics can quickly draw a picture of the size of the dependency graph, and the relative importance of the different nodes. The total number of edges in graph divided by the number of edges in the minimal spanning tree can be used as a measure of duplicated edges (edges that don't contribute to the minimal spanning tree).

[3]The longest distance between any two nodes, or more precisely the longest shortest path in a graph.

1.4 Structure of the thesis

We'll take a look at the rest of the chapters in this thesis. The chapters are outlined here in order to give a quick impression of the overall structure of the thesis.

Chapter 1 - Introduction

You are here.

Chapter 2 - Library dependencies

This chapter discusses dependencies in software libraries. It shows that according to the Open Web Application Security Project (OWASP) [13], most developers don't keep track of the libraries their systems depend on, and makes an example of how a complex system with a lot of dependencies can be heavily impacted by a vulnerability in a dependency. Indirect dependencies and homogeneity in software are examined, and we look at the relation between defects and system size. Lastly, we also take a look at how build processes and modern package managers can worsen the problem of complex dependency graphs, and detail cross-build injection, an exploit class that targets the build process.

Chapter 3 - Infrastructure dependencies

In this chapter, we will look into dependencies on a more "fuzzy" level: at the level of entire systems and infrastructures. We will see that dependencies can limit a system's level of security. We will look into how complex systems and vague dependencies make it harder to know the dependency graph, and how modern cloud-based systems and service-oriented architectures can further exacerbate this problem.

Chapter 4 - Trust in dependency relations

Dependencies create a need for trust. We examine the web of trust that exists on the Internet between the different networks, and how trust is diluted in large dependency graphs that contain indirect dependencies. We also look at trust in situations where there is limited knowledge of the entity you trust. Finally, we look at how trust in non-professionals can be misplaced, and try to answer whether or not developers have an informed level of trust in dependencies.

Chapter 5 - Case studies

We perform case studies of “real world” systems, and examine how to model and analyse systems. The chapter describes a few of the available tools to model dependency graphs at various levels of abstraction, and describes a modelling tool written for this thesis to simulate downtime events in dependency graphs.

We will evaluate the dependency management and dependency graphs in Dynamic Presentation Generator and Netflix, and the `npm` package manager with its dependency explosion. Finally, we will look more into build systems and cross-build injection and examine the safeguards the build systems have to protect against this type of attack.

Chapter 6 - Solutions

We discuss attempts currently being made at solving the problems outlined in this thesis, and suggest possible extensions to these methods. Finally, we also suggest some original solutions and possible mitigations.

Chapter 7 - Conclusions and summary

We round off this thesis with conclusions and a summary, and then we suggest further work that could be done given more time, as well as open research questions.

Chapter 2

Library dependencies

People think that computer science is the art of geniuses but the actual reality is the opposite, just many people doing things that build on each other, like a wall of mini stones.
– Donald Knuth

In this chapter, we will look at dependencies in software libraries. We will look closer at indirect dependencies, and how package managers have contributed to large dependency graphs and made indirect dependencies more common. Finally, we will attempt to illuminate the connection between source code size and number of vulnerabilities, and how cross-build injection can be used to inject hard-to-detect vulnerabilities into the final product.

2.1 What are library dependencies?

A **library dependency** (or **component dependency**) is a dependency on a software library or component [14]. This component can be a large framework that does almost everything, or it can be a small class for converting between time formats. It can be written by a large organisation, or a single developer in her spare time. It can be well documented or not documented at all.

Introducing a component dependency into a system can be immensely useful. Instead of having to implement every single component of the system, the programmer can import a suitable component or library to do the work. Developers don't have to reinvent the wheel every time, and can spend their time working on the domain logic and figuring out the major important points instead of diving head-first into complex algorithms that they don't necessarily have the expertise to properly understand, or waste time on tedious details. Offloading work to prewritten components allows developers to work faster when it matters, and go back to improve or replace components later if necessary.

A component dependency is also a security dependency. Allowing external code to run in an application allows for exploitation of vulnerabilities contained in the imported code. This means that the security of included components has as much impact on the systems security as ones own code, and should be treated as such. There are attempts to sandbox or otherwise limit the effects of vulnerabilities in dependencies [15, pp. 19–21], but they only attempt to solve the set of direct vulnerabilities, not the fact that unexpected results from the imported code can cause vulnerabilities in the in-house code, such as a time calculation library giving out wrong dates, or broken random number generators [16].

It is hard to find solid data on the number of dependencies in the average "modern" software project, but a survey of 473 software projects done by White Source [17] indicates that the average project has 64 open source dependencies. Note that this only includes open source components; the total number of dependencies is probably higher. In my experience, this estimate is probably not far from the truth (as a lower bound), and I have seen projects with hundreds of dependencies.

As we will see in Section 2.5, this many dependencies can be the cause of problems simply due to the total number of lines of code involved. It isn't unusual for small to medium-sized projects to have substantially more code residing in third-party libraries than in the in-house code. Many software development teams don't put in the effort needed to keep track of all of the dependencies, leading to a mess of out-dated libraries, including libraries with known vulnerabilities [13, 14].

Keeping libraries up-to-date brings its own set of challenges, as trying to stay on the "bleeding edge" and only use the most recent releases often incur high risks of instability and a lot of work with reacting to possible changes in the library. Staying too far behind can often mean being vulnerable to issues already fixed in newer releases, or having to manually patch fixes to stay secure. Both ends of the release cycle bring challenges; pain and uncertainty on the bleeding edge, and vulnerabilities on the tail end.

A quick test: analysing Norwegian banking websites

To test how well library dependencies are handled in practice, I performed a quick evaluation of Norwegian Internet banking websites in July 2013.

Because I didn't have access to the banks' server-side source code, I simply checked the versions of JavaScript libraries in use on each site to see if they were kept up-to-date, and cross-referenced with public vulnerability databases and security bulletins to find if the libraries had any known vulnerabilities. The banking sites surveyed was DnB, Nordea, Skandiabanken, Sparebank1 and Sparebanken Vest. See Table 2.1 for an overview of the findings.

	jQuery	jQuery UI	swfobject	dojo
`dnb.no`	🟧 1.7.1	🟧 1.8.18	-	-
`nordea.no`	🟧 1.8.2	-	🟩 2.2	-
`spv.no`	-	-	-	🟥 1.3.2
`skandiabanken.no`	🟥 1.3.2/1.5.1	-	-	-
`sparebank1.no`	🟧 1.7.1	🟧 1.8.18	🟧 1.4.4	-

Table 2.1: *Versions of major Javascript libraries used in Norwegian banking websites. Red marks versions with known vulnerabilities, orange are out-dated versions (older than the next-newest version), and green are up-to-date versions.*

As can be seen from the table, none of the sites had the most recent version of jQuery, and Skandiabanken even used a version[1] that dated back to 2009, with a known vulnerability [18].

It should be noted that all the sites using jQuery also used multiple jQuery plugins of varying quality. These are open source plugins, and some are authored and maintained by a single person. This could be risky if the code is not verified or analysed internally. Some of the plugins were very old (as old as 2008), indicating a lack of maintenance.

Sparebanken Vest (`spv.no`) used a severely out-dated version of the Dojo Toolkit [19], version 1.3.2, released in July 2009. This version has several known vulnerabilities, including Cross-Site Scripting (XSS) and open redirect vulnerabilities. See Common Vulnerabilities and Exposures (CVE) advisories CVE-2010-2273, CVE-2010-2274, CVE-2010-2275 and CVE-2010-2276 [20] for more information. I have verified that both Skandiabanken and Sparebanken Vest are not vulnerable to the known vulnerabilities described here, although this is not because of any active mitigation or workarounds on their part.

It is important to emphasise that this informal and crude test cannot be used to draw any hard conclusions. However, the test can be used as an indication of how library dependencies are handled in the rest of the

[1]Skandiabanken used two different versions of jQuery, version 1.3.1 for the "public," non-authenticated website, and version 1.5.1 for the authenticated "secure" banking site.

banks' server-side code base. Client-side code typically doesn't have the same security consequences as server-side code, but as can be seen from the table, some of the libraries used had several known vulnerabilities.

2.2 Indirect dependencies

The Open Web Application Security Project (OWASP) organisation publishes a Top 10 list over the most important web application vulnerabilities every few years. The most recent, OWASP Top 10 – 2013 [21], features the vulnerability class "A9 Using Components with Known Vulnerabilities" as one of the most important classes of web application vulnerabilities in 2013:

> *Virtually every application has these issues because most development teams don't focus on ensuring their components/libraries are up to date. In many cases, the developers don't even know all the components they are using, never mind their versions. Component dependencies make things even worse.*
>
> *OWASP Top 10 – 2013* [13]

The last sentence is quite interesting, as indirect dependencies (dependencies that has dependencies of their own) really do complicate the issue. Indirect (or **transitive**) dependencies are dependencies that are outside the control of the system developers. When each library include its own set of dependencies, this can lead to a dependency "explosion," where each library adds more dependencies, which in turn adds more dependencies and so on.

A recent security incident with Spotify's account creation system shows just how hard it is to spot issues arising from indirect dependencies:

> *So changes in the standard python library from one python version to the next introduced a subtle bug in twisted's* `nodepre.prepare()` *function which in turn introduced a security issue in Spotify's account creation* [22].

A change in the standard python library caused a hard-to-find bug in a third-party library function that in turn introduced a unicode-related vulnerability in Spotify's account management system. This shows that indirect dependencies and dependency chains can lead to situations where a dependency two steps removed can still introduce security vulnerabilities.

2.3 Automatic package management

Modern package managers and build systems automatically resolve and download dependencies, and make it vastly simpler to add a new dependency to a project. Modern package managers like `npm` (JavaScript), NuGet (.NET) or Maven (Java) are widely used in their respective ecosystems, largely due to their ease-of-use compared to manual handling of dependencies.

While package managers can help resolve and download dependencies, there are still a lot of tasks left to manage. Making sure everything is as up to date as it can be, handle potential unwanted changes in the indirect dependencies and verify that there are no known vulnerabilities in any dependency, direct or indirect. This can be hard to do right, and with large dependency graphs, the graph will also change more often, demanding yet more resources to manage it.

Modern package managers simplify adding and resolving dependencies, but do not make it easier to keep track of changes in the dependency graph, vulnerabilities in packages, or what authors and packages are trustworthy or not. This leads to a point where most developers lose track over the dependencies they have in their code. Package managers also encourage making small packages with less duplicated functionality, which again leads to each package having more dependencies, although the total application size doesn't necessarily increase.

2.4 Homogeneity in library usage

Software homogeneity in a network is highly conducive to spreading of malware [23]. A network where all systems are running the same operating system or other piece of software is a network where malware spread very easily, as it can exploit the same vulnerability in all nodes on the network.

As an example, 54.8% of *all* web sites depend on the Javascript framework jQuery [24, 25]—including mega-sites such as Amazon.com, Microsoft.com, Wikipedia and Tumblr—and the trend is rising. A vulnerability in jQuery would impact over 50% of the world's websites, and as the major sites could act as hubs in the network, malware distributed through an exploited jQuery vulnerability would spread easily and fast.

Most Javascript libraries benefit greatly from being served from a Content Delivery Network (CDN), mostly due to the increased chance of the user already having the library in their browser cache, thus decreasing loading times and bandwidth usage. The probability of a library already being in the users cache is proportional to the number of sites that use that particular

CDN, so a primary reason for choosing a particular CDN is the number of users. This causes a "rich-get-richer" effect, where the "richest" CDN grows fastest, leading to a network dominated by hubs. This is known in network theory as preferential attachment [26].

Google Hosted Libraries [27, 28] is used in 14.9% of all websites to serve Javascript libraries, and is used by 90.9% of all sites that use a CDN for Javascript. Consequently, Google's CDN has the potential of being a hub that can be used to efficiently spread malware.

2.5 Lines of code and vulnerabilities

Source Lines of Code (SLOC) is a metric given by the number of lines of code in the source code of a program [29]. Lines in the source text containing only whitespace or comments are typically excluded. Additional metrics based on SLOC has been proposed, such as XLOC (executable LOC) or LLOC (logical LOC), but aren't as widely used. Automatically generated code and library code are also commonly ignored when counting SLOC, as the intent typically is to count the lines of code written by a developer or team, not by the development environment.

As SLOC has historically been misused by managers as a measure of programmer efficiency and productivity,[2] and is in itself somewhat vaguely defined, SLOC is often berated as a useless metric. As a measure of complexity, on the other hand, lines of code are just as useful as cyclomatic complexity [30]. Cyclomatic complexity[3] is a useful metric that carries a much higher computational cost, but is proven to have about the same predictive power. SLOC can be a useful metric to give a ballpark estimate of the magnitude of a development project, although the precision is low.

Number of defects are strongly correlated with SLOC [31]. The defect count is a *linear* expression of the total SLOC [32], meaning that a larger system typically has more defects than a smaller system. To keep the number of defects small, system size must be strictly controlled.

"My point today is that, if we wish to count lines of code, we should not regard them as 'lines produced' but as 'lines spent': the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger."
– E. W. Dijkstra [33]

[2] *"Measuring programming progress by lines of code is like measuring aircraft building progress by weight."* – Often attributed to Bill Gates

[3] Cyclomatic complexity is the number of linearly independent paths through a program.

According to the static analysis vendor Coverity, a typical program has a defect density of 0.1 to 20 defects per 1000 SLOC (kSLOC) [34]. Static analysis of the Linux kernel in 2004 showed 0.17 bugs per kSLOC [35].

Based on defect and vulnerability data from different versions of the Microsoft Windows and Red Hat Linux operating systems, Alhazmi et al. [36, 37] found a relation between defect density and vulnerability density. As could be expected: more bugs, more vulnerabilities. The ratio between defect density and vulnerability density was found to be in the 1–5% range.

A typical small or medium-sized system will have a large amount of code residing in external libraries. In some cases there are way more SLOC in library dependencies than in the code that makes up the system, written by the in-house development team. This makes internal development guidelines and processes less effective than if everything was written in-house.

2.6 Licences

Large dependency graphs can make license compliance harder. If a leaf node changes its license, or a new node gets added to the graph with an unacceptable license, you have to remove and replace this node. With a large graph, the control you have over each node diminishes, as the number of indirectly attached nodes increases.

This is especially hard with open-source dependencies, as some open-source licenses are “viral,” and require any system redistributing their code to be licensed with the same conditions.[4] Examples of such viral licenses are the GNU General Public Licence (GPL) or Creative Commons Attribution-ShareAlike (CC BY-SA). A library far out in the dependency graph with a viral license like GPL can force the entire system to be licensed as GPL. Conflicting licenses in different libraries can also occur, which would prevent any legal distribution of the system at all [17].

An example of how potential licensing conflicts are handled in the NuGet package manager can be seen in Listing 2.1. NuGet handles potential conflicts by releasing Microsoft (the original developer of NuGet) of any liability related to licence issues, and only warns the user of potential issues. NuGet doesn’t enforce any licence restrictions.

```
Each package is licensed to you by its owner. Microsoft is not responsible
  ↪  for, nor does it grant any licenses to, third-party packages. Some
  ↪ packages may include dependencies which are governed by additional
  ↪ licenses. Follow the package source (feed) URL to determine any
  ↪ dependencies.

Package Manager Console Host Version 2.7.40808.167

PM> Install-Package Microsoft.AspNet.WebApi.Client
Attempting to resolve dependency 'Microsoft.Net.Http (>= 2.0.20710.0)'.
Attempting to resolve dependency 'Newtonsoft.Json (>= 4.5.6)'.
Installing 'Microsoft.Net.Http 2.0.20710.0'.
You are downloading Microsoft.Net.Http from Microsoft, the license
  ↪ agreement to which is available at http://www.microsoft.com/web/webpi
  ↪ /eula/MVC_4_eula_ENU.htm. Check the package for additional
  ↪ dependencies, which may come with their own license agreement(s).
  ↪ Your use of the package and dependencies constitutes your acceptance
  ↪ of their license agreements. If you do not accept the license
  ↪ agreement(s), then delete the relevant components from your device.
Successfully installed 'Microsoft.Net.Http 2.0.20710.0'.
```

Listing 2.1: *NuGet handles licences in dependencies by issuing a warning, but will not detect or handle any licensing conflicts.*

2.7 Cross-build injection

Cross-build injection (XBI) is a relatively novel type of code injection attack, first described in a white paper by Fortify Software in 2007 [38]. XBI exploits the fact that modern build processes often fetch dependencies from remote servers in an insecure manner [38, 39]. Modern automated build processes with library package managers, such as those offered by Maven, NuGet, `npm` or RubyGems will resolve and fetch declared library dependencies on build-time, by downloading them from a central repository or a local cache. See Figure 2.1 for an overview of the typical build process with package managers in a modern application.

If the process of fetching these dependencies from the remote server is insecure, this opens up for Man-in-the-Middle (MITM) attacks, where an attacker can inject arbitrary code, and even run code directly on the computer that initiates the build. The remote server could also be compromised (like what happened to a Sourceforge download mirror in 2012 [40]), and if there is no verification of server integrity, the compromised server would be able to inject malware into the build process and affect the resulting binaries.

[4] With AGPL, using the code in a web-based system is regarded as distribution.

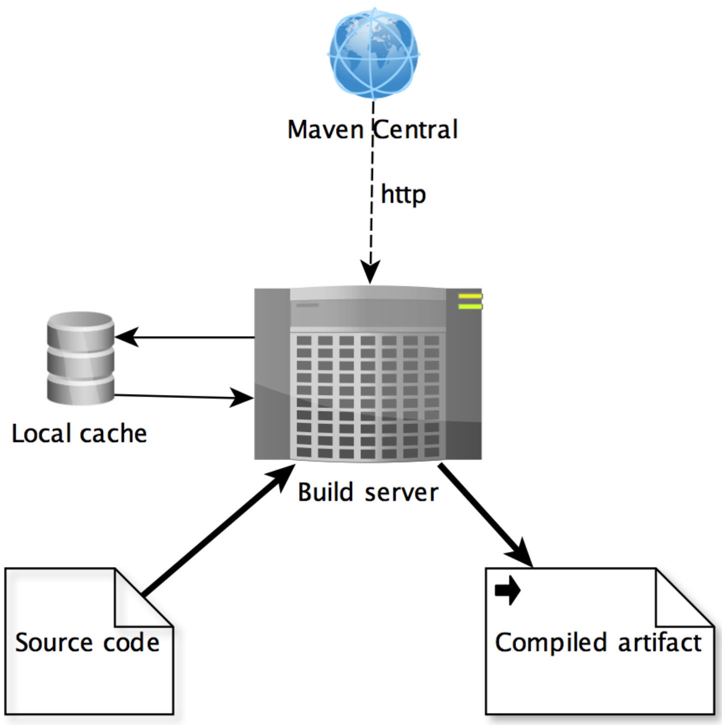


Figure 2.1: *Typical flow in a Maven-based build process. Note that the only external node in the diagram is Maven Central, the central Maven package repository. XBI attempts to exploit the dotted edge to inject code.*

What's so scary about XBI is that the attack is designed to change the produced binaries, which can then be distributed to thousands of users (or put in production on a website) before the attack is discovered. The impact is potentially very large. Discovery can also be very hard, as it is hard to detect if the source behind a binary has been tampered with [41].

The attack window for successful XBI through a MITM attack is significantly increased when using package managers, as each developer machine and build server fetches the dependencies from the web at least once (and sometimes when starting each build) [38]. The number of dependencies in use is also larger, so the total number of requests made are significantly larger, increasing the window of opportunity.

Continuous integration and continuous delivery [42], where the application is automatically compiled, tested and deployed in a continuous manner by a build server, is getting traction as a best practice for agile software development, as it allows for simpler integration and faster deployment. The risk and potential impact of XBI in this kind of software development process is higher, since a successful attack can potentially get into production-ready code and even reach the production environment without human intervention.

2.8 Dependency management systems

Earlier in this chapter, we saw how important it is to properly manage the dependencies of a system. This is a hard task that could be simplified by tools that assist in ensuring licence compliance, handling mapping of explicit library dependencies and notifying developers about new releases and known vulnerabilities.

I haven't been able to find many of these systems. There are a lot of systems that claim to do dependency management, but in reality are only package managers that also performs dependency resolution (traverse the dependency graph to make sure dependencies are only included once) and handles version requirements ("use version `x.y` of dependency `z`") as a part of their functionality. There exists a select few static analysis tools such as Veracode Analytics [43] that can perform static analysis and generate reports of external dependencies and any known vulnerabilities.

The UK Centre for Protection of National Infrastructure tasked the standards organisation Open Group with writing a dependency modelling standard [44]. The result were published in December 2012 [45] as the Dependency Modelling (O-DM) Standard [46]. The dependency modelling system iDepend [47] promises to be compatible with this specification and assist in solving and mapping dependency issues in software, processes and infrastructure. As of September 2013, however, iDepend is still in alpha, and the website has been down for the last few months, indicating that the project has been abandoned.

To summarise, the current state of dependency management systems is that there are no systems that satisfy our needs. Whether or not this functionality is best taken care of in a dedicated dependency management system, or if it should be handled directly in the package managers and development environments is also up for debate. On the one hand, a dedicated system would be able to handle different environments and ecosystems, but implementing this in package managers may be easier and faster, as a package manager is more closely tied to the development process.

Chapter 3

Infrastructure dependencies

The central enemy of reliability is complexity.
– Dan Geer et al. [48]

In this chapter, we will take a look at dependencies in ICT infrastructures. We begin with an overview, look at dependencies in the Internet "web of trust," and then look more specifically at dependencies in Service Oriented Architectures (SOAs) and cloud computing environments.

3.1 What are infrastructure dependencies?

Dependencies are not limited to well-defined low-level relations in source code. When looking at entire systems, services and infrastructures, we also find dependency relations to the entities surrounding the system. A system can depend on basic services, like stable power, networking connections and cooling, as well as higher-level services like external APIs, cloud platform management systems, payment processors and more.

As with software library dependencies, a dependency graph of the system can be constructed and analysed, hopefully providing more insight into the behaviours and vulnerabilities of the entire system. For the purposes of this chapter, the concept of a **dependency** can also be understood in a slightly different way: there exists a dependency relation between two entities when the state of one entity is correlated to the state of the other. This extended definition frees us from the requirement that a dependency must be well defined, and allows us to reason about vague and even potentially undefined dependencies.

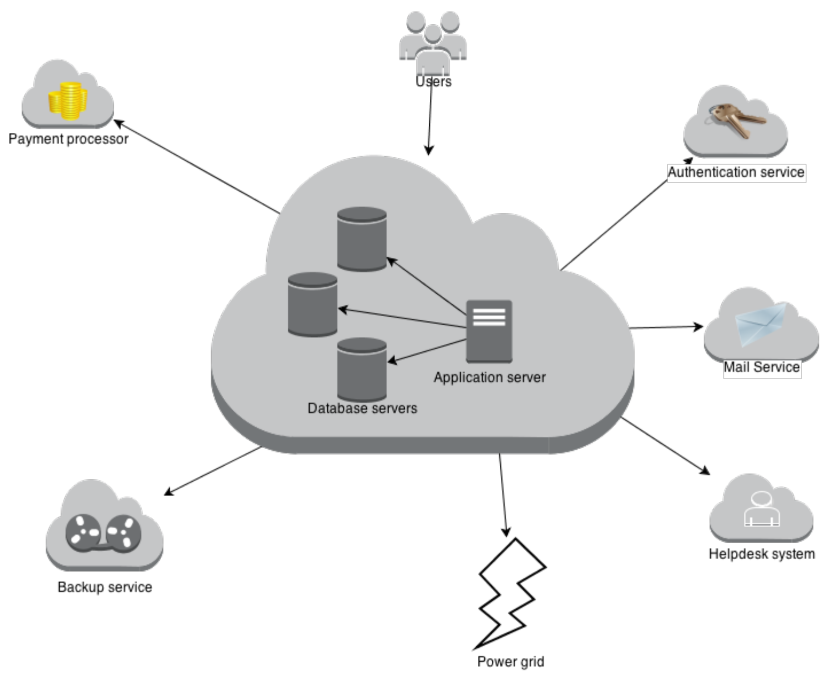


Figure 3.1: *Dependency graph of the infrastructure in and around a typical cloud-hosted application. The contents of the cloud is what is typically thought of as "the system", but system-impacting incidents can occur in all elements seen in the illustration.*

A typical modern ICT system has many dependencies to its surrounding infrastructure. An example of a typical cloud-hosted application can be found in Figure 3.1. As with library dependencies, it makes perfect business sense in many cases to outsource parts of a system to a third party. Unlike library dependencies, however, there is a risk for outages and information leakage, and many applications are not properly isolated from such failures. With third party services, there exist legal techniques such as Service Level Agreements (SLAs), to provide legal recourse in case of failures, although SLAs do nothing to prevent the failure in the first place. After-the-fact recourse is often too little too late.

These higher-level dependencies are often weakly defined and more vague and non-technical, and thus tend to be shunned by developers and academics. The dependencies are still important, especially when considering large, national systems and infrastructures, and the very large impact of incidents in these systems. Some of the systems are deemed "too big to fail," and should be scrutinised closely. In Norway, these systems include the payment systems run by Nets and Evry; the communications backbone by Telenor; the national power grid; the eGov system Altinn; and the mobile phone networks of Telenor, NetCom and Network Norway.

Incidents in such important, complex systems can cause major problems [49], disrupt important societal functions and impact large parts of the population. This was exemplified when the 2011 winter storm Dagmar knocked out a lot of mobile base stations in Norway. People were stranded without power and unable to call the emergency services. The mobile network operators blamed the utility companies, and claimed the problem was that the base stations lost power, and that the utilities companies were unable to restore power before the battery backup gave out. However, a report published by the Norwegian Post and Telecommunications Authority in January 2012 revealed major vulnerabilities in the way extreme events are handled by mobile network operators [50]. The operators weren't prepared to handle events much outside normal operations, including securing critical points of failure in their central infrastructure.

3.2 Complex systems

A complex adaptive system is defined as a system that has "a large number of components or agents that interact and adapt or learn" [51]. Examples of complex adaptive systems include the Internet, the brain, the stock market, but also sufficiently large ICT systems, especially as human agents acting in collaboration with the system can be considered part of the system. We argue that modern public cloud infrastructures also are complex adaptive systems, as they feature self-regulation, a vast number of independently acting entities, and non-linear interactions that can cause cascading failures.

Failures in complex adaptive systems typically have complex causes [6], often with multiple faults colluding to cause the failure. As an example, we take a quick look at the post-mortem from the Microsoft Azure service disruption on leap day 2012 [52].

An outage occurred on leap day 2012, lasting for over 10 hours. In Microsoft's post-mortem and other media coverage of the outage [53], the problem was attributed to a simple leap year related programming error, but the actual problem was the fact that Microsoft's Azure is a complex adaptive system, with a large number of semi-autonomous systems and complex interactions. This resulted in a tightly interleaved systems architecture where a fault in one component could propagate and take down large parts of the system.

Complex systems can have complicated dependency graphs, and can even have dependency graphs that cannot be accurately defined. The boundaries of the system often cannot easily be described, and the dependency graphs may change as the systems adapt. Accurately monitoring and controlling dependencies in complex adaptive systems is an open research question.

Traffic systems are interesting real-world examples of complex, loosely connected systems. The systems are complex adaptive systems, for the most part due to the adaptive and convoluted behaviours of the drivers interacting with each other in unpredictable ways.

In Bergen on August 30, 2013, a truck leaked about 50 litres of hydraulics oil onto the outbound lane of the northern main road. To ensure traffic safety, police had to shut down traffic for a while while they washed away the oil. The resulting traffic jams lasted from 09:00 to 19:30, and caused slow-moving or jammed traffic in and out of the northern, western, eastern, and southern main roads [54].

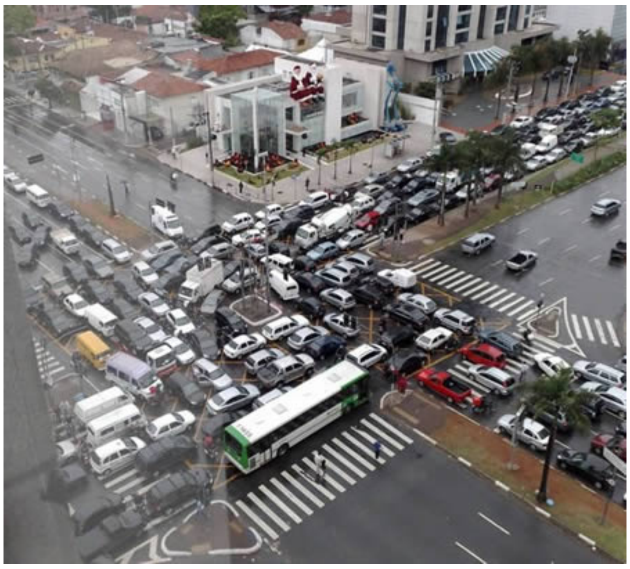


Figure 3.2: *A complex adaptive system gone terribly wrong.*

This traffic incident is a good example of how an infrastructure with insufficient over-capacity can experience cascading failures. Once certain parameters are outside the normal operating range, inter-dependencies between different parts of the system can affect each other in hard-to-predict ways. Undiscovered dependencies in infrastructures can cause problems way outside what is thought to be the location of the failure itself. In the traffic incident in Bergen, even traffic going in the *opposite* direction of the accident spot was slowed to a grinding halt by the fact that other deadlocked streams of traffic were blocking roundabouts and intersections for outbound traffic, as seen in Figure 3.2.

3.3 Vague and indirect dependencies

Dependencies on elements surrounding a system can be ill defined, non-technical and vague. The relations can be between weakly connected entities, and thus be hard to discover. By employing the expanded understanding of a dependency as stated in the chapter introduction, we also include highly correlated systems. By including such non-obvious dependencies, we can uncover new dependencies and also reason about dependency relations that are not as clear-cut and well defined as in the previous sections of this thesis.

As an example, let's examine the dependency graph for the Norwegian identification system BankID. The user-facing aspect of BankID is a Java applet [55]. As such, it depends on Oracle, who owns and maintains Java.

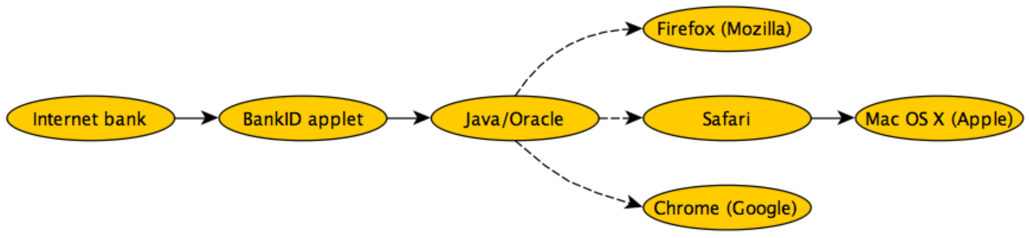


Figure 3.3: *A dependency chain within the BankID dependency graph, showing the indirect and ill-defined dependencies with regards to interactions with browsers' security features.*

This is an obvious and well-defined dependency relation. BankID has to trust that Oracle stewards Java properly. But Java applets are run within a Java applet plugin in the browser, and modern web browsers have security features that make them able to block insecure or out-dated plugins at will [56, 57]. So, when vulnerabilities are discovered in Java, browser vendors (and in the case of Safari and Internet Explorer, OS vendors) are able to disable the plugin to ensure the safety of their users. In the end, the availability of BankID depends on browser/OS vendors. This can be thought of as a dependency chain hidden within the larger dependency graph, as seen in Figure 3.3.

In January 2013, a series of critical Java applet plugin vulnerabilities were discovered, published and exploited in short order [58]. Oracle released an updated version, and claimed to have fixed the vulnerabilities, but Apple still blocked the Java applet plugin [59] from their operating system, as Apple did not have confidence that the release had fixed all the known vulnerabilities. This affected all of BankID's users who used Apple computers.

Oracle and Apple are large multinationals, and the "small" BankID does not have any leverage over them. In a situation like this, they just have to sit and wait. The only thing BankID could do was to inform users about the situation, suggest temporary work-arounds, and wait for the situation to be resolved [60].

In the context of dependencies in infrastructures, such indirect (transitive) dependencies are problematic, as they cannot be effectively controlled. The case with BankID also demonstrates that the dependencies surrounding a system can be less well defined and "clear-cut" than intuitively thought. Dependencies can exist in the entire ecosystem surrounding the system—legal, social and technical—and attempts at discovering and modelling dependencies in infrastructures must take these aspects into account.

Other challenges loom on the horizon for BankID and their Java applet-based system, as both Google and Mozilla have announced their intent to block many Netscape Plugin-API (NPAPI)-based plugins (which includes the Java plugin) from their browsers as of January 1st 2014, and then completely block all NPAPI-based plugins by the end of 2014 [61]. While BankID has presented a roadmap to transition to a applet-less implementation [62], the timeframe outlined by BankID for this transition is too long to avoid these problems.

3.4 Cloud computing and Service-Oriented Architectures

The interesting thing about cloud computing is that we've redefined cloud computing to include everything that we already do.
– Larry Ellison, Oracle CEO [63]

Cloud computing is—as the above quote makes clear—a term that has been widely misused in the past, to the point where it almost makes no sense anymore. For this thesis, however, we will define cloud computing as a massively distributed service for hosting applications and systems, providing virtually infinite scaling.

In this thesis, we will mostly talk about two kinds of cloud computing services. Platform-as-a-Service (PaaS) clouds, where the cloud provider delivers a platform within which the customer can build a system, using the provided APIs and supported programming languages, and Infrastructure-as-a-Service (IaaS) clouds, where the cloud provider provides the infrastructure (virtual servers, networking, and some way to manage it), but where the customer has full control over the runtime environment [64]. An example of a publicly available PaaS cloud is Google App Engine [65], while Amazon EC2 and Microsoft Azure are examples of public IaaS clouds.

Cloud computing services are conducive to systems with many external services (and thus having many infrastructural dependencies), as a system designed to run on a cloud computing platform has to be designed for loose coupling between components. As a response, many services have popped up to deliver parts of a typical cloud-based system (such as logging, monitoring, provisioning etc.), which easily can be integrated into an existing system. Due to scaling requirements, many cloud-based systems have gravitated towards a Service-Oriented Architecture.

Service-Oriented Architecture (SOA) is a software architecture pattern, where the system is divided into self-contained and separate services that each provide a part of the application functionality [66]. Together, the services provide the full functionality of the application or system.

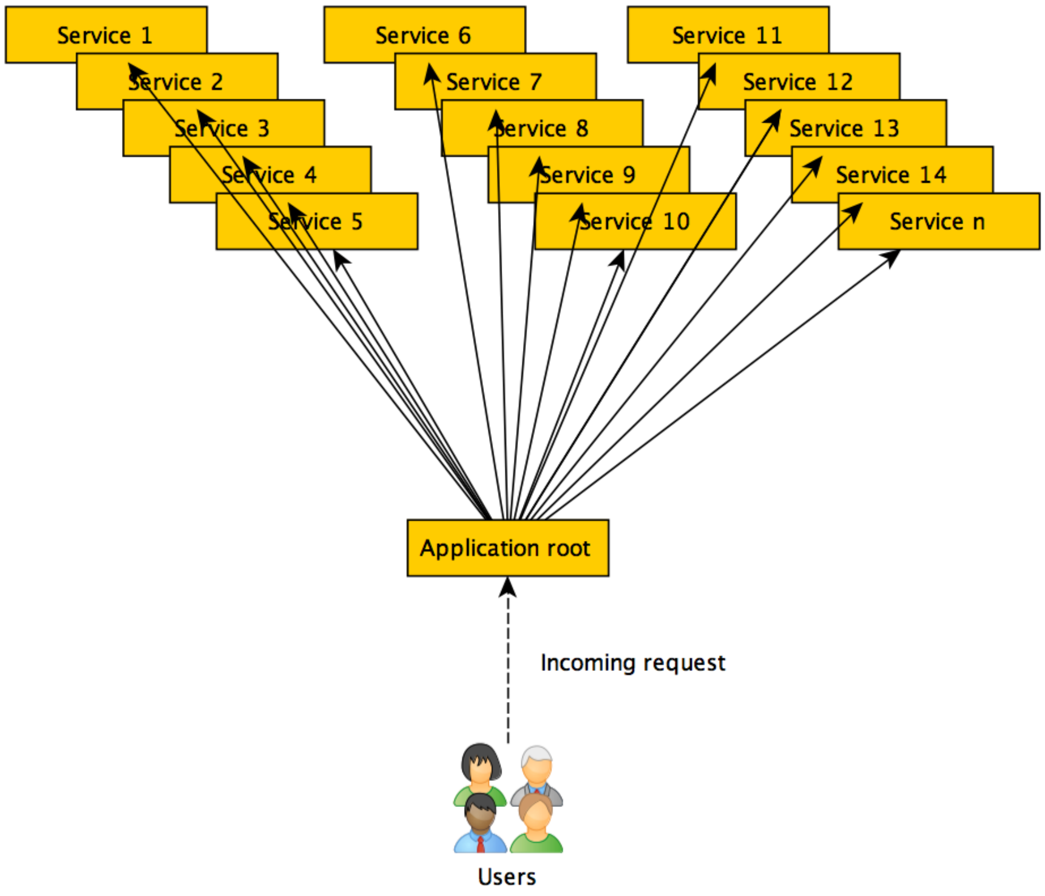


Figure 3.4: *Birds-eye view of an imagined web-application with a service-oriented architecture.*

The benefits of SOA are many (and beyond the scope of this thesis), including providing looser coupling between the different parts of an application, simplifying redundancy and making it easier to scale a system to higher loads by adding servers (scaling out or "scaling horizontally"). It can be seen as the architectural counterpart to programming with modules and components, as both strive to avoid unmaintainable monolithic applications by dividing it into small, maintainable, loosely coupled modules or services. Figure 3.4 shows an illustration of an application with typical service-oriented architecture.

The services in a SOA don't all have to be local services within the system, but can also be external outsourced services. In recent years, this has become more and more popular, as web-centric API technologies like REST and SOAP have matured, and cloud computing made it possible to cheaply deliver services to a large customer base without large up-front costs. This is evidenced by the multitude of "Web 2.0" services providing everything from payment processing and subscription management to application monitoring and media transcoding.

There are some potential problems with SOA—although it must be emphasised that the benefits typically outweigh the problems, especially compared to "traditional" monolithic applications. SOA moves the inherent complexity of a system from the application into the structure of the dependency graph, in much the same way that "clean" software development (through the Single Responsibility Principle [67]) moves complexity from the function to the call graph. The inherent complexity doesn't disappear, and one may argue that this complexity is easier to handle and understand when the logic is less distributed.

SOA introduces complexity in the network between services, as buffers, timeouts and queues are necessary to insulate against single failures, and avoid resource contention. If the system doesn't insulate against single failures, a single failure can take down the entire system, and the availability of the system is going to be unacceptable. If 30 services each have 99.99% uptime, the uptime of the entire system is going to be just $0.9999^{30} \approx 99.7\%$.

Insulation against errors is harder than most think, and can introduce significant headaches as the scale of a system increases. In the solutions chapter, in Section 6.2.4, we will discuss the solutions employed by the industry to avoid the potential problems mentioned in this section.

A system doesn't have to be fully cloud-based to be impacted by the issues described in this section. Modern startups (using what's jokingly called "trendy programming" or "cargo cult programming" [68]) often offload services that traditionally used to be in-house, like code hosting, build servers, mailing, billing, reporting etc., to save up-front costs on hardware, software licenses and expensive in-house expertise, even if the system itself is hosted locally.

Chapter 4

Trust in dependency relations

Trust, but verify.
– Ronald Reagan

In this chapter we look at trust in dependency relations. We start by examining what trust is, then look at trust as it appears in dependency graphs. We then examine trust in relationships with non-professional agents (or amateurs). Finally, we try to answer the question of whether or not developers have an informed level of trust in their dependencies.

4.1 What is trust?

Trust is a difficult concept to define, as it is both context-dependent and agent-dependent. Trusting someone when you have no choice (trust as despair) is vastly different than trusting someone you know really well (cognitive trust). In the same way, the trust between two equal individuals is different from the trust between unequal individuals, or between an individual and a corporation. For this reason, many differing definitions occur in the literature, based on different contexts.

The definition of trust we will use in this thesis is from Marsh and Dibben [69]:

> *"Trust concerns a positive expectation regarding the behavior of somebody or something in a situation that entails risk to the trusting party."*

An important element of trust is the acceptance of risk [69]. That is, by trusting someone else, there has to be some risk that you accept. Without any risk, there would be no need for trust. If there is a possibility to remove the need for trust, such as by adding redundancy, contractual guarantees or otherwise insulating against failures, it should be considered thoroughly, as it lowers or even removes risk.

In deciding to trust an entity, a person is forced to also trust that entity's choice in who to trust. This transitive trust may be a source of conflict, as there can be different opinions on the trustworthiness of third parties. This is why dependency graphs with multiple paths between nodes make it harder to remove the need for trusting an entity, as the entity may be a—possibly unknown or undeclared—dependency of another node. Discovery of such superfluous edges is critical.

When it comes to security dependencies, the accepted risk is very high, and there are not many ways to avoid trust, since the security of your system will depend on the security dependencies. As we have seen in Section 2.5, each added line of code brings a non-zero probability of containing vulnerabilities and thus a certain risk of negatively impacting the security of your system. In other words, a library dependency is a security dependency.

Cognitive trust is built from accumulated knowledge and experience [70]. If a service provider or framework developer has a bad history of security issues or downtime incidents, your level of trust will be lower. This is an iterative process, where the level of trust can be lowered by undesirable behaviour or incidents, and increased by long periods of stability.

In a software library dependency setting, there is often not much available knowledge about the person, organisation or company that is delivering the service or library. Thus, until any experience can be acquired, the trust has to involve a "leap-of-faith" step. Due to the nature of vulnerabilities, an insecure library can appear to be secure simply from lack of a proper security review. No evidence of vulnerabilities is not evidence of no vulnerabilities.

4.2 Trust in dependency graphs

To model trust in dependency graphs, one can visualise that trust "flows" outwards along the edges from node to node. Each node decides to trust in a set of nodes, but has to trust these nodes' choice in who they trust. The result is that a choice to trust a few nodes quickly inflates to a choice of much larger consequence when the entire graph is taken into account.

Comparing dependency graphs to the **trust chains** used in systems like SSL/TLS, we immediately see major differences. In a trust chain, the concern is the length of the chain. That is, how long do we have to walk until we reach a trusted node? In dependency graphs, we navigate the graph the

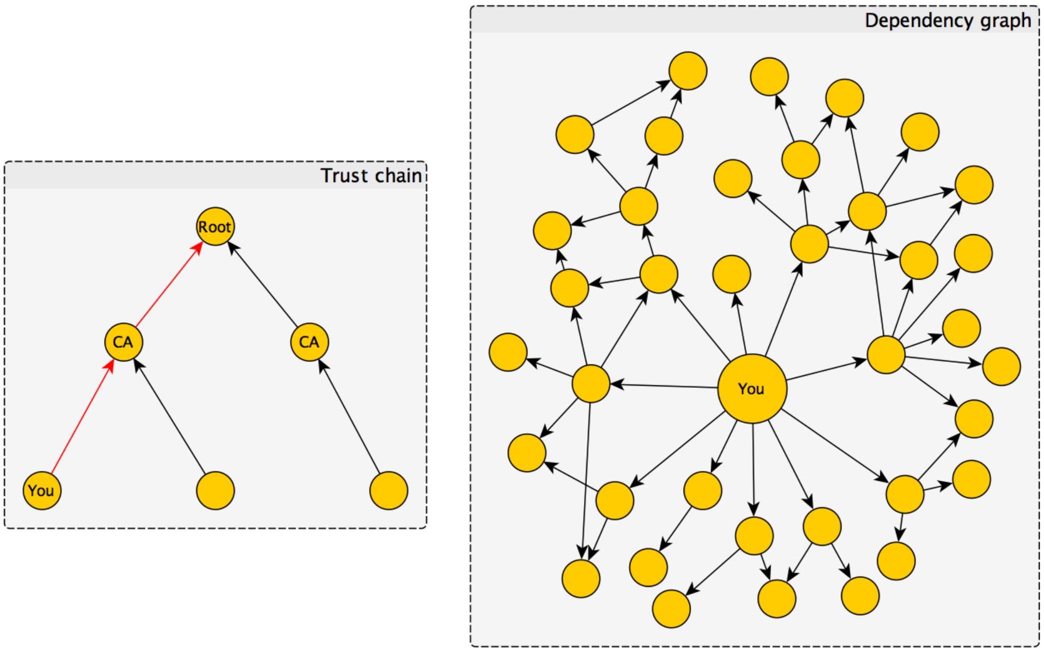


Figure 4.1: *Comparison of the standard SSL "trust chain," and trust dilution in a dependency graph.*

other way, from the trusted node and outwards. The problem here is not one of depth, but of total graph size. See Figure 4.1 for a comparison.

The concept of "trust dilution" [71] is applicable here, where the strength of a trust relationship must be weakened if it passes through a large set of intermediaries or when it is spread out over a large number of nodes. The massive spread of a dependency graph leads to a need to trust a much larger set of nodes than in a dependency chain, leading to a more diluted trust.

4.3 The Internet web of trust

In this section, we will take a look at the "web of trust" that determines routing on the Internet, and how the complexity of this web can cause cascading failures, and how it has caused cascading failures in the past.

Reachability and routing on the Internet is controlled by the Border Gateway Protocol (BGP), which is a protocol for different Autonomous Systems (ASs)[1] to reach a consensus as to how traffic should be routed from network to network. For this reason, BGP is one of the most important protocols on the Internet, even though it is less known than most other protocols. In broad

[1]ISPs or other peering partners [72].

terms, each AS is assigned a unique ID (AS number), and can announce an IP prefix, along with a path comprised of the AS numbers that must be traversed to reach this prefix. Typically, an AS announces IP prefixes that they themselves own or that they provide a route to.

This system is based on trust between peers. At the peering point, an AS determines if it trusts the routes the other AS announces. ASs must themselves validate whether someone advertises a prefix they actually own, and if the route path makes sense. This is a "web of trust," which can be modelled as a graph of trust relationships — or dependency relationships. As each AS interfaces with a relatively small set of other ASs, the web of trust also has transitive (indirect) trust relationships, and a typical announced AS path has a length of about four hops [73].

This trust can be exploited, and honest mistakes and accidents happen. Since the BGP protocol is responsible for defining the routing and reachability of traffic, it can be used for low-level MITM attacks that are much harder to detect for end-users and network administrators than the more common DNS-based attacks [74]. BGP can also be used for Denial of Service (DoS) attacks, where traffic to a network or website is routed through networks that discard the traffic [75], effectively "blackholing" it.

Of the many incidents over the years [76, 77], two incidents in particular demonstrate how vulnerable BGP routing is:

In February 2009, a Czech ISP made a configuration error on one of their routers while attempting to de-emphasise a specific route. The configuration caused that router to announce an unusually long[2] routing path [78]. This triggered a buffer overflow in Cisco routers that caused them to regard the route as invalid [79], and disconnected and reconnected the session, causing routing instability and a storm of route updates. Other routers passed the invalid route along, so the announcement spread globally, and eventually cascaded into a global routing instability that lasted for about an hour. The instability peaked at 107780 routing updates broadcasted *every second* and 4.8% of all IP prefixes on the Internet suffered instability or outage. See Figure 4.2 for an overview of the instability by country.

In November 2012, an Indonesian ISP were making a configuration change to block access to Google from inside their network by "null routing" or "blackholing"—specifying a route with an invalid destination [80]. Through a configuration error, the announcement leaked to a peering partner, and spread across the Internet. This caused the route to Google to be routed to the Indonesian ISP's invalid network destination for about a half hour, effectively taking Google offline for about 3-5% of the Internet's users [81].

[2]A route containing over 256 AS numbers.

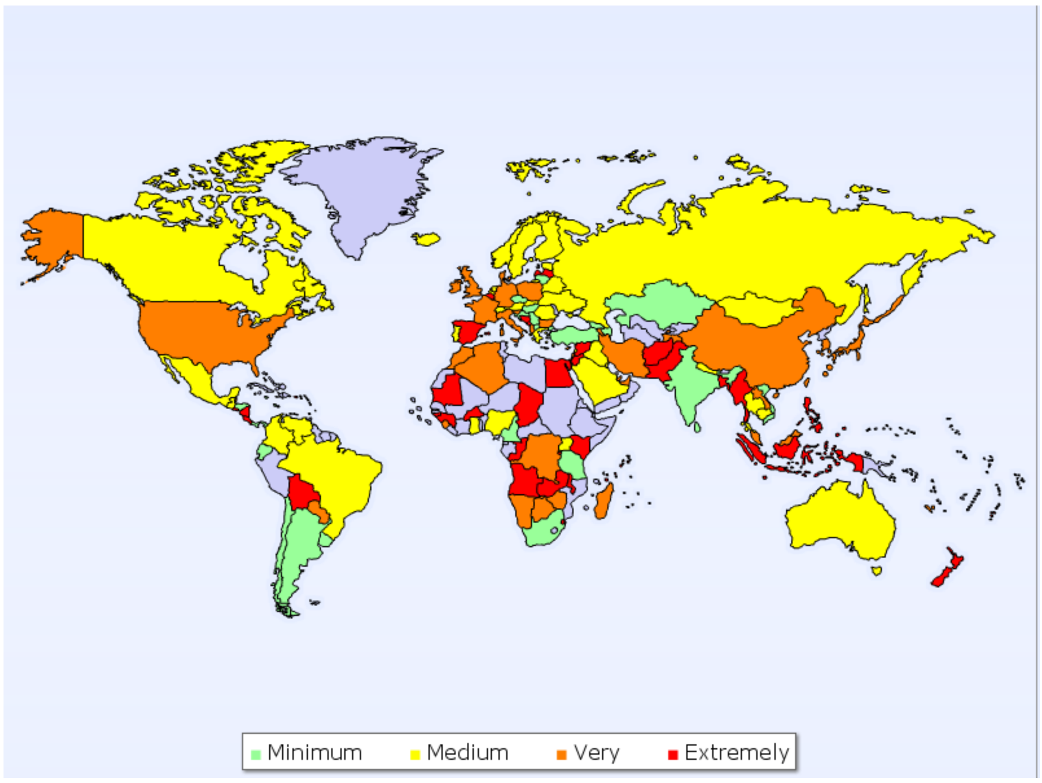


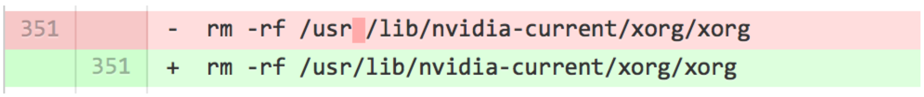
Figure 4.2: *Overview of the global routing instability caused by the 2009 BGP incident.*

A similar event happened in Pakistan in 2008, causing YouTube to be unavailable in large parts of Asia [82].

Since the system is based on a web of trust between peers, a peer halfway across the globe is implicitly trusted through three or four intermediaries, leading to problems with indirect (transitive) trust. As seen with the Google incident, this can adversely affect companies and individuals that have nothing to do with Internet routing directly.

The Internet and its routing protocols constitute a complex adaptive system, as it has a large number of independently acting agents[3] who change and adapt over time. This complex adaptive system can experience emergent and hard-to-predict properties, leading to the kind of cascading failures we have seen.

[3]Over 40 000 ASs per September 2013 [83].



```
-  rm -rf /usr /lib/nvidia-current/xorg/xorg
+  rm -rf /usr/lib/nvidia-current/xorg/xorg
```

Figure 4.3: *Can you spot the error being fixed here? The code was supposed to delete a specific driver-related folder deep in the file system, but instead removed the entire `/usr` folder due to a single misplaced space character.*

4.4 Trusting "amateurs"

With the blossoming of project hosting websites that encourage people to share and "fork" source code, like GitHub [84], Google Code [85] and Microsoft CodePlex [86], there has been an explosion in small open-source libraries and plugins. Some of these (like Google Code) only allow open-source projects, and some (like GitHub) charges money for private repositories, so a large share of the projects hosted on these sites are open source. All of the sites encourage sharing code and improving other peoples code, either through forking [87], pull requests or suggesting changes.

While this undoubtedly leads to more innovation, and hopefully lowers the bar for new programmers learning to write code, it also has led to somewhat of a mess. Reliable and well-managed projects live side-by-side with mismanaged projects, and with so little information available, it isn't always easy to determine how efficiently a project is managed.

Horrible examples of how this can go wrong can be found in the commit logs of several projects, where a misplaced space [88] (see Figure 4.3) or lack of scrutiny of a pull request [89] caused users to lose data on their systems. In these cases, the data loss was mostly limited to few users, and only on their private systems, but it is not hard to imagine that something like this could have happened on a larger scale, especially if it is a hard-to-find vulnerability.

It is interesting to read the responses from the developers. Security issues and vulnerabilities are sometimes put aside as "not important" or even funny. The developer of the `npm` package `n` [89] merged a pull request that accidentally deleted important system folders on many users' computers. When a bug report on this was filed, he added this comment to the report:

> *yeah it's kinda tough when you have 250+ OSS projects, inevitably some get messed up over time and I merge broken shit haha*
>
> – *GitHub user and `n` author @visionmedia* [89]

And the response a security researcher got for pointing out flaws in the security of the `npm` package registry: "Fork or gtfo." [90], essentially saying "If you think you can do it better, then go ahead. If not, get lost".

It isn't surprising that there are immature and unprofessional developers out there [91], everybody has to learn sometime, and a lot of people do this on their spare time. But there aren't any good ways of building an understanding of the level of professionalism of a library or program until something goes wrong.

The same problem can be an issue with infrastructures and services. It is really hard to judge the professionalism of a service just by the website, and in many cases, the website is the only reference point. A well-designed website can give the impression of a well-run organisation with many employees (and thus appear trustworthy), but may just be run by a few people in their spare time.

4.5 Do software developers have an informed level of trust in dependencies?

An **informed** (or **understood**) level of trust is based on experience and knowledge (cognitive trust), where the trusting individuals are sufficiently informed about the trusted entity, and thus understand their own level of trust and the basis of this trust.

Based on what we have discussed in this chapter, I don't think most software developers have an informed level of trust in their dependencies, because most software developers don't even know the full extent of their dependencies (as evidenced by the inclusion in OWASP Top 10 2013 [13]). A modern software application can have a very large dependency graph, making it difficult to gain sufficient knowledge on all dependencies.

When it comes to entire systems, it is possible to have contractual guarantees and SLAs that remove risk, and thus entirely remove the need to trust. But the trend toward a "feudal" Internet [92], with "take it or leave it" SLAs (as seen with many cloud computing providers [93]) makes this harder. The same problems relating to knowing your dependency graph still apply to systems, as we will see in later chapters.

Chapter 5

Case studies

An insightful quote found on the Internet can never be fake.
– George Washington

In this chapter, we will first evaluate different tools for modelling and mapping dependencies, and then perform a case study of a real-world system called Dynamic Presentation Generator. Further, we will look closer at package management and build systems, study `npm`, and explain the concept of dependency explosion.

5.1 Dependency modelling and mapping tools

Section 1.3 explained how graph theory can be used to build a dependency graph to describe and elucidate the neighbourhood of a software component. A tool to automate the process of constructing and analysing dependency graphs can be very useful.

In this section, we will evaluate some existing modelling tools that use graph-theoretic concepts to analyse source code. Finally, we will examine a set of modelling tools I developed to analyse and simulate the effects of dependencies in infrastructures.

There are many different ways to approach the problem of mapping and modelling dependencies, because there are many different contexts where mapping and modelling of dependencies can be useful. Different contexts require different approaches and solutions. The systems we evaluate fall into two different categories, based on the approach and scope of their analysis:

Mapping dependencies in source code

This approach attempts to map the dependencies that exist in source code, between modules, classes and functions. The method is primarily used in tools to map and illustrate tightly interconnected (coupled) code and to illuminate and enhance the architecture of the system. Tools using this approach mainly act as an architectural tool to guide system developers toward a simpler and more easily maintainable system architecture.

Generic dependency modelling

Dependency modelling focuses on modelling the interactions of dependencies in systems and infrastructures. The model is either manually constructed or automatically generated with a tool. The purpose of the model is to approximately simulate a system, and show which dependencies may be hard to remove, and thus which dependencies one should attempt to isolate or introduce fallbacks for.

5.1.1 Spoiklin Soice

Spoiklin Soice [94, 95] is a tool to map dependencies in Java source code. It can show dependency relations within a program at different levels, between components, namespaces, classes and even functions. It can generate a dependency graph and perform some automated analysis, such as finding cyclical dependencies. Cyclical dependencies exist when an entity has an indirect dependency on itself. As a change to an entity can have implications in any dependent entities, a cyclical dependency can create a situation where any change to the entity can require the entity to change itself, creating a feedback loop.

The visualisations in Spoiklin Soice are based on Spoiklin diagrams, in which nodes represent functions, classes or entire packages, and the edges represent dependencies. The diagram is hierarchical, with most edges going from top to bottom, thus avoiding the need for the visual clutter of arrows on directed edges. Edges that go upwards are distinguished from normal edges by being drawn as curved lines.

Spoiklin Soice is primarily a tool for analysing and guiding program architecture as a visualisation tool, but it can also provide some useful metrics about the dependency graph, as seen in Figure 5.1. The visualisation can be used to view the program architecture at different levels, and find classes and functions that are too closely coupled—and should be decoupled. Figure 5.2 shows how Spoiklin Soice visualises the same Java web application

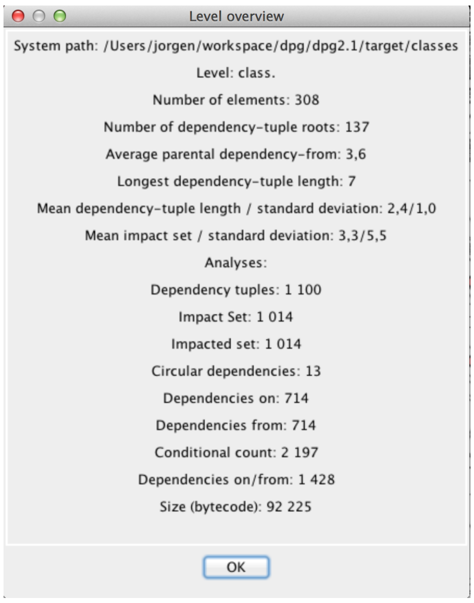

```
System path: /Users/jorgen/workspace/dpg/dpg2.1/target/classes
Level: class.
Number of elements: 308
Number of dependency-tuple roots: 137
Average parental dependency-from: 3,6
Longest dependency-tuple length: 7
Mean dependency-tuple length / standard deviation: 2,4/1,0
Mean impact set / standard deviation: 3,3/5,5
Analyses:
Dependency tuples: 1 100
Impact Set: 1 014
Impacted set: 1 014
Circular dependencies: 13
Dependencies on: 714
Dependencies from: 714
Conditional count: 2 197
Dependencies on/from: 1 428
Size (bytecode): 92 225
```

Figure 5.1: *Example Spoiklin Soice [95] analysis output. The data is from running Spoiklin Soice on a large Java-based web application, scoped at the class level.*

as in Figure 5.1 at the class level. Spoiklin Soice can only analyse Java applications.

5.1.2 Visual Studio 2012 Code Map

The most recent version[1] of Microsoft's flagship integrated development environment (IDE), Visual Studio 2012 (VS2012), has features for mapping dependencies in the code of a solution/project called Code Map [96]. Code Map shows interactions between modules, classes, and functions, and also displays some metrics in the graph (number of edges, circular edges and more). See Figure 5.3 for an example.

The feature set is very similar to Spoiklin Soice, but Code Map can only analyse systems written in Microsoft's languages (C#, VB.NET, etc.), as it is tightly coupled to the IDE, and also requires you to buy the most expensive license of the IDE, the Ultimate edition—which as of this writing costs 121 750 kr [97].

Code Map can also visualise relationships beyond the call graphs and dependency graphs that Spoiklin Soice makes, allowing you to answer questions such as "which methods interact with this data field?" As with Spoiklin Soice, Code Map is primarily intended as an architectural guide, but also—

[1]As of this writing, Visual Studio 2013 is still only a Release Candidate.

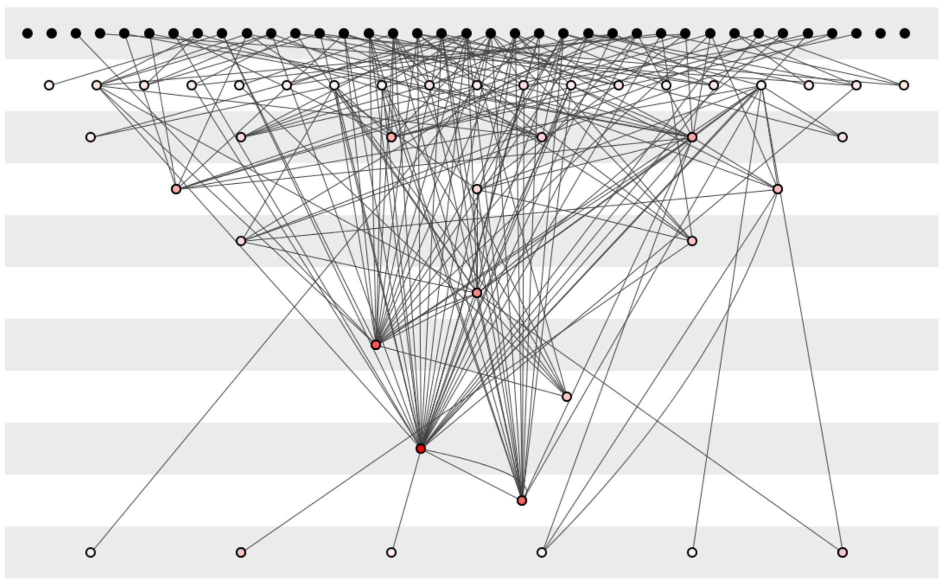


Figure 5.2: *Screenshot of Spoiklin Soice [95] showing the package-level structure of a Java web application.*

helped by it's tight integration with the rest of the IDE—to assist in debugging and in understanding unfamiliar code bases [98]. In contrast to Spoiklin Soice, it ignores some of the detailed analysis in favour of user friendliness.

5.1.3 `depend.py`

These are my own homemade tools, written during the preliminary thesis work. I originally made a model in NetLogo [99] (see Figure 5.4 for a screenshot), and then made a more sophisticated model called `depend.py` in Python—while ensuring the same results. The python model is an order of magnitude faster and is slightly more precise, but lacks some features, such as being able to dynamically change parameters when the model is running.

The improved version, `depend.py`, uses the NetworkX graph library [100], is written in Python, and the source code can be found in Appendix A. The application simulates downtime events in a user-defined dependency graph and attempts to calculate the downtime of each node. The structure of the graph and reliability (average uptime) of each node is specified in a JSON-formatted file. If no information about historical reliability exists, we can assume the SLA numbers to be a lower bound.

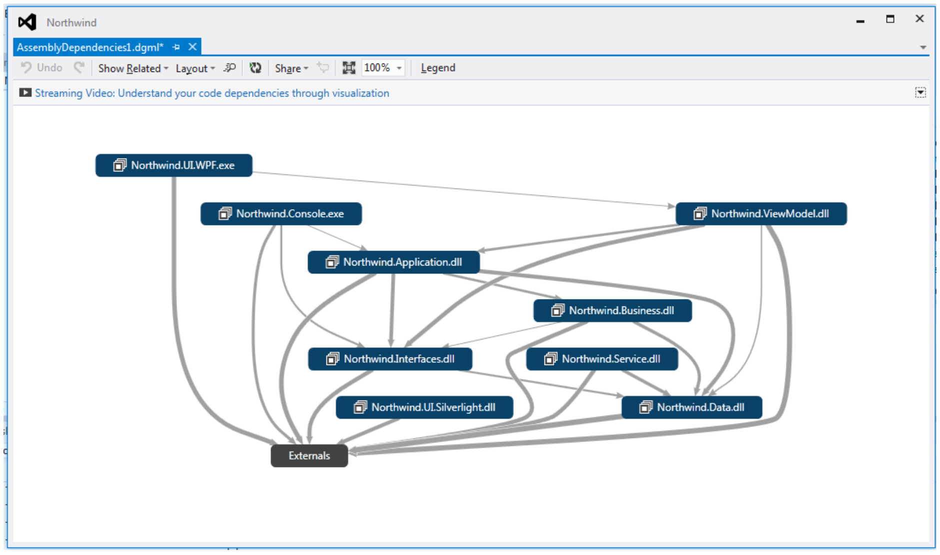


Figure 5.3: *Screenshot of Visual Studio 2012's Code Map feature, showing the relationships between the different assemblies in a project.*

From these inputs, the model can show that for typical infrastructure dependency graphs, the root node (the application itself) will have a much lower reliability than intuitively thought, since the downtime gets aggregated. As we saw earlier in Section 3.4, a node with 30 dependencies, each with 99.99% uptime, will still only be able to get $0.9999^{30} \approx 99.7\%$. A test run of `depend.py` with an imagined cloud-based infrastructure shows the same phenomenon. See Listing 5.1 for the exact result dump, and Figure 5.5 for a screenshot of `depend.py` showing the associated graph.

Note the abysmal reliability of the root node (`App`). Even though the individual availability is defined as *minimum* 99.3%, the root node has a real availability of only 95.8%, which is over 14 days of downtime in a year, or 1.3 days downtime in a month.

The NetLogo version has the capability of tagging an edge in the graph as *redundant*, meaning that a failure in one node will not propagate along this edge. This allows us to simulate what happens if this dependency is insulated and controlled. It also shows the fact that isolating a single edge is not necessarily enough to isolate failures originating in a specific node, as there may be alternate paths to this node. We will look closer into the problems—and potential solutions to—indirect dependencies in Section 6.2.3. The source code of `depend.py` can be found in Listing A.2 in the Appendix.

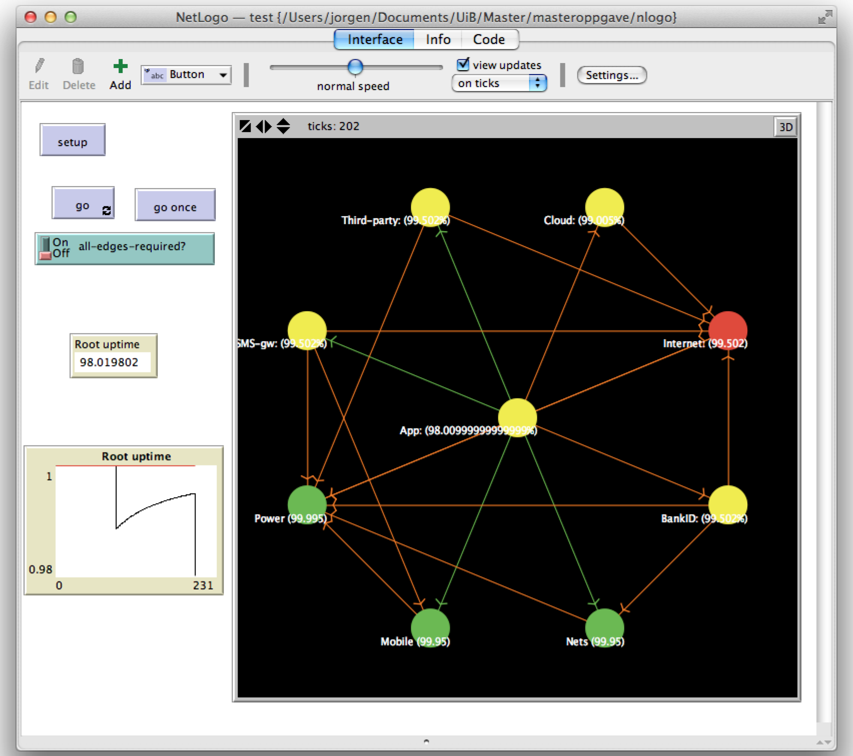


Figure 5.4: *Screenshot of the NetLogo-based dependency modelling application, showing an outage incident (red node) that propagates to outages in dependent nodes (yellow nodes). Note that the plot of root node uptime has a sawtooth-like appearance, with a slow, gradual rebound after each outage.*

5.2 Dynamic Presentation Generator

Dynamic Presentation Generator (DPG) is a content management system for the “Java in distant learning” (JAFU[2]) e-learning system. The system was initially written in 2004 by Yngve Espelid as part of his master’s thesis [101], and in the years since, large parts of the system have been written and re-written by students at the University of Bergen as part of their master’s work. The system is written using Java Enterprise Edition and the Spring web application framework, and Maven is used as package manager and to handle the build process.

[2]Norwegian: “Java i Fjernundervisningen”

```
Ran 900000 iterations, summary:
(0, {'name': u'App', 'sla': 100, 'uptime': 95.758})
(1, {'name': u'Cloud', 'sla': 99.95, 'uptime': 99.618})
(2, {'name': u'Internet', 'sla': 99.95, 'uptime': 99.795})
(3, {'name': u'BankID',  'sla': 99.3, 'uptime': 96.438})
(4, {'name': u'Nets', 'sla': 99.8, 'uptime': 99.225})
(5, {'name': u'Mobile', 'sla': 99.95, 'uptime': 99.813})
(6, {'name': u'Power', 'sla': 99.995, 'uptime': 99.984})
(7, {'name': u'SMS-gateway', 'sla': 99.95, 'uptime': 99.436})
(8, {'name': u'Third-party', 'sla': 99.95, 'uptime': 99.626})
```

Listing 5.1: *Output of a test run of* `depend.py`

The following analysis delve into the dependencies underpinning the system, and will also look at how the dependencies have been managed in the project, and investigate what attacks are possible due to the dependencies of the system. The version tested was DPG version 2.1.

Dependencies

In this project, the Maven build file specifies all dependencies, which is beneficial, as there is only one place to look up what dependencies are used. However, even if a dependency is included in the build, it doesn't necessarily mean that it is in use in the code. As the dependencies are injected at runtime through dependency injection, simply removing a dependency from the build will only result in an error when the removed code is actually invoked. This makes it hard to accurately test whether a dependency is used or not.

There is also something to be said about dependency creep. Dependency creep occurs when the ease of adding new dependencies for every (small) problem that needs to be solved makes the number of dependencies—and the system complexity—steadily increase for the duration of the development. Lack of solid project management can lead to developers adding unsafe dependencies, or multiple dependencies doing the same job.

The Maven build definition file for DPG lists 65 dependencies, and when transitive dependencies were resolved, the build ended up with a total of 138 dependencies. Some of the dependencies included are duplicates, or at least provide some duplicate functionality, such as jQuery—versions 1.2.3, 1.3.2 and 1.4.4 are all used in different parts of the system. In addition to the

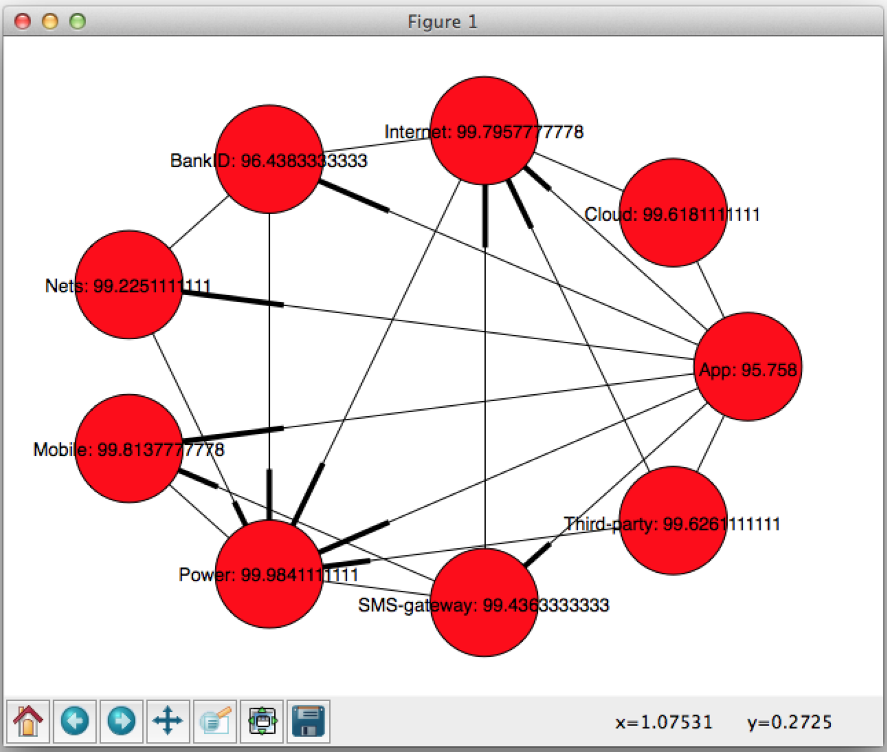


Figure 5.5: *Screenshot of* `depend.py` *in action.*

vulnerabilities present in such old versions, debugging is a lot harder when you don't know which version of a dependency is in use in a particular part of the application, or if a dependency is in use at all.

The DPG version tested is from Autumn 2012, so the dependencies are expected to be somewhat outdated compared to the current cutting edge, but many dependencies are grossly outdated. The most recent version of jQuery in the project dates back to 2010, as do the versions of jUnit and Hibernate, and the version of `hsqldb` database engine date all the way back to 2008. It is clear that updating and keeping track of dependencies has not been a priority in this project.

The sheer number of dependencies is also too high, causing the number of SLOC in the dependencies to outweigh the number of SLOC in the "actual" system—the DPG code itself. To find the sizes, gross estimates of the number of SLOC in the individual declared dependencies were obtained from

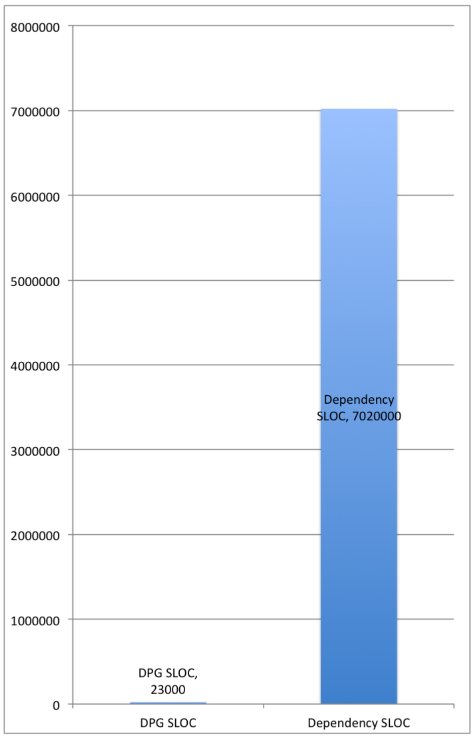


Figure 5.6: *An illustration of the relationship between DPG SLOC and dependency SLOC. Note the thin blue line representing the total SLOC in DPG.*

Ohloh [102], a web site dedicated to tracking open source projects and publishing associated metrics, including SLOC over time. The number of SLOC in the DPG code itself (excluding any dependencies)—as measured by the tool `sloccount` [103]—is about 23 000. A gross estimate of the number of SLOC in the dependencies is *at least* 7 020 000 SLOC. There is vastly more code in the dependencies than in the actual system, causing us to conclude that DPG only contains 0.32% DPG! See Figure 5.6.

As we discussed in Section 2.5, the expected number of vulnerabilities in a system is a linear expression of the total SLOC. Assuming conservative estimates of 5 defects per kSLOC ($D_D = 0.005$), and a vulnerability density (V_D) to defect density-ratio of 1%, we find the vulnerability density to be

$$V_D = 0.01 \times D_D = 0.00005$$

The expected number of vulnerabilities in DPG is thus

$$|V| = V_D \times 7043000 \approx 352.2$$

If DPG was only made up of the 23 000 SLOC in the core, the expected number of vulnerabilities would only be

$$|V| = V_D \times 23000 \approx 1.2$$

Using conservative estimates based on empirical data (see Section 2.5 for references), we arrived at an expected 352 vulnerabilities in the entire system, where about 351 of them are expected to exist in the external libraries. Cutting down on the number and size of external dependencies should help bring the number of expected dependencies down to a safer level.

Cross-build Injection

All Maven repositories listed in the build file are accessed over unencrypted HTTP (instead of encrypted HTTPS), and a Cross-build Injection (XBI) attack against the build process is definitely possible. A developer working over an insecure WiFi network (or a spoofed WiFi network) is vulnerable to a MITM attack that can easily compromise the code downloaded by Maven on build time, especially since the high number of dependencies (138) that must be downloaded allows for a large window of opportunity.

It should be noted that the server hosting the DPG internal (but externally accessible) Maven repo runs a version of Ubuntu that no longer receives security updates, and thus runs an Apache version that has known vulnerabilities. The investigation of DPG strongly indicates a potential for improved security.

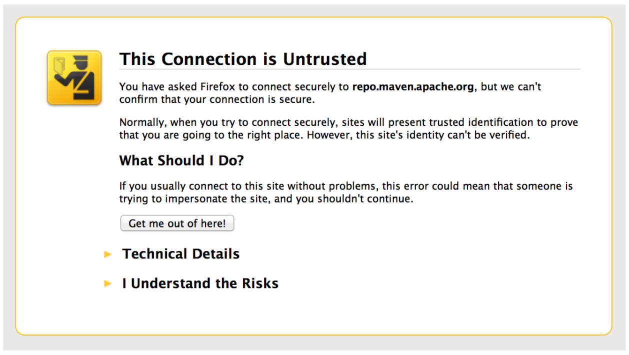
5.3 Build systems and Cross-build Injection

This section contains a short survey of the security capabilities in the most popular modern library package management systems. It is determined what countermeasures are in place to defend against XBI attacks (see Section 2.7 for a description of XBI). The package management systems surveyed are:

- Maven (Java)
- NuGet (.NET)
- `npm` (JavaScript)
- Ruby Gems (Ruby)

	Secure transport (TLS/SSL)	Checksums	Package signing
RubyGems	No	No	Optional
Maven	Optional, default off	Optional	Optional
`npm`	Yes, but not for installers	No	No
NuGet	Yes	Yes, SHA512	No

Table 5.1: *Summary of security capabilities in four of the most popular library package managers. Some have plugins to expand the security capabilities.*



This Connection is Untrusted

You have asked Firefox to connect securely to **repo.maven.apache.org**, but we can't confirm that your connection is secure.

Normally, when you try to connect securely, sites will present trusted identification to prove that you are going to the right place. However, this site's identity can't be verified.

What Should I Do?

If you usually connect to this site without problems, this error could mean that someone is trying to impersonate the site, and you shouldn't continue.

Figure 5.7: *Certificate warning when trying to access `repo.maven.apache.org` using https. All examples in the documentation also use unencrypted http.*

A summary of the security capabilities of the different systems can be seen in Table 5.1.

While some of the systems support certificates or checksums to secure the packages, both `npm` and Ruby Gems do not use these mechanisms by default. Checksumming alone is not sufficient to prevent XBI if there is no secure transport, as the checksum could also be tampered with. NuGet uses both secure transport and checksumming, successfully protecting against XBI based on a MITM attack. However, it does not protect against a compromised host, and the process of uploading packages does not include any vetting or security analysis.

`npm` uses TLS/SSL by default,[3] but older versions don't validate the certificates, allowing an attacker to perform a MITM attack with self-signed certificates. In tests, `npm` version 1.2.18 (the most recent at the time) even accepted a self-signed certificate with a mismatched hostname. RubyGems defaults to plain unencrypted HTTP. Maven supports HTTPS, but it is only

[3]Packages containing installers that download more components—like the package `phantomjs`—make their own choices about transport security, and may decide not to use TLS/SSL.

available for paying Sonatype customers, and thus is not supported on Maven Central (see Figure 5.7).

Both Maven and RubyGems support signed packages, but Maven requires you to implement your own validation, and RubyGems only has rudimentary support with no proper supporting infrastructure, so almost no packages use the functionality [104].

None of the major package management systems have a satisfying degree of security "out of the box". Maven can be configured for secure operation, but requires an expensive license and a lot of manual work to implement checks to validate certificates and checksums. NuGet has the best grip on security "out of the box", but still lacks package signing.

The largest missing piece, however, is that none of the package managers have methods and routines for validating packages before they are published, and for stopping impersonators from uploading malicious versions of known libraries. None of them have any way to notify users of vulnerabilities and insecure versions, and none of them have mechanisms in place to allow the users to notify other users of unsafe or impersonating packages.

5.4 `npm` and dependency explosion

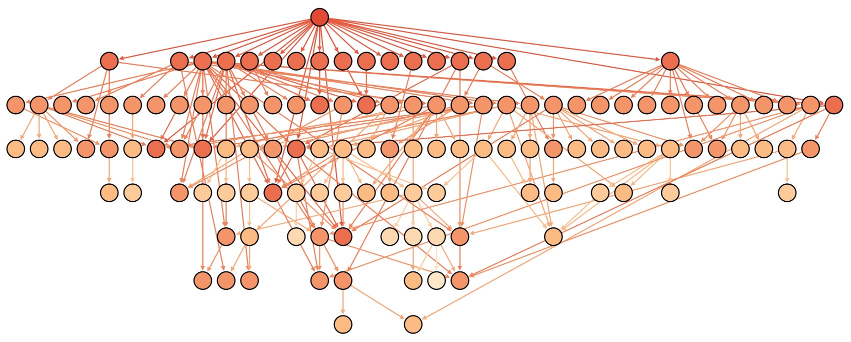


Figure 5.8: *Dependency graph of the* `npm` *package* `yeoman` *(top row, centre). A total of 129 packages, with a radius of 6. The degree of the* `yeoman` *node itself is only 25. Not included here are potential multiple versions of the same package.*

In the previous section, we saw how `npm` [105] could be vulnerable to XBI, and we have previously claimed that the number of dependencies in a typical software project grows larger when using a package manager, because adding new dependencies get easier with a package manager. Here, we will investigate the structure of dependency relations in the `npm` package repositories, and look at the inter-package dependency graphs.

The datastore behind the official `npm` repository is an open, publicly readable CouchDB instance,[4] which makes it easy to retrieve data for analysis. The repository also keeps good track of the dependencies between packages, along with the set of packages depending on a specific package.

Package dependency graphs

To see the effects of encouraging the use of multiple dependencies, we can take a look at the `npm` package `yeoman`, which is a package for project workflow management [108]. See Figure 5.8 for the full package tree that is downloaded when installing `yeoman`. Note that this actually is not a tree, but a Directed Acyclic Graph (DAG) that contains the minimal spanning tree that `npm` uses to avoid downloading packages more than once. The extraneous edges in the DAG makes—as we have previously discussed—removal of unwanted dependencies harder.

The dependency graph of `yeoman` is a good example of indirect dependencies leading to a dependency “explosion”, where the number of packages increases beyond expectation due to the contributions of transitive dependencies. `yeoman` has listed only 25 dependencies in its package definition, but due to indirect dependencies, the total number of packages in the graph increases to 129. With `yeoman` in the centre of the graph, the radius is 6, so there are 6 levels of dependency relations separating `yeoman` and the furthest layer of libraries. It should be noted that this is not a construed case of a useless meta-package, but an actual useful and popular package that like many others has a very large dependency graph.

The degree distribution

`npm` does not seem to discourage adding dependencies to a project, on the `npm` home page, they even have a listing of “Most Depended-upon Packages” [109] as a measure of popularity! As of July 10, 2013, the most depended-upon package is `underscore`, with 3274 dependents, or about 9.5% of all packages in the `npm` registry.

Figure 5.9 is a plot of the in-degree distribution of packages in the `npm` dependency graph, sampled on July 10, 2013. The in-degree of a package is the number of dependents it has. The cut-off for the plot is at 10 dependents, and there are 715 packages with 10 dependents or more. The far-left package is the package `underscore`, with 3274 dependents. The total num-

[4]This “feature” actually caused a serious incident in 2012, when it was discovered that the user information—including password hashes—in the database also was publicly readable, and had been for some time [106, 107].

ber of packages in the npm registry at the time of drawing this graph was about 34 500.

From this graph, and its power-law distribution with a long tail, we can infer that the package dependency graph in npm probably is a small-world graph, with a small diameter and a large number of hubs.

The depended-upon graph

Another way to study the dependency graph is to build the graph of the dependents of a specific package. That is, the subgraph of packages that has a path to a specific package in the dependency graph. In Figure 5.10, I have mapped the reverse dependency graph of the package async as it were on July 17th 2013. The data was collected directly from the npm datastore.

The network was built using a small Python script that ran a depth first search (DFS) from node to node, using the dependedUpon relation defined in the database. The search was started on the package async. Due to computing constraints, the search was stopped at a max depth of 5, so this is just a subgraph of the full graph. The source code of the Python script used to build the graph is shown in full in Listing A.1 in the Appendix.

Like Figure 5.8, this is not a dependency tree, but a DAG that contains a minimal spanning dependency tree that npm constructs to avoid dependencies being loaded multiple times.

The package async itself is depended on by 2437 other packages (about 7% of all packages in the npm registry), but the full reverse dependency graph contains a total of 8797 nodes (about 25%). Over a fourth of all npm packages depend directly or indirectly on the async package, and over 72.3% of the packages that depend on async do so indirectly.

This means that a flaw in the async package will affect over 25% of all packages, and over 72% of the affected packages cannot remedy the situation by removing this dependency themselves, but instead have to rely on others to remove this dependency. Given that there are multiple paths between each set of two nodes—there are 13725 edges and a total of 39296 unique shortest paths between any two nodes—more than one edge must typically be removed to delete a node from the graph.

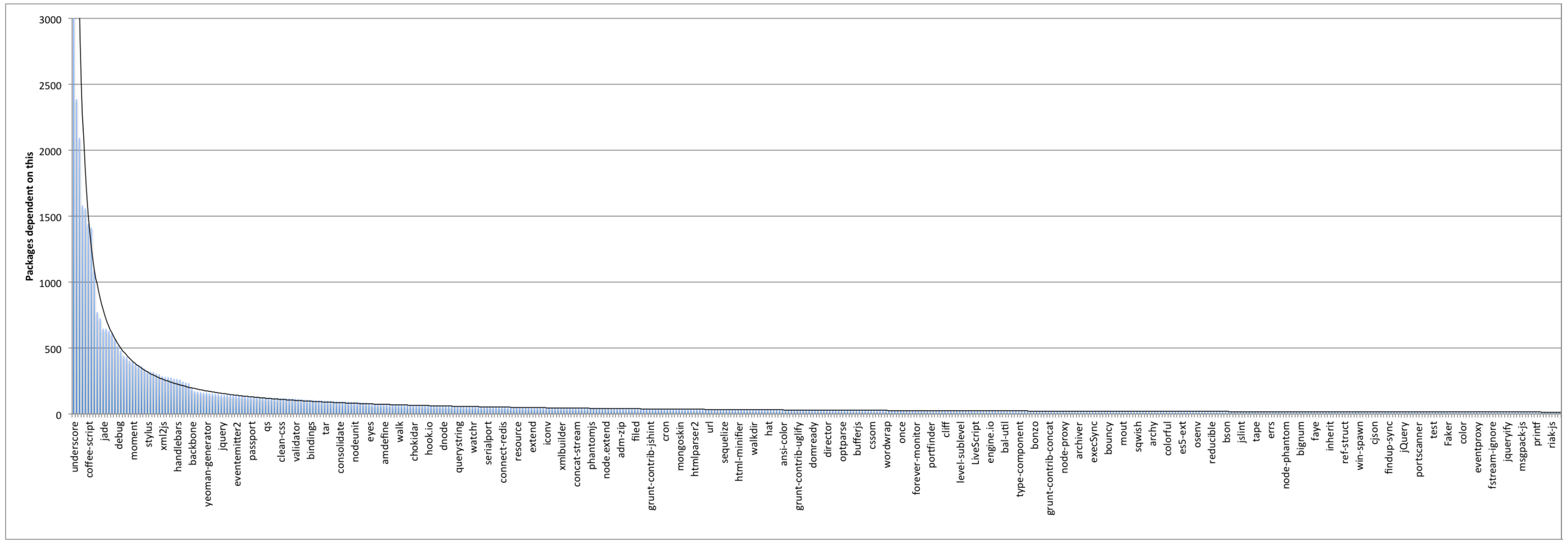


Figure 5.9: *In-degree distribution of* ***npm****'s most-depended-upon packages [109] as of July 10 2013. It is a power-law distribution, with a long tail. The black line is a fitted power-law curve.*

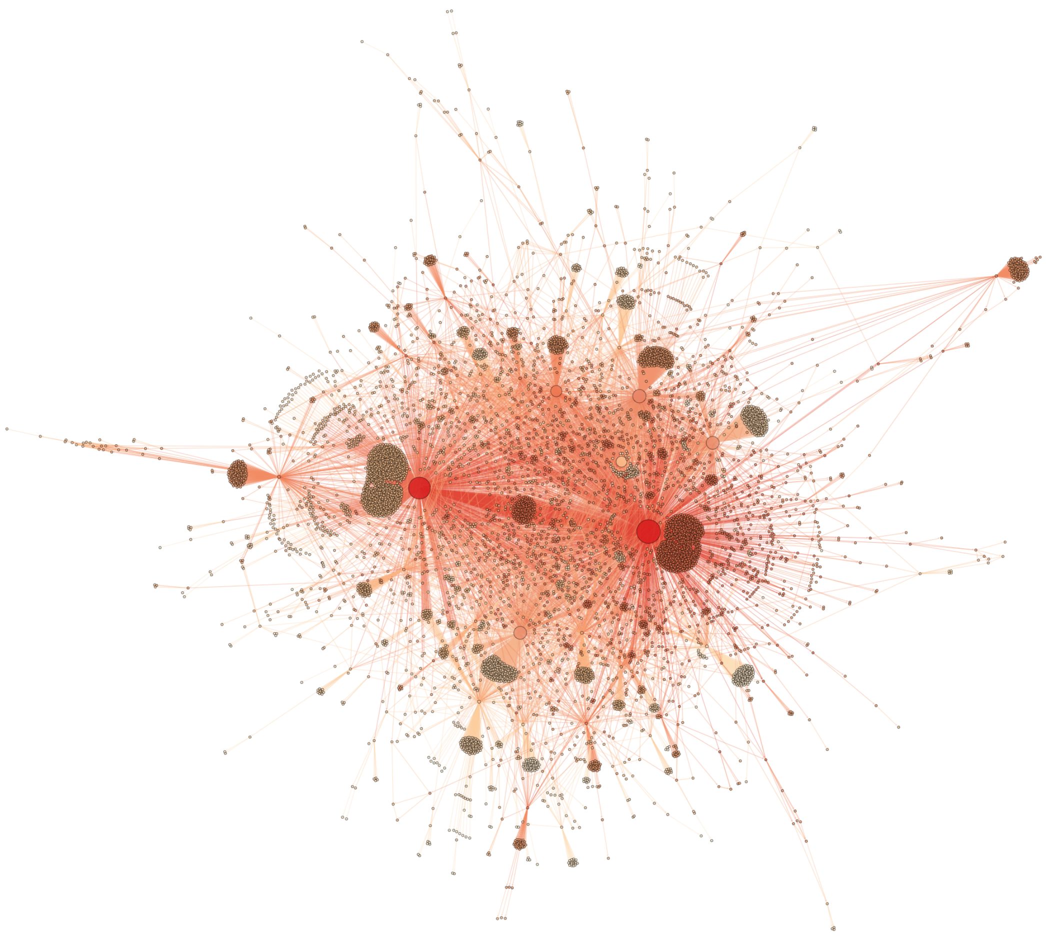


Figure 5.10: *The dependency graph of the* `npm` *package* `async` *(red, middle right). The graph has a total of 8 797 nodes and 13 725 edges, the* `async` *package itself only has a degree of 2 437. The figure is rendered using Gephi [110].*

Chapter 6

Solutions

A lot of banks don't encrypt, a lot of those agencies that you think might encrypt Social Security Numbers actually don't, because it is very complicated. ***It is cumbersome and there's a lot of numbers involved with it.***
– South Carolina Governor Nikki Haley, October 2012

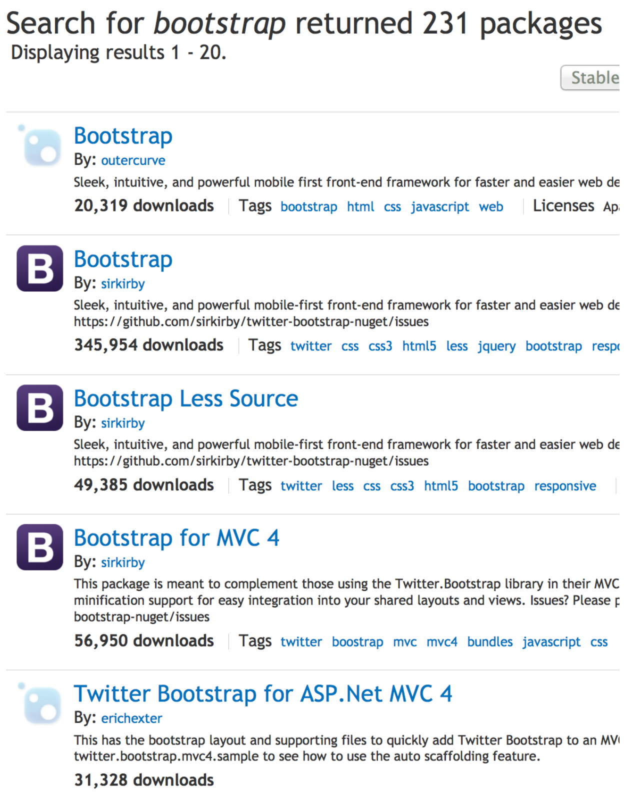
The previous chapters focused on problems, issues and vulnerabilities. We will attempt to turn the negative message around by showing some of the solutions in use today, as well as describing potential solutions that could be put in place or at least be researched further. Both technical and non-technical (process- and management-oriented) solutions are described, including solutions to facilitate proper handling of infrastructural dependencies and to mitigate issues caused by library dependencies.

6.1 Libraries and package managers

There are two technical solutions to problems with library dependencies: extending and improving package managers to limit the number and seriousness of dependencies, and implementing security wrappers to insulate against failures and vulnerabilities in software libraries.

6.1.1 Extensions to package managers

As we have seen in the previous chapters, current package managers leave much to be desired with regards to authenticity, integrity and reputation management.


Search for *bootstrap* returned 231 packages
Displaying results 1 - 20.
Stable

Bootstrap
By: outercurve
Sleek, intuitive, and powerful mobile first front-end framework for faster and easier web de
20,319 downloads | Tags bootstrap html css javascript web | Licenses Ap

Bootstrap
By: sirkirby
Sleek, intuitive, and powerful mobile-first front-end framework for faster and easier web de
https://github.com/sirkirby/twitter-bootstrap-nuget/issues
345,954 downloads | Tags twitter css css3 html5 less jquery bootstrap resp

Bootstrap Less Source
By: sirkirby
Sleek, intuitive, and powerful mobile-first front-end framework for faster and easier web de
https://github.com/sirkirby/twitter-bootstrap-nuget/issues
49,385 downloads | Tags twitter less css css3 html5 bootstrap responsive |

Bootstrap for MVC 4
By: sirkirby
This package is meant to complement those using the Twitter.Bootstrap library in their MVC
minification support for easy integration into your shared layouts and views. Issues? Please p
bootstrap-nuget/issues
56,950 downloads | Tags twitter boostrap mvc mvc4 bundles javascript css

Twitter Bootstrap for ASP.Net MVC 4
By: erichexter
This has the bootstrap layout and supporting files to quickly add Twitter Bootstrap to an MV
twitter.bootstrap.mvc4.sample to see how to use the auto scaffolding feature.
31,328 downloads

Figure 6.1: *Screenshot of a search in NuGet returning multiple different packages with different authors, all claiming to provide the same library. None of the authors identify themselves in NuGet as the official author. Only after some research outside NuGet can you find that* ***sirkirby*** *is the official package maintainer.*

As an example of the problems that stem from lack of author authentication, see the screenshot in Figure 6.1. Most package managers (the example is from NuGet) have multiple packages with the same names and different—sometimes unknown—authors. It is often hard to find which version is the correct edition from the correct author. It is interesting to note that over 50 000 have downloaded the various Bootstrap packages from the wrong author [111]. Some of the packages that claim to provide Bootstrap don't even provide the most recent version.

But even with their inherent problems, package managers are one of the best places to attempt to solve the problems faced when managing dependencies, as the package managers are closer to the problem domain and already have much of the necessary infrastructure in place.

In this section, we will review existing extensions to package managers, and briefly describe the ideal secure package manager.

The Node Security Project

The Node Security Project [112] is a community project organised by ^Lift Security. The project aims to perform audits of every package in `npm`, provide advisories and patches, and then provide a public API of audit results, possibly integrated into `npm`. Their plan is ambitious, and not much has come of it so far—only four published advisories from March 2013 to July 2013 [113]. It will be interesting to see if they can get the wide support they need in order to execute the plan. The project also intends to implement a distributed (crowd-sourced) audit mechanism, which has the potential to provide quicker audit feedback and improve the quality of the audits. There is great potential here if the project successfully gets off the ground.

Semantic versioning (SemVer)

Semantic Versioning (SemVer) [114, 115] is an attempt at applying a standardised set of semantics to the version numbers of a package or library. The intent is that the version number should accurately describe the importance and scope of the changes that this version represents.

The version numbers in SemVer are in the form $major.minor.patch - branch$. Changes to the *branch* section are for internal use, for example to identify a particular build. A change to the *patch* section denotes a bugfix or other change that don't change the packages' API. The *minor* section gets incremented when new—but backwards compatible—changes to the API is introduced. The *major* section is incremented when changes that are not backwards compatible are introduced.

If all packages conform to the SemVer specification, one can add rules to the build process to automatically use the latest available *patch* version, without worrying about any breaking changes. Keeping dependencies up to date will be easier, as you typically can update to the most recent *minor* version as well without much consideration.

Both NuGet and `npm` recommends SemVer [116], and `npm` requires that it be parseable as a SemVer version [117], but none of them actually enforce the semantics of it. In the end, the actual version number assigned to the released package is up to the individual developer.

SafeNuGet

SafeNuGet [118] is a NuGet package designed to check the versions of the packages in your project against known vulnerable versions, and give you a warning or halt the build if any of the packages are known to be unsafe. The

SafeNuGet project was initiated by Erlend Oftedal (of BEKK Consulting) in June 2013, and was quickly adopted as an official OWASP project [119].

To identify unsafe versions, SafeNuGet relies on an official blacklist being kept up-to-date. The registry is maintained by just one person, although anyone can suggest additions via email, pull requests or issue tickets on GitHub. As of October 2013, there are still only 10 packages on the official blacklist, and it was last updated in June 2013 [120].

Extending on the concept of a blacklist by implementing it as a service instead of just a flat file could be a nice move towards making the package really useful. Ideally, we would like to see the job of keeping track of vulnerable versions of packages be performed by an impartial third-party (or even crowd-sourced), and exposing this as a platform-agnostic API available to all package managers. This way, the feature can be a part of all package managers instead of only NuGet.

The ideal secure package manager

Here, I describe the most important features and requirements of an ideal secure package manager. The list is not complete, and should be considered as a wish-list to be implemented on top of the features that exist in major package managers today.

- Verify the authenticity of package authors, to avoid multiple versions of the same package from different authors, and to make package hijacking close to impossible.
- Warn users about known vulnerabilities in packages, or vulnerable versions of a package. Notify users about vulnerable versions, even when the system is already in production or no longer in active development. By default, it should not allow a user to install a package known to be vulnerable.
- Notify subscribers or users about licensing updates or changes to package authorship. It should block the installation if there are license conflicts.
- Have a system for managing author and package reputation, and clearly communicate the reputation of the packages and authors to the users.
- Use *and enforce* semantic versioning through the above mentioned package/author reputation mechanism. Updates that don't conform to SemVer (such as introducing breaking changes in a *minor*-level update) will get flagged, and a warning will be shown to the user.

- Use secure transport methods (HTTPS) and signed packages to counter XBI attacks.
- Have a mechanism to discourage duplicate dependencies in projects: *"Do you want to install this package? It may overlap with the package 'XXX' that you already have installed."*

6.1.2 Security wrappers

If it is not possible to avoid having a potentially unsafe dependency in systems, the dependency can be isolated using security wrappers. A security wrapper act as shield around the dependency, and—depending on the level of distrust in the dependency—validates all input and output parameters. See Figure 6.2 for a simple conceptual diagram. If a security wrapper attempts to totally isolate the dependency, even from the underlying filesystem and networking stack, it's called a sandbox. Security wrappers and sandboxes are often used in web browsers to insulate against security issues in plugins, and to stop malicious web content from affecting browsers [121].

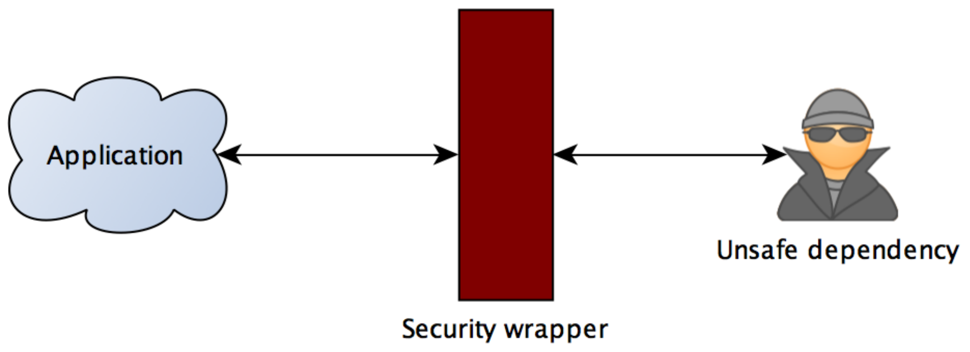


Figure 6.2: *Conceptual diagram of a security wrapper isolating the dependency from the rest of the system or application.*

There are two scenarios to consider, each with a different set of solutions and different levels of complexity:

1. Protecting against a dependency that can pose an active threat to your system, and may attempt to do bad stuff. This is a potentially malevolent dependency.
2. Protecting against a dependency that may contain vulnerabilities that threaten the security of your system. This is an unsafe dependency.

In the latter case, implementing a security wrapper that filters or validates input to the dependency is sufficient, as it can be assumed that the dependency doesn't do anything wrong on purpose. The filters will typically need to check for invalid input, such as XSS payloads or other malformed input. To apply such filters to dependencies that directly access networking or file system resources, modern programming languages support modifying the code through reflection to add the necessary filter code.

In the case of the potentially malevolent dependency, it must be totally isolated from the system with a sandbox, but even then it is conceivable that the dependency can do damage—like miscalculate the amount of money in a bank account if the account is owned by the attacker. To implement checks that uncover any malevolence, one would have to reimplement the dependency oneself, which is not feasible. A potentially malevolent dependency should never ever be allowed in a project.

Modern package managers allow potentially malevolent dependencies into systems by not verifying the authenticity of the claimed package authors. An attacker could potentially publish a package to NuGet that claims to be jQuery, but contains a slightly modified, malicious version. Security wrappers can stop most obvious attempts, but a sufficiently sophisticated attack can be impossible to stop by introducing security wrappers or even sandboxes.

6.2 Infrastructures

Dependencies in an infrastructure require its own set of solutions. The classical solutions typically involve removing the dependency or insulating the system from the dependency. More recent techniques involve accepting the existence of faults and errors, and instead focus on improving the way a system handles failures. As we will see at the end of this section, the case of Netflix illustrates how failures, latency, noise, and errors can be embraced as stabilising agents.

6.2.1 Insulating against dependency failures

A system should be designed to insulate against faults and failures in its dependencies, and to provide graceful degradation—if possible—in the case of a failed dependency. If the dependency provides a non-essential service, it may be acceptable to give the user a degraded level of service. When Netflix's recommendation service suffers an outage, Netflix simply gives you less accurate recommendations instead of taking down the entire system.

However, dependencies can provide essential services—such as payment, billing or authentication—that have to be in place for any useful interaction with the system. While graceful degradation may not make much sense in the case of an outage in these kinds of services, it can be handled by introducing redundancy, by having multiple dependencies in place to handle the tasks assigned to the service.

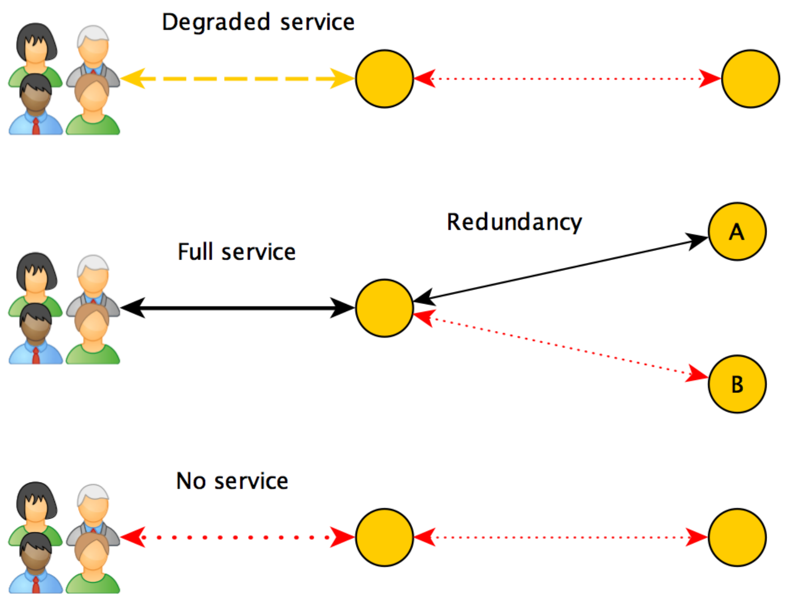


Figure 6.3: *Conceptual diagram of the different dependency failure insulation patterns.*

Some dependencies are either too difficult to isolate, or cannot be made redundant without unacceptable consequences, like high costs. Contingency plans and incident response plans should be in place to minimise the impact of unwanted incidents.

See Figure 6.3 for an example of the different ways of insulating against dependency failures.

6.2.2 Metaclouds and insulating against cloud platform failure

Cloud platforms like Amazon EC2, Microsoft Azure and Google App Engine (GAE) sometimes fail [122], and isolating a system from faults in this kind of dependency is hard, as a cloud platform provides the foundation for the system. For this reason, cloud platforms are typically designed to be very fault resistant, but some cloud platforms still have more outages than what is acceptable for most use cases. Heroku is an example of a cloud provider with poor uptime, with uptime as low as 99.2% for December 2012 [123].

A possible solution to this problem is to design a system that is capable of running on multiple clouds. Solutions such as AppScale [124] exists that aim to be a "Google App Engine Compatible" platform on top of any IaaS cloud, meaning that a GAE application can be deployed to any other cloud provider, avoiding vendor lock-in by Google. However, the complexity and cost of performing a successful migration in the event of an outage is prohibitively high.

Migration of systems hosted on IaaS-based public clouds are simpler, as one would only need to migrate the individual virtual machines, although the necessary control systems and provisioning tools would also need a costly and complex migration, ruling out any chance of using this method to avoid an outage. It would have to be carefully planned and executed—or else the migration itself could cause a system outage.

Moving an entire system is a huge undertaking. It requires large and invasive changes to tools, routines, and the surrounding systems. However, as we will see later, if the system is kept in a state of partial outage, these surrounding systems, tools and routines will—out of necessity—have evolved to handle migrations. I therefore propose a meta-cloud, an abstraction on top of several cloud platforms, in which the system running on the meta-cloud is unaware of which cloud platform the individual server instances are executing on [125].

One can imagine a simple implementation that simply chooses a cloud platform randomly from a set of acceptable cloud platforms when asked to instantiate a new virtual sever. By necessity, the system would be forced to tolerate this kind of infrastructure. This is definitely possible, but probably would run into difficulties with latency or cost.

6.2.3 Indirect dependencies

When attempting to lower exposure to risk by removing dependencies, indirect dependencies can be a serious hindrance. When bringing in a new dependency to replace an unsafe or untrusted dependency, the same indirect dependencies may still be underlying the new dependency, or it may be depended on by another existing dependency in the system. As an example, we can imagine two competing payment service providers both hosting their infrastructure on the same public cloud or in the same datacenter, or two competing website monitoring services using the same SMS provider for notifications. See Figure 6.4.

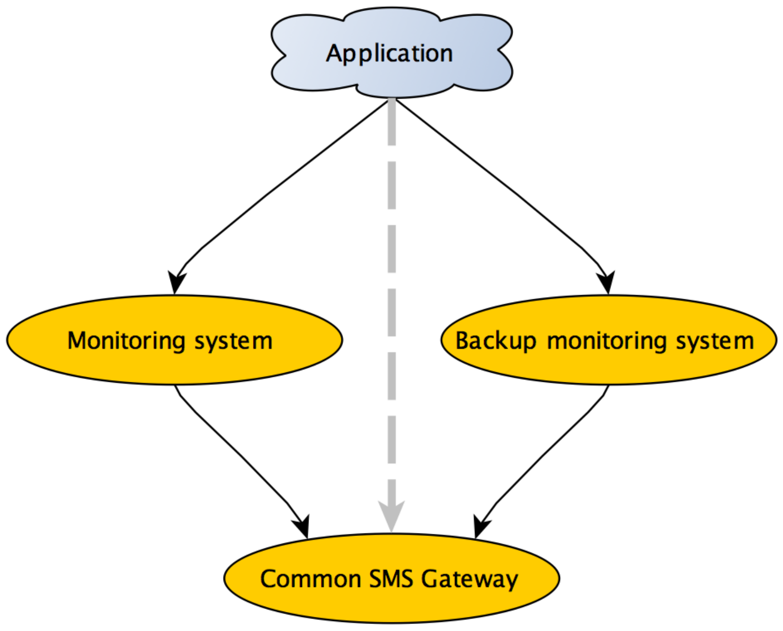


Figure 6.4: *A simple diagram illustrating the problem of indirect dependencies. While there are two redundant monitoring systems in place, they both have the same underlying SMS Gateway. The application thus has an indirect dependency (dotted grey line) to the SMS Gateway, and loses the intended redundancy.*

A real-world example of indirect dependencies causing loss of intended redundancy is PagerDuty [126], which is a third-party service for paging operations personnel about downtime or other issues that require immediate attention. When the system breaks down at 4AM on a weekday night, PagerDuty is responsible for paging or calling the person who has to get up and fix the problem.

The PagerDuty service is hosted on Amazon EC2, but most of their customers are hosted there as well [127]. This means that a downtime event in the common underlying dependency (EC2) takes down both the customers system *and* PagerDuty, leaving the customer with no way to get warnings about the system outage! An event causing reduced performance (but no total outage) in the underlying common dependency will be exacerbated by the fact that PagerDuty will require more resources in order to report this event to their customers.

To avoid indirectly placing all the eggs in one basket, the dependencies of a service must be analysed and considered before choosing to depend on the service. Having two redundant backup providers is of little use in a crisis if both are located in the same burning building.

6.2.4 Designing for failure

No matter how hard you try, no matter how much effort is put into making the system fail-safe, components will always break, hard drives will fail, and servers will go down. The focus has traditionally been on designing the system with fail-safes so it never goes down, or specifying hardware solutions to avoid having hardware that goes down, like RAID and redundant power supplies. This obviously doesn't scale well. In a large, "Google-sized" datacenter, a hard drive failure is expected to happen almost every minute [128].

Instead of attempting to avoid any and all failures, the most important thing is to be able to get the system quickly back to a normal running state. Instead of focusing on Mean Time Between Failure (MTBF), which can be vulnerable to single-component failures, the system should be optimised to lower the Mean Time To Recovery (MTTR) [129]. In other words: recovering quickly from failures is more important than having fewer failures. By "embracing" failure and accepting it as inevitable, the system can be designed to function as well as possible despite failures—the system is "designed for failure".

The multi-discipline area of **resilience engineering** [130] deals with some of these topics. Resilience engineering attempts to create resilient systems, systems that are robust under changing conditions, and have the ability to recover from catastrophic failure. Resilience engineering has traditionally been a focus in engineering disciplines such as airplane design, power plant design and other life-and-death engineering disciplines. In recent years, resilience engineering has seen a renaissance with the advent of cloud computing and "web-scale" applications, which sets harder demands for system resilience.

Netflix, Hystrix and Chaos Monkey

Netflix is an example of an organisation that has adopted the paradigm of designing for failure, and Netflix has developed several new technologies to incorporate this line of thinking into their systems.

Even though Netflix's architecture was designed for loose coupling and graceful degradation, it became apparent that they still needed to implement even looser coupling. Simple timeouts were not sufficient to avoid resource contention due to full buffers and queues, and failures would spread from service to service. Netflix has developed several novel solutions to the problems inherent in handling dependency graphs in SOA architectures. Hystrix is one of the most well-known solutions [131].

Insulation of failures and providing graceful degradation is harder than many think. Tools like Hystrix demonstrate this by adding a complicated layer of logic between the different services to absorb problems and issues instead of propagating them through the system [132]. Hystrix does this by employing a complex arrangement of thread pools, connection pools and timeouts at different levels, in order to allow a certain level of temporary failures before failing entirely.

Hystrix also facilitates service isolation by employing a “circuit-breaker” technique that shuts down requests to service nodes if the latency or number of failures exceed a certain threshold [131, 133]. This is done in order to avoid cascading failures and to isolate failures to only the services that actually fail, which allows for graceful degradation. See Figure 6.5 for an overview of the flow of a request through a Hystrix “circuit”.

Another solution developed at Netflix to improve the system’s resilience is Chaos Monkey (and Chaos Gorilla, its bigger brother) [134], inspired by a fuzz testing tool used in the development of the original Macintosh [135]. Chaos Monkey maintains a certain level of constant failure by randomly shutting down individual cloud server instances. This ensures the system is resilient against some classes of failures, and that any lack of resilience will cause it to fail early, forcing developers to implement the needed resilience.

“The best way to avoid failure is to fail constantly.” [136]

Chaos Monkey is part of Netflix’s “Simian Army”, which also includes Chaos Gorilla, and Latency Monkey. Chaos Gorilla shuts down entire Amazon availability zones in order to test the systems that automatically rebalance services to functioning availability zones. Latency Monkey induces random latencies in client-server communications and between services to simulate degradation and make sure applications are resilient to latency spikes and other networking issues.

By keeping the system in a constant state of partial failure, the system is forced to be resilient against failures. Others have adapted this approach, such as modern web browsers, which are undergoing continuous “fuzz testing” to tease out problems in the parsing code before releases [137]. This way of improving resiliency by actually testing it is inspired by the natural world: introducing noise, latency, failures and uncertainty in order to force systems to be resilient to failure scenarios are analogous to the evolutionary pressure we find in nature, where organisms are forced to adapt and improve in order to survive.

In my opinion, this way of thinking is immensely beneficial for large systems, and should be a best practice for constructing large systems that need to be resilient to errors, such as national infrastructures and too-big-to-fail systems. Systems designated as "too big to fail" should rather be recognised as *too big not to fail*, and should be designed to handle frequent small failures rather than waiting for a catastrophic failure before improving its robustness.

6.3 Non-technical solutions

Technical solutions aren't always sufficient. There are many examples of good technical solutions that aren't as effective as intended due to non-technical reasons. TLS is a good example. TLS uses mostly good, decent cryptography based on a mathematically and technically sound process[1], but in the end TLS easily falls victim to attacks directed at the non-technical part of the system: the user. TLS are not particularly useful when the users simply press "OK" on any popup that warns them of a certificate problem.

The same goes for secure development. Even if every developer has access to state-of-the-art static analysis tools to detect almost every vulnerability that exist in a program, they still have to be used properly to gain any benefit.

Secure development requires secure dependencies. How can one expect to develop a more secure system by implementing rigorous secure development processes if the libraries and frameworks used do not follow the same methodologies? When deciding what dependencies or libraries to include in the system, the architect or designer should include the security of the dependency as an important factor, along with its size and scope. The dependency should be as narrow in scope as possible given the problem it intends to solve. Multiple dependencies doing nearly the same thing should be avoided.

When deciding on a Secure Development Lifecycle (SDL), dependency management should be included in this lifecycle. In the following section, we will evaluate a few different secure development life-cycles, in order to determine which handles dependency management best. "Traditional" development life-cycles should be modified to include secure dependency management.

[1]Some technical attacks on TLS exist, most notably the CRIME, BEAST and Lucky Thirteen attacks. [138]

6.3.1 Choosing a Dependency Management Lifecycle

In order to select a development lifecycle that properly addresses the problems related to dependency management, we first need to outline what we want from the development lifecycle. The development lifecycle needs to include:

- Handling of updates to libraries and other dependencies for the entire life of the product.
- Vulnerability monitoring (looking for published or in-the-wild vulnerabilities) in libraries and other dependencies for the entire life of the product.
- A well-defined incident response to new vulnerabilities in 3rd party libraries and other dependencies
- Guidance on infrastructure dependencies, and mechanisms for keeping track of them and insulating against them if necessary.
- A step where library security is evaluated, either directly, through audits and code reviews, or indirectly from reputation and public vulnerability information before being included into the project.

Let us take a look at the most popular SDLs in use in the industry, and see how they stack up against our short set of requirements. The SDLs reviewed are Microsoft Security Development Lifecycle [139], Cisco Secure Development Lifecycle [140], and OpenSAMM [141]. We want to determine which is the best candidate to provide guidance and best practices for secure handling of dependencies in a development process.

Microsoft Security Development Lifecycle (SDL): SDL does not include any mention of unsafe libraries and secure handling of dependencies. It only recommends that *"Analyzing all project functions and APIs and banning those determined to be unsafe helps reduce potential security bugs with very little engineering cost."* This advice is aimed at banning developers in the organisation from using "unsafe" constructs, not banning unsafe dependencies.

As a whole, the Microsoft SDL doesn't sufficiently address the challenges that dependencies pose, and cannot be recommended to provide guidance to an organisation that attempts to perform secure dependency management.

Cisco Secure Development Lifecycle (CSDL): CSDL pays careful attention to library dependencies, which it terms “3rd Party Software”. It offers guidance for choosing secure versions of libraries, and registering library dependencies. After the application is launched, the Response stage of CSDL includes vulnerability monitoring of library dependencies, making sure that vulnerabilities in library dependencies are caught and the libraries updated. It also includes strategies for responding to vulnerabilities discovered after launch.

CSDL seems to be a good lifecycle to adhere to, in terms of dependency management.

Open Software Assurance Maturity Model (OpenSAMM): OpenSAMM is not a fully specified development lifecycle model, but a framework to build a security development model tailored for the individual organisations. It still provides guidance for each step in a products lifecycle, so we’ve included it here.

OpenSAMM includes objectives for defining and enforcing security requirements in third-party dependencies in the planning stage, and for preventing unexpected and possibly redundant dependencies by avoiding “one-off implementation choices” with well-maintained lists of recommended frameworks and libraries. In the after-launch stages of the lifecycle, OpenSAMM includes activities to monitor “high-risk dependencies” and keep dependencies updated through a consistent process.

OpenSAMM looks to be a guidance tool that can help to avoid a lot of the vulnerabilities that stem from poor handling of library dependencies, and would be my personal first choice. Although it, along with all the others, doesn’t mention infrastructural dependencies.

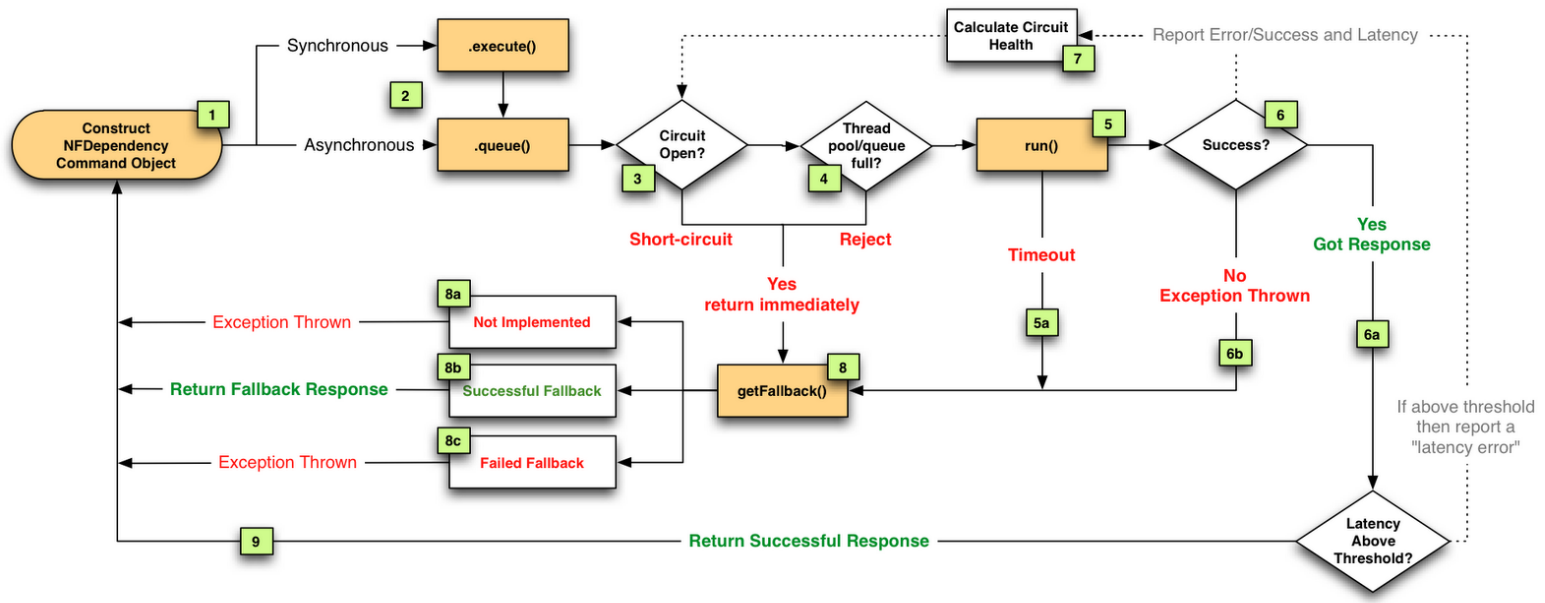


Figure 6.5: *Flowchart of a service implemented as a Hystrix circuit. Note how it is designed to short-circuit and return a fallback response if the circuit is closed due to a failure.*

Chapter 7

Conclusions and summary

Controlling complexity is the essence of computer programming.
– Brian Kernighan [142]

In this final chapter we summarise the thesis, chapter by chapter, and offer a conclusion, before looking at possible extensions of the thesis work as well as open research questions.

7.1 Summary

We have seen that dependencies play a larger role than most developers think. Chapter 1 explained what dependencies are, and showed us how dependencies can be described with graph theory and modelled as a dependency graph. Graph theory allows us to perform computation on a graph, and can be useful to transform vague and hard to define relations into simple graphs and associated metrics. We also illustrated the problem of discovering and mapping dependencies, a problem which has no good solutions except hard thinking and careful analysis.

Dependencies were analysed in two different contexts. In Chapter 2, we looked at dependencies in source code, and especially dependencies to libraries and frameworks. We saw how indirect dependencies can make the dependency graph larger than first thought, and how modern package managers can exacerbate this problem. A large dependency graph can lead to larger programs, and more lines of code are correlated with higher complexity, more vulnerabilities, and more defects. We have also studied a novel attack technique—cross-build injection—that attacks the build process in modern build systems and package managers.

The other context in which dependencies are especially relevant, is dependencies in infrastructures, which we examined in Chapter 3. We saw how dependencies in infrastructures can cause problems with reliability and lead to cascading failures. Complex adaptive systems are especially vulnerable, since they consists of a large number of interacting components.

Systems can also have vague, indirect or even undefined dependencies, which are hard to discover, but still have far-reaching consequences. Cloud computing platforms and service-oriented architectures are especially conducive to outsourcing work to dependencies, and we examined the problems of the resulting dependencies.

Trust is an important part of life, but trust in dependency relations has received relatively little study. In Chapter 4, we looked closer at how trust can manifest itself in dependency graphs, and how trust in dependency graphs differ from trust in trust chains. We examined the Internet web of trust, and how transitive trust in this web leaves the Internet less resilient as a consequence.

Chapter 5 looked at several case studies, and evaluated tools that are useful when analysing and reasoning about dependency graphs. The Dynamic Presentation Generator system was analysed, and the security mechanisms in modern package managers were compared, and mostly found to be lacking. The dependency graphs within `npm` were analysed, and we defined the concept of a dependency explosion.

As we didn't want to leave the reader with a sense of despair, Chapter 6 contained reviews of existing solutions, and suggestions for possible solutions to the problems brought to light by the previous five chapters. Especially interesting are the solutions that embrace failure. We focused on attempts to introduce resiliency to systems by introducing controlled levels of failure.

7.2 Conclusions

It is impossible to write completely secure software, and it is impossible to build a meaningful system that is totally independent of its surroundings. However, the number of dependencies and the size of the dependency graph should be kept as small as possible.

Too many dependencies will create a scenario in which the dependencies dominate the complexity, reliability and vulnerability of the system.

Isolating the system as much as possible from its dependencies is paramount. The impact of dependencies on a system is large, and reducing it to a minimum is very important.

A good development methodology is needed to keep track of dependencies and avoid having dependencies with known vulnerabilities.

7.3 Extensions of thesis work

Given more time, I would:

- Create better and more robust modelling tools, and examine how complex adaptive systems should be modelled. `depend.py` could be extended to be more precise, faster, and more interactive.
- Attempt to completely define a "dependency-managed lifecycle" for use in software development processes.
- Implement the "ideal" package manager outlined in Section 6.1.1, either directly or as an extension of one of the existing package managers.

7.4 Open research questions

These are open questions which would be worthy of further study.

Can complex adaptive systems learn to become increasingly robust over time?

A complex adaptive system has an equilibrium outside the desired state, and must be actively maintained and adjusted to be kept in this state. Can a complex adaptive system be designed to have the desired state as the equilibrium point, or to have the desired state as an equilibrium point given a (small) set of prerequisites, such as "at least 40% of the nodes must have power", or to have an equilibrium that meets at least a subset of the requirements of the desired system state, such as "at least 35% of the services are up and responding" or "the response time is below 2 seconds"? Meeting this set of requirements will thus be sufficient to guarantee stability and robustness.

The biosphere on earth is an example. It is a complex adaptive system that appears to be very robust. Although the biosphere will return to an equilibrium that doesn't necessarily benefit a specific race (as seen in numerous mass extinction events), it does suggest that a complex adaptive system can be very robust if you remove enough constraints and requirements.

High robustness may be possible because every single individual and race want to survive, and because evolution makes individuals adapt to the ever-changing environment. Could evolutionary algorithms combined with systems that exert pressure (ChaosMonkey-like) allow nodes in a computer system to adapt and successfully provide increasing robustness over time?

The book "Antifragile: Things That Gain from Disorder" by Nassim Taleb [143] discusses the topic of increasing robustness over time by embracing failure and disorder, and is a strongly recommended read.

Appendix A

Source code listings

This appendix includes the source of the most important scripts and programs I have written during the thesis work. In order to avoid formatting issues, each script is on its own separate page, and non-essential elements have been removed.

```python
import requests
import json
import networkx

package = "yeoman"

def get_dependencies_of_package(graph, packageName):
    url = "http://isaacs.iriscouch.com/registry/%s" % (packageName)
    res = requests.get(url)
    data = json.loads(res.content)
    if "versions" in data:
        versions = data["versions"]
        newestKey = sorted(versions).pop()

        if "dependencies" in versions[newestKey]:
            dependencies = versions[newestKey]["dependencies"]

            for dependency in dependencies:
                graph.add_edge(packageName, dependency)
                print "add_edge(%s, %s)" %(packageName, dependency)
                get_dependencies_of_package(graph, dependency)

G = networkx.DiGraph()

get_dependencies_of_package(G, package)

print G.size()
networkx.write_pajek(G, "%s-graph.net" % (package))
```

Listing A.1: *Source code of the script used to fetch the most-depended-upon packages in* `npm`*, and export it as a graph file in Pajek format.*

```
def availability_simulation(G, iterations = 10000):
    # Preprocess graph
    for n, data in G.nodes(data=True):
        data['failures_left'] = 0
        data['num_failed'] = 0
        data['failed'] = False

    for i in range(iterations):
        # for each iteration, simulate failure according to the defined sla
        for n, data in G.nodes(data=True):
            if should_fail(n):
                fail_node(n)
            else:
                data['failed'] = False

        if i % 100000 == 0 and i != 0:
            print 'Ran %d iterations, summary:' % i
            for n, data in G.nodes(data=True):
                data['uptime'] = 100 - ((data['num_failed']*1.0 / i*1.0) * 100)
                pprint((n, data), width=160)

def fail_node(n, dependent=False):
    if G.node[n]['failures_left'] == 0:
        if not dependent:
            G.node[n]['failures_left'] = get_downtime()
    else:
        G.node[n]['failures_left'] -= 1

    G.node[n]['num_failed'] += 1
    G.node[n]['failed'] = True

    for v in G.predecessors(n):
        if not G.node[v]['failed']:
            fail_node(v, dependent=True)

def should_fail(n):
    return random.uniform(0, 100) > G.node[n]['sla'] or G.node[n]['failures_left'] > 0

def get_downtime():
    return math.ceil(numpy.random.lognormal(0.5 , 0.8))

G = load_graph('testgraph.json')
availability_simulation(G, 1000000)

draw_graph.draw_graph(G)
```

Listing A.2: *The source code of* ***depend.py****, the tool built to model outage events in dependency graphs of infrastructures.*

Bibliography

[1] L. Lamport, "Email message sent to the DEC SRC bulletin board," May 1987, http://research.microsoft.com/en-us/um/people/lamport/pubs/distributed-system.txt.

[2] P. Lenssen, "Why good programmers are lazy and dumb," http://blogoscoped.com/archive/2005-08-24-n14.html, August 2005.

[3] J. M. Johansson, "Island hopping: Mitigating undesirable dependencies," *TechNet Magazine*, February 2008, http://technet.microsoft.com/en-us/magazine/2008.02.securitywatch.aspx.

[4] B. Du Bois, S. Demeyer, and J. Verelst, "Refactoring-improving coupling and cohesion of existing code," in *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pp. 144–151, IEEE, 2004.

[5] "Wikipedia: Dependency hell," http://en.wikipedia.org/wiki/Dependency_hell.

[6] R. Cook, "How complex systems fail," *Cognitive Technologies Laboratory, University of Chicago. Chicago IL*, 1998, http://www.ctlab.org/documents/How%20Complex%20Systems%20Fail.pdf.

[7] P. Bahl, P. Barham, R. Black, R. Chandra, M. Goldszmidt, R. Isaacs, S. Kandula, L. Li, J. MacCormick, D. A. Maltz, *et al.*, "Discovering dependencies for network management," in *ACM SIGCOMM 5th Workshop on Hot Topics in Networks (Hotnets-V)*, pp. 97–102, Citeseer, 2006.

[8] S. Basu, F. Casati, and F. Daniel, "Web service dependency discovery tool for SOA management," in *Services Computing, 2007. SCC 2007. IEEE International Conference on*, pp. 684–685, IEEE, 2007.

[9] F. Baiardi, S. Suin, C. Telmon, and M. Pioli, "Assessing the risk of an information infrastructure through security dependencies," in *Critical Information Infrastructures Security*, pp. 42–54, Springer, 2006.

[10] yWorks, "yEd Graph Editor," `http://www.yworks.com/en/products_yed_about.html`.

[11] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical society*, 7(1):pp. 48–50, 1956.

[12] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, 1(1):pp. 269–271, 1959.

[13] J. Williams and D. Wichers, "OWASP Top 10 - 2013, A9-Using components with known vulnerabilities," Technical report, OWASP Foundation, June 2013, `https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities`.

[14] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," Technical report, Aspect Security Inc, March 2012, `https://www.aspectsecurity.com/uploads/downloads/2012/03/Aspect-Security-The-Unfortunate-Reality-of-Insecure-Libraries.pdf`.

[15] F. Long, D. Mohindra, R. C. Seacord, D. F. Sutherland, and D. Svoboda, *The CERT Oracle Secure Coding Standard for Java*, Addison-Wesley Professional, 2011.

[16] B. Schneier, "Random number bug in debian linux," `https://www.schneier.com/blog/archives/2008/05/random_number_b.html`, May 2008.

[17] A. Bridgwater, "Indirect dependencies are killing open source licenses," May 2013, `http://www.drdobbs.com/open-source/indirect-dependencies-are-killing-open-s/240154702`.

[18] US-CERT/NIST, "Vulnerability summary for CVE-2011-4969," `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2011-4969`, March 2013.

[19] "Dojo toolkit," `http://dojotoolkit.org/`.

[20] "Dojo toolkit redirection weaknesses and cross-site scripting," *Kaspersky Labs Securelist*, March 2010, `http://www.securelist.com/en/advisories/38964`.

[21] J. Williams and D. Wichers, "OWASP Top 10 - 2013, The ten most critical web application security risks," Technical report, OWASP Foundation, June 2013, `https://www.owasp.org/index.php/Top_10_2013-Top_10`.

[22] M. Goldmann, "Creative usernames and Spotify account hijacking," *Spotify Labs*, June 2013, `http://labs.spotify.com/2013/06/18/creative-usernames/`.

[23] M. Watson, "Malware detection in the context of cloud computing," in *The 13th Annual PostGraduate Symposium on The Convergence of Telecommunications, Networking and Broadcasting (PGNet 2012)*, 2012.

[24] N. McAllister, "Study shows half of all websites use jQuery," `http://www.theregister.co.uk/2012/08/14/jquery_used_by_half_of_all_websites/`, August 2012.

[25] W3Techs, "Usage statistics and market share of jquery for websites," `http://w3techs.com/technologies/details/js-jquery/all/all`, July 2013.

[26] D. Easley and J. Kleinberg, *Networks, Crowds, and Markets: Reasoning about a highly connected world*, Wiley Online Library, 2012.

[27] "Google Hosted Libraries," `https://developers.google.com/speed/libraries/`.

[28] "Usage of JavaScript content delivery networks for websites," `http://w3techs.com/technologies/overview/content_delivery/all`, July 2013.

[29] "Lines of code," `http://c2.com/cgi/wiki?LinesOfCode`, January 2013.

[30] L. Hatton, "The role of empiricism in improving the reliability of future software," Presented at Testing: Academic and Industrial Conference 2008, August 2008, `http://www.leshatton.org/Documents/TAIC2008-29-08-2008.pdf`.

[31] C. Perrin, “The danger of complexity: More code, more bugs,” *TechRepublic*, February 2010, `http://www.techrepublic.com/blog/security/the-danger-of-complexity-more-code-more-bugs/3076`.

[32] J. Rosenberg, “Some misconceptions about lines of code,” in *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pp. 137–142, IEEE, 1997.

[33] E. W. Dijkstra, “On the cruelty of really teaching computing science,” *Unpublished manuscript EWD*, 1036, 1988.

[34] B. Chelf and A. Chou, “Controlling software complexity,” Technical report, Coverity, 2008, `http://www.coverity.com/library/pdf/ControllingSoftwareComplexity.pdf`.

[35] F. Cook, “Coverity’s kernel code quality study,” *Linux Weekly News*, December 2004, `http://lwn.net/Articles/115530/`.

[36] O. Alhazmi, Y. Malaiya, and I. Ray, “Security vulnerabilities in software systems: A quantitative perspective,” in *Data and Applications Security XIX*, pp. 281–294, Springer, 2005.

[37] O. H. Alhazmi, Y. K. Malaiya, and I. Ray, “Measuring, analyzing and predicting security vulnerabilities in software systems,” *Computers & Security*, 26(3):pp. 219–228, 2007.

[38] B. Chess, F. DeQuan Lee, and J. West, “Attacking the build through cross-build injection,” Technical report, Fortify Software, 2007, `https://www.fortify.com/downloads2/public/fortify_attacking_the_build.pdf`.

[39] S. Mak, “Cross-build injection attacks: how safe is your build?” `http://branchandbound.net/blog/security/2012/03/crossbuild-injection-how-safe-is-your-build/`, March 2012.

[40] L. Constantin, “Compromised sourceforge mirror distributes backdoored phpmyadmin package,” *IDG News Service – Computerworld*, September 2012, `http://news.idg.no/cw/art.cfm?id=22A830B5-0751-6533-C4281B3BDD5EB8E2`.

[41] K. Thompson, “Reflections on trusting trust,” *Communications of the ACM*, 27(8):pp. 761–763, 1984.

[42] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*, Pearson Education, 2010.

[43] "Veracode analytics website," `https://www.veracode.com/products/application-security-analytics.html`.

[44] E. Messmer, "New standard/tool address security dependencies," *Network World*, December 2012, `http://www.networkworld.com/news/2012/121412-dependency-modeling-standard-265125.html`.

[45] I. Dobson and J. Hietala, "Operational resilience through managing external dependencies," *The Open Group Blog*, December 2012, `http://blog.opengroup.org/2012/12/10/operational-resilience-through-managing-external-dependencies/`.

[46] "Dependency modeling (O-DM) standard," *The Open Group*, 2012, `https://www2.opengroup.org/ogsys/catalog/C133`.

[47] "Dependency modelling - why iDepend?" Technical report, Cambrensis Ltd., 2012, `http://www.cambrensis.org/dependency-modelling/`.

[48] D. Geer, R. Bace, P. Gutmann, P. Metzger, C. P. Pfleeger, J. S. Quarterman, and B. Schneier, "Cyberinsecurity: The cost of monopoly," *Computer and Communications Industry Association (CCIA)*, 2003.

[49] S. M. Rinaldi, J. P. Peerenboom, and T. K. Kelly, "Identifying, understanding, and analyzing critical infrastructure interdependencies," *Control Systems, IEEE*, 21(6):pp. 11–25, 2001.

[50] "Sårbarhetsanalyse av mobilnettene i Norge," Technical report, Norwegian Post and Telecommunications Authority, January 2012, `http://www.regjeringen.no/upload/SD/sarbarhetsanalyse_mobilnettene.pdf`.

[51] J. H. Holland, "Studying complex adaptive systems," *Journal of Systems Science and Complexity*, 19(1):pp. 1–8, 2006.

[52] B. Laing, "Summary of Windows Azure service disruption on Feb 29th, 2012," *Azure MSDN Blog*, March 2012, `http://blogs.msdn.com/b/windowsazure/archive/2012/03/09/summary-of-windows-azure-service-disruption-on-feb-29th-2012.aspx?Redirected=true`.

[53] C. Williams, "How a tiny leap-day miscalculation trashed Microsoft Azure," *The Register*, March 2012, http://www.theregister.co.uk/2012/03/12/azure_leap_day_confirmed/.

[54] "Slik var trafikkaoset," *Bergensavisen*, August 30th 2013, http://www.ba.no/nyheter/article6835760.ece.

[55] BankID, "Java kjøremiljø på brukerens datamaskin," https://www.bankid.no/Dette-er-BankID/Trygg-bruk-av-BankID/Java-kjoremiljo-pa-brukerens-datamaskin/.

[56] D. Goodin, "Apple sneaks malware protection into Snow Leopard," *The Register*, August 2009, http://www.theregister.co.uk/2009/08/25/snow_leopard_malware_protection/.

[57] "Mozilla support: How to enable Java if it's been blocked," https://support.mozilla.org/en-US/kb/how-to-enable-java-if-its-been-blocked.

[58] J. Finkle, "U.S. warns on Java software as security concerns escalate," *Reuters*, January 2013, http://www.reuters.com/article/2013/01/11/us-java-security-idUSBRE90A0S320130111.

[59] J. Tanos, "Apple remotely blocks Java OS X web plugin for the second time," *the Mac Observer*, January 2013, http://www.macobserver.com/tmo/article/apple-remotely-blocks-java-os-x-web-plugin-for-the-second-time.

[60] BankID, "Apples safari har stengt for Java," https://www.bankid.no/Presse-og-nyheter/Nyhetsarkiv/2013/Apples-Safari-har-stengt-for-Java/, February 2013.

[61] J. Schuh, "Saying goodbye to our old friend NPAPI," *Chromium Blog*, September 2013, http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html.

[62] BankID press release, "BankID 2.0 blir Java-fri," April 2013, https://www.bankid.no/Presse-og-nyheter/Nyhetsarkiv/2013/BankID-20-blir-Java-fri/.

[63] D. Farber, "Oracle's Ellison nails cloud computing," *CNET*, September 2008, http://news.cnet.com/8301-13953_3-10052188-80.html?part=rss&subj=news&tag=2547-1_3-0-5.

[64] P. Mell and T. Grance, "The NIST definition of cloud computing," *NIST special publication 800-145*, pp. 6–7, September 2011.

[65] "What is Google App Engine?" `https://developers.google.com/appengine/docs/whatisgoogleappengine`, September 2013.

[66] M. N. Huhns and M. P. Singh, "Service-oriented computing: Key concepts and principles," *Internet Computing, IEEE*, 9(1):pp. 75–81, 2005.

[67] R. C. Martin, *Agile software development: principles, patterns, and practices*, Prentice Hall PTR, 2003.

[68] E. S. Raymond, "cargo cult programming: n." *The Jargon File*, `http://www.catb.org/jargon/html/C/cargo-cult-programming.html`.

[69] S. Marsh and M. R. Dibben, "The role of trust in information science and technology," *Annual Review of Information Science and Technology*, 37(1):pp. 465–498, 2003.

[70] D. Johnson and K. Grayson, "Cognitive and affective trust in service relationships," *Journal of Business research*, 58(4):pp. 500–507, 2005.

[71] B. Alcalde and S. Mauw, "An algebra for trust dilution and trust fusion," in *Formal Aspects in Security and Trust*, pp. 4–20, Springer, 2010.

[72] J. Hawkinson and T. Bates, "RFC 1930: Guidelines for creation, selection, and registration of an autonomous system (AS)," Technical report, IETF, March 1996, `https://tools.ietf.org/html/rfc1930#section-3`.

[73] M. Kühne, "Update on AS path lengths over time," *RIPE NCC Blog*, September 2012, `https://labs.ripe.net/Members/mirjam/update-on-as-path-lengths-over-time`.

[74] A. Toonk, "Accidentally stealing the internet," *BGPmon Blog*, January 2013, `http://www.bgpmon.net/accidentally-stealing-the-internet/`.

[75] D. Turk, "RFC 3882: Configuring BGP to block denial-of-service attacks," Technical report, IETF, September 2004, `http://tools.ietf.org/html/rfc3882`.

[76] "Router glitch cuts net access," *CNet News*, April 1997, `http://news.cnet.com/2100-1033-279235.html`.

[77] M. Prince, "Today's outage post mortem," *CloudFlare Blog*, March 2013, http://blog.cloudflare.com/todays-outage-post-mortem-82515.

[78] E. Zmijewski, "Reckless driving on the internet," *Renesys Blog*, February 2009, http://www.renesys.com/2009/02/the-flap-heard-around-the-world/.

[79] J. Vijayan, "How a small Czech company (accidentally) caused a global internet disruption," *Computerworld*, February 2009, http://www.computerworld.com/s/article/9128478/How_a_small_Czech_company_accidentally_caused_a_global_Internet_disruption?intsrc=news_ts_head.

[80] S. Gallagher, "How an Indonesian ISP took down the mighty Google for 30 minutes," *ars technica*, November 2012, http://arstechnica.com/information-technology/2012/11/how-an-indonesian-isp-took-down-the-mighty-google-for-30-minutes/.

[81] T. Paseka, "Why Google went offline today and a bit about how the internet works," http://blog.cloudflare.com/why-google-went-offline-today-and-a-bit-about, November 2012.

[82] M. Brown, "Pakistan hijacks YouTube," *Renesys Blog*, February 2008, http://www.renesys.com/2008/02/pakistan-hijacks-youtube-1/.

[83] American Registry for Internet Numbers, "List of autonomous systems numbers and names of all registered ASNs." ftp://ftp.arin.net/info/asn.txt, September 2013.

[84] "GitHub: Online project hosting using Git." https://github.com/.

[85] "Google Code: Google's official developer site. featuring APIs, developer tools and technical resources." https://code.google.com/.

[86] "CodePlex: Project hosting for open source software," https://www.codeplex.com/.

[87] "GitHub help: Fork a repo," https://help.github.com/articles/fork-a-repo, July 2013.

[88] MrMEEE, “GIANT BUG... causing /usr to be deleted... so sorry....” https://github.com/MrMEEE/bumblebee-Old-and-abbandoned/commit/a047be85247755cdbe0acce6f1dafc8beb84f2ac, May 2011.

[89] “Running ”n stable” removed bin, lib, share and include directories from /usr/local,” https://github.com/visionmedia/n/issues/86, October 2012.

[90] “Security: HTTP(S) connections to registry.npmjs.org are not validated, package database not validated; DNS injection could compromise users of npm,” https://github.com/isaacs/npm/issues/1204#issuecomment-1696578, July 2011.

[91] “Shooting the messenger,” *Secunia Research Blog*, July 2013, https://secunia.com/blog/372/.

[92] B. Schneier, “When it comes to security, we’re back to feudalism,” *Wired*, November 2012, http://www.wired.com/opinion/2012/11/feudal-security/.

[93] G. Jacobs, *Clearing the Sky in Cloud Computing*, Master’s thesis, Eindhoven University of Technology, http://alexandria.tue.nl/extra2/afstversl/tm/Jacobs_2012.pdf.

[94] E. Kirwan, “What is good source code structure? three objective, syntactic properties.” http://edmundkirwan.com/general/goodstructure.html.

[95] E. Kirwan, “Spoiklin Soice. a documentation and static analysis tool.” http://edmundkirwan.com/general/spoiklin.html.

[96] M. Gousset, “Use code maps to understand code relationships,” *Visual Studio Magazine*, April 2013, http://visualstudiomagazine.com/articles/2013/04/25/use-code-maps-to-understand-code-relationships.aspx.

[97] “Visual Studio Ultimate 2012 med MSDN - Microsoft Norge nettbutikk,” https://www.microsoft.com/visualstudio/eng/buy, October 2013.

[98] “Visualize and understand code with code maps in visual studio,” *MSDN Visual Studio*, http://msdn.microsoft.com/en-us/library/vstudio/jj739835.aspx.

[99] S. Tisue and U. Wilensky, "NetLogo: A simple environment for modeling complexity," in *International Conference on Complex Systems*, pp. 16–21, 2004.

[100] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using NetworkX," Technical report, Los Alamos National Laboratory (LANL), 2008.

[101] Y. Espelid, *Dynamic Presentation Generator*, Master's thesis, Department of Informatics, University of Bergen, 2004.

[102] "Ohloh, the open source network," `http://www.ohloh.net/`.

[103] D. A. Wheeler, "SLOCCount," `http://www.dwheeler.com/sloccount/`.

[104] M. Gunderloy, "RubyGems guides - publishing your gem - gem security," `http://guides.rubygems.org/publishing/#gem_security`, February 2013.

[105] "Node packaged modules," `https://npmjs.org/`, July 2013.

[106] "Security alert - please reset your npm registry account," `https://gist.github.com/jashkenas/2001456`, March 2012.

[107] K. Finley, "Node package manager accidentally leaks developers' password hashes," *SiliconANGLE*, March 2012, `http://siliconangle.com/blog/2012/03/08/node-package-manager-accidentally-leaks-developers-password-hashes/`.

[108] P. Irish, A. Osmani, S. Sorhus, M. Daniel, E. Bidelman, F. Ros, B. Ford, P. Hartig, and S. Sawchuk, "Yeoman - modern workflows for modern webapps," `http://yeoman.io/`.

[109] "npmjs.org - most depended-upon packages," `https://npmjs.org/browse/depended/0/`.

[110] M. Bastian, S. Heymann, and M. Jacomy, "Gephi: An open source software for exploring and manipulating networks," 2009, URL `http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154`.

[111] "NuGet.org: Search for 'bootstrap'," `http://www.nuget.org/packages?q=bootstrap`.

[112] "The node security project," Technical report, UpLift Security, April 2013, `http://nodesecurity.io/`.

[113] "Node security project: View advisories," `https://nodesecurity.io/advisories`, October 2013.

[114] T. Preston-Werner, "Semantic Versioning 2.0.0," `http://semver.org/spec/v2.0.0.html`, June 2013.

[115] "Semantic Versioning," Technical report, OSGi Alliance, 2010, `http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf`.

[116] "NuGet docs - versioning," `http://docs.nuget.org/docs/reference/versioning`.

[117] "`npm` documentation - package.json," `https://npmjs.org/doc/json.html#version`.

[118] E. Oftedal, "OWASP SafeNuGet," `https://github.com/OWASP/SafeNuGet`, June 2013.

[119] E. Oftedal, "NuGet developer: Help me help you," `http://erlend.oftedal.no/blog/?blogid=138`, June 2013.

[120] E. Oftedal, "Github: `SafeNuGet/blob/master/feed/unsafepackages.xml`," `https://github.com/OWASP/SafeNuGet/blob/master/feed/unsafepackages.xml`.

[121] "Mozilla developer network: XPConnect wrappers," `https://developer.mozilla.org/en-US/docs/XPConnect_wrappers`, August 2013.

[122] J. Raphael, "The worst cloud outages of 2013 (so far)," *InfoWorld*, July 2013, `http://www.infoworld.com/slideshow/107783/the-worst-cloud-outages-of-2013-so-far-221831`.

[123] "Heroku uptime status," `https://status.heroku.com/uptime`.

[124] "What is AppScale?" `https://github.com/AppScale/appscale/wiki`, October 2013.

[125] J. Bloomberg, "Cloud brokering: Building a cloud of clouds," *ZapThink*, April 2011, `http://www.zapthink.com/2011/04/19/cloud-brokering-building-a-cloud-of-clouds/`.

[126] "PagerDuty website," http://www.pagerduty.com/, October 2013.

[127] "Outage post-mortem," *PagerDuty Blog*, August 2011, http://blog.pagerduty.com/2011/08/outage-post-mortem/.

[128] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population." in *FAST*, volume 7, pp. 17–23, 2007, http://research.google.com/archive/disk_failures.pdf.

[129] J. Allspaw, "MTTR is more important than MTBF (for most types of F)," http://www.kitchensoap.com/2010/11/07/mttr-mtbf-for-most-types-of-f/, November 2010.

[130] J. Allspaw, "Resilience engineering: Part i," http://www.kitchensoap.com/2011/04/07/resilience-engineering-part-i/, April 2011.

[131] B. Christensen, "Introducing Hystrix for resilience engineering," http://techblog.netflix.com/2012/11/hystrix.html, November 2012.

[132] B. Christensen, "Fault tolerance in a high volume, distributed system," http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html, February 2012.

[133] B. Schmaus, "Making the Netflix API more resilient," *The Netflix Tech Blog*, December 2011, http://techblog.netflix.com/2011/12/making-netflix-api-more-resilient.html.

[134] Y. Izrailevsky, "The netflix simian army," *The Netflix Tech Blog*, July 2011, http://techblog.netflix.com/2011/07/netflix-simian-army.html.

[135] A. Hertzfeld, "Monkey lives," http://www.folklore.org/StoryView.py?project=Macintosh&story=Monkey_Lives.txt, October 1983.

[136] J. Ciancutti, "5 lessons we've learned using AWS," *The Netflix Tech Blog*, December 2010, http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html.

[137] "Microsoft SDL, practice #12: Perform fuzz testing," https://www.microsoft.com/security/sdl/process/verification.aspx.

[138] P. G. Sarkar and S. Fitzgerald, "Attacks on ssl: A comprehensive study of BEAST, CRIME, TIME, BREACH, Lucky 13 and RC4 biases," Technical report, iSEC Partners, Inc, August 2013, `https://www.isecpartners.com/media/106031/ssl_attacks_survey.pdf`.

[139] "Microsoft security development lifecycle," `http://www.microsoft.com/security/sdl/default.aspx`.

[140] C. Systems, "Cisco secure development lifecycle (CSDL)," `http://www.cisco.com/web/about/security/cspo/csdl/index.html`.

[141] P. C. et al., "Open software assurance maturity model," Technical report, OWASP, 2009, `http://www.opensamm.org/downloads/SAMM-1.0.pdf`.

[142] B. W. Kernighan and P. J. Plauger, "Software tools," *ACM SIGSOFT Software Engineering Notes*, 1(1):pp. 15–20, 1976.

[143] N. N. Taleb, *Antifragile: things that gain from disorder*, Random House Digital, Inc., 2012.

List of Figures