A comprehensive view of Software Bill of Materials

Lars Bendix[1,2], Andreas Göransson[1,3]

Abstract:
Recent years have seen a lot of interest in the concept of Software Bill of Materials (SBoM). Most of the recent work on SBoM has been solely from a cyber security perspective. Even if that work is not completely wrong, it is neither completely right nor does it give the full idea of what an SBoM is and what it can be used for. The cyber security focus gives a very narrow scope that has limited relevance to Software Configuration Management (SCM) and software configuration managers. It feels like SBoM has been stolen from SCM. With this white paper, we want to bring home SBoM and re-establish it as an important SCM concept and tool. In general, knowledge and information about SBoM is rather scattered and difficult to get hold of. Likewise, there are many differing opinions about exactly what an SBoM is and how it can be used. So, in this white paper we also want to give a comprehensive presentation of existing knowledge and practice about SBoM.

We did an extensive literature study to collect as much "theoretical" information about SBoM as possible. We also conducted a number of interviews with software configuration managers, developers and other stakeholders of SBoM to gather "practical" knowledge and experience, and wishes for use. Finally, we did an analysis on the obtained data to give some hints of what requirements need to be satisfied for different use cases.

We collected a large number of different use case examples that were then grouped into 10 more general use case categories. We sketched what requirements are needed for implementing each of the use case categories. We also identified a number of general issues that need to be taken into consideration for the smooth implementation of SBoMs. With this information software configuration managers will be able to utilise and exploit the concept of SBoM to its full potential and provide better service and support for development teams, organisations and customers.

For unknown reasons it seems that SBoM had fallen out of use - we hope that with this white paper it will fall back into use in all its many different shapes and variants.

[1] Scandinavian Network of Excellence in Software Configuration Management, sneSCM.org (bendix@snescm.org)
[2] Department of Computer Science, Lund University, Sweden.
[3] QCM, Malmö, Sweden.

1. Introduction

We think it is time that *Software* Configuration Management (SCM) takes back control over the concept of *Software* Bill of Materials (SBoM).

For hardware products, the concept of a bill of materials (BoM) has been around for a long time. The BoM has been used for production in the factory, it contains information about alternative suppliers for different parts and so on. The BoM can also be used to handle problems reported "in the field". For example, a defective part in a car can lead to fatal catastrophes. Once the problem has been identified, the affected cars have to be recalled to have the defective part replaced with a new one. Each car produced has its own individual BoM that keeps track of exactly what parts were used to produce this car. For some cars we might have used an alternative supplier for a part, for others we might have changed the design to use a slightly different part. Using these individual BoMs will allow us to identify all the affected cars and recall only those cars and not all cars ever produced.

For software, it seems that the use of a BoM is currently not as common as for hardware - and not as common as it used to be. Some companies provide an SBoM due to legal requirements while other companies have no or only a partial use of SBoM. It is an open - and interesting - question why SBoM is not so used anymore and has had a fading presence within the SCM community. Maybe modern development tools and environments have taken over some of the use cases. Maybe modern development methods and languages have made some use cases superfluous. Or maybe a declining number of software configuration managers and people with knowledge about and experience with SBoM in companies means that knowledge has fallen below the critical mass and the awareness of SBoM has slowly faded away. Whatever might be the reason, we hope that this white paper will remedy the situation and bring back SBoM into more prominent and wide-spread use. After all, SBoM is - if not by name then by concept - something that has been known and used in the SCM world for close to 40 years - if not more.

In recent years the National Telecommunications and Information Administration (NTIA) has done a lot of work on SBoM [NTIA]. Current interest in SBoMs comes primarily from the work of the NTIA based on an executive order from the president of the US. The NTIA did not invent the concept of an SBoM, which has always been a central concept in SCM. However, they surely revived the general interest for SBoMs. Unfortunately, all the attention that the executive order has gotten means that the general impression is that the only use of SBoMs is that of the NTIA - vulnerability scans. The work of the NTIA seems to ignore software configuration management and its 40 or so years of experience with SBoM, so in effect they have "reinvented the wheel". Apparently the SCM community has not done a good job in making known that SBoM is a central SCM concept and service that *they* know how to provide - and has never bothered to give a comprehensive exposition of all aspects of SBoMs and their use. The work from the NTIA gives a good and deep treatment of SBoM, but exclusively from a cyber security use aspect, which in our opinion is far too narrow given the many other use cases there are for an SBoM. The reality is that SBoMs is a far more varied and valuable "tool" than that. An SBoM is more complex than a "list of ingredients" and the list of possible use cases is much more varied and richer than "vulnerability scan".

When the NTIA considers an SBoM to be like a "list of ingredients", to a software configuration manager it sounds pretty much like a *baseline* - when they say that it shows how the different "ingredients depend on each other", to us it sounds a little like a *makefile* -

when they want to “trace what components are used in a product”, to us it sounds very much like *traceability* - all three very important and central SCM concepts. So, SBoM definitely has very strong relations to SCM - and as software configuration managers we need to know how we can best service and support all SBoM stakeholders and who they are.

The recent and narrow focus on cyber security has the consequence that people see this as the only use case for SBoM. Furthermore, people do not see it as an SCM expertise - in fact on WikiPedia the term SBoM redirects to “Software Supply Chain” - which ironically in a certain sense actually brings it into SCM (Supply Chain Management). Since both authors are SCM “purists” and have used/taught SBoM for decades and for everything else than vulnerability scans, this state somewhat disturbed us and we had to do something about it to rectify the situation. The renewed interest in SBoM has given us the possibility to spread “the full story of SBoM” to anyone interested. So, the goals of this white paper are: to demonstrate that SBoM has a long SCM history; to show that SBoM has a very broad set of use cases; to inform about general “implementation issues” for SBoM use cases; and to give an “informal definition” of what an SBoM is.

We carried out a literature study to trace back to the historic roots of SBoM and to compile as much knowledge as possible about BoM both in the context of software and also, for inspiration, from a couple of other contexts. We then conducted a number of interviews with industrial practitioners and stakeholders to complement and update the knowledge from the literature study with practical experience and stakeholder opinions. Finally, we did an analysis of the collected data to establish use case categories and some initial general considerations for implementations.

The primary target group for this white paper is software configuration managers. We will give them a deeper insight into what SBoMs are and their historic background. We will show them that SBoMs is a useful and very varied “tool” that can be used to service and support a broad group of different users. Finally, we will point to some general things that they must take into consideration before implementing SBoMs. The secondary target group are all other SBoM stakeholders. They will get an idea of what an SBoM is and a catalogue over use cases for SBoMs where they should be able to see how it can be useful for them.

With the results from this white paper it should be clear that the concept of SBoM is a central SCM principle and a useful tool for much more than just vulnerability scans. Our catalogue of SBoM use cases will allow stakeholders to find and request an SBoM and corresponding functionality that will enable them to do their daily work more easily and efficiently. The catalogue will also give software configuration managers insight into how they can use SBoMs to provide value to the people in their organisation and to the customers. In addition, our general considerations for working with SBoMs will give them information about what common issues they will have to address before they implement any SBoM use case.

In the following, we will first provide some background putting this white paper into context and motivating the use of SBoMs, making clear the delimitations to other SCM concepts and unravelling the history of SBoM. Then we will categorise the high number of specific use cases for SBoMs we found into a smaller and more manageable number of overarching use case categories, motivating the use case categories and giving indications for implementation. Finally, we will present some general considerations that emerged during our work with the use cases and the categories, considerations that will have to be addressed no matter what use case or category you want to implement.

2. Background

Before we dig deeper into the concept of an SBoM, its use cases and general considerations when implementing SBoMs, we would like to provide the reader with some background information. It will remind software configuration managers that SBoM is a valuable SCM concept that can be used for many different purposes and by different user groups. For others (and for software configuration managers that have not been introduced to SBoM before) it will give sufficient knowledge for them to be able to understand and find use for the rest of the white paper. To understand the "*theoretical origins*" of SBoM we will show how SBoM has roots in the traditional BoM from hardware and as such some similarities with BoM, but due to the different nature of developing and producing software there are also many differences. To understand the "*practical evolution*" of SBoM we will give a historic account of all the different uses over time while the SCM world was struggling with understanding how to make sense of BoM in a software context. Put together this will give a deeper understanding of why and how SBoM has evolved and been adapted from BoM.

In this white paper, we consider an SBoM to be strictly connected to "something" and to contain data/information about interesting and relevant aspects of that "something". We will try to consistently use the term "*object*" when we talk about the "something" that is described by the SBoM. However, we might occasionally also use the terms component, binary, application, executable or product.

In the following, we will first provide some context and motivation for the use of SBoMs. Then we will give an account of how we have traced the use of the concept BoM in the software world back to the early 1980s.

2.1 Context and motivation

Just like SCM was born out of CM, then SBoM was born out of BoM (even if our items to control are "soft" they are still items) - and just like SCM has found its own feet in the context of software, then SBoM has also found its own feet. Just like it was done for SCM, there were some aspects of SBoM that could be "lifted" verbatim from BoM for hardware, for other aspects BoM principles had to be adapted to the software context - and then there were some aspects that are particular for software and for which people had to come up with new principles - following the CM spirit. However, for many aspects the abstract principles are the same, but the way and degree to which they are and can be implemented will differ.

The concept of a BoM is much older than software engineering and much more generally applicable. Traditionally it has been used for producing hardware. More tangible physical things - not just computer hardware but also things like cars, trains and front loaders. However, at a more abstract level the concepts and principles behind the traditional BoM can also be applied to software and used as inspiration for how a *Software* Bill of Materials could work and be useful - and how it would differ from a traditional BoM given that the development and production of software for certain aspects differ from that of hardware. The following exposition of the traditional concept of a BoM is based on [Watts, 2000].

The BoM acts as an "interface" between Design Engineering, Manufacturing and Field Support. It allows them to "communicate" about and have a common vision of the product that is being designed, manufactured and maintained. The BoM module is made up of at least

two components: the parts information and the assembly information. Furthermore, the BoM can be “queried” for information about for instance how many parts a product has, where (in which products) a certain part is used or the combined weight of a product. The BoM also describes the parent-component relationships listing all the components the parent consists of and the use of “modular” BoMs allows you to abstract away the internal details of components that you want to consider as “black boxes”. Some parts will be interchangeable because they are identical given the level of details about “form, fit and function” stated in their BoM. It is recommended to have only one BoM information database to avoid redundancy and possible inconsistencies between multiple copies of the same piece of information. The level of detail and exactly what information should go into the BoM is open for discussion and depends on the specific context. Should fixtures and tools be included in the BoM? Are process consumables part of the BoM information? In any case, Configuration Management is responsible for all BoM input data and for the BoM accuracy.

Actually another “soft” industry has also picked up the BoM concept and adapted it to their context. That is the garment industry or more precisely the fashion industry [Harrop, 2021]. Adopting the “traditional” BoM to this context they see the need to also add a Bill of Process and even a Bill of Labour to obtain true traceability. This will allow brands and retailers to demonstrate that they are truly sustainable as they can show the impact of their material choices as well as their manufacturing processing.

Moving on to software development, the SBoM is related to a BoM for hardware in that it describes “what is in this product”. However, there are several aspects where software differs more or less radically from hardware:

- Creating multiple copies of a software product is easy - and they are identical
- During development software is "changed&built" much more frequently that hardware
- For hardware one specific “as designed BoM” can result in more “as manufactured BoMs”
- Information about the software and the production process can be recorded automatically in real-time
- Many different kinds of licences such as open source licences, proprietary licences to keep track of
- Software can change behaviour by changing a parameter during build time or install time
- Sometimes users can change the behaviour of software by configuration at run time
- Software is often updated by the user

These differences will also reflect in the details of what an SBoM is, how it can be created and for what it can be used. The “as designed SBoM” is also, but not limited to, a “list of ingredients” that will be used for manufacturing a product. In this case other than the source code, it will also contain build information. An “as manufactured SBoM” could be a simple “list of ingredients” that shows what is running in production, but it could also contain data about properties of the object or its parts. The fact that software is often built and run hundreds of times before it gets released to production means that most SBoMs will not be for what is in production. They will be for objects that run in QA, integration test, unit test or simply for “build and smoke” during development. So, these SBoMs are for objects that are created and consumed during development and before release to production. During the studies and the interviews it became clear that an SBoM can be many things. The SBoM can be a “build record” that is used by machines in a build chain with compilers and linkers, it

can be a document that needs to be approved by an audit from an external organisation and the SBoM can take many other forms.

However, for now we can give a simplistic “definition” of what an SBoM is. It is a record that is attached to an object and contains information about that object. Precisely what that is will have some similarities with hardware BoM, but there will also be many differences due to the nature of software. In its simplest case it is a list of the unique IDs of all the objects that this product is composed of - so for now let’s just say we are interested in what libraries and other components are contained in this product. We will see later - in section 2.2 and chapter 3 - that there are many other aspects about an object that could be interesting and useful and how this gut feeling "definition" will be made more precise and detailed. How much and exactly what information an SBoM should contain depends very much on the specific context and use case - in this white paper we will often refer to it as “to the level of detail needed”.

SBoMs can be very useful in many contexts and for many user groups. For a company it could be used as a sales pitch. Even for customers that do not require an SBoM as part of the product it can be a selling point. The US government may have the capability to scan an SBoM from time to time, but most customers will not have that capability - or time to do it. However, the fact that there is an SBoM for the product they buy and we promise to scan it for vulnerabilities from time to time may please them. As the use of an SBoM becomes more widespread and in some cases mandatory, it will probably become a standard in business-to-business relations. If we are using a third-party library in our software and do not have an SBoM for that, then we will not be able to provide a complete SBoM for our own software.

In addition to the users outside of the company or development organisation there are also internal users. As we shall see later there are many situations and stakeholders within a development organisation where an SBoM will come in handy in their day-to-day work. The possibility to have information about the objects that they handle and use can be very useful. It could be how an object was built, from what it is built, what licences it uses or its current status in the development life cycle. So, if a software configuration manager is able to provide targeted SBoMs to each different user group, SCM will more broadly deliver value both inside and outside an organisation.

Just like BoM is considered an important part of CM, then SBoM is also an important part of SCM. SCM is managing configurations and their parts and everything around them and how they are handled. SCM is keeping track of what objects went into released software products. As a consequence SCM is responsible for taking care of these objects and information about them and how this information can be changed and how it can be used.

There are several things from SCM that are related and connected to the concept of SBoM, but that we will not investigate further in this whitepaper. *Traceability*: An SBoM is not about general traceability - but it contains traceability information (if the SBoM is not flat). *Baseline*: It is not a baseline - but we start from a fixed configuration (the baseline for a product - the binaries) and then we are interested in the values of various attributes for the configuration items in the baseline (or the originating source code). *Makefile*: It is not a makefile - a makefile is prescriptive and also uses “implicit and external information” (like options and compiler version). An SBoM is descriptive and has all information explicitly baked in - and it is about a specific configuration and not a generic model. That said, we will treat SBoM in its own right in the remainder of this white paper to keep a strict focus.

Basically, SBoM is part of and very related to the Configuration Identification and Configuration Status Accounting (CSA) activities for a software configuration manager. So, instead of talking about “objects”, we should talk about “configuration items”. Instead of thinking of “information about objects”, we should think of it as “data for configuration items”. Instead of calling it “scanning for …”, we should call it “a CSA query on the Configuration Management Database (CMDB)”. So simply put, if an SBoM is “data about a configuration item” then that data should allow us to answer questions about the configuration item like “is log4j used in this configuration?”.

2.2 History of SBoM

The concept of an SBoM is not as new and narrow as the latest hype may lead you to believe. Let us try to trace the concept of an SBoM back in time.

The most recent interest comes from Executive Order 14028 issued by the president of the US on May 17, 2021. The cause for this executive order was probably a number of cyber security incidents from Heartbleed in April 2014 over SolarWinds in December 2020 and topped by the Colonial Pipeline hack on May 7, 2021. The objective was to strengthen the US’ ability to respond quickly and efficiently to any future such incidents. The result was a paper from the US Department of Commerce on “The Minimum Elements For a Software Bill of Materials” [USDC, 2021] published on July 12, 2021. The quick response was to a large degree possible because the NTIA had been working on issues around software component transparency specifically since 2018 and cyber security vulnerabilities in general since 2015 - caused by the Heartbleed incident - and published various results on SBoM between 2019 and 2021 [NTIA].

Standards - like the present IEEE Standard for Configuration Management in Systems and Software Engineering [IEEE828-2012] - say very little about SBoM. In the IEEE standard the only mention is: “Builds have been performed in accordance with the CMP in the following aspects: 1. Build sequence and procedure, 2. Bill of materials, 3. Build environment, 4. Build reproducibility, 5. Build identification (both input and output) and versioning”. An upcoming update of the SCM standard will probably contain a little more about SBoMs - unless SBoMs are completely left to the domain of cyber security.

In March 2014 Electric Bee wrote a blog titled: “Why you need a Bill of Materials” [Bee, 2014]. In that, he starts from the traditional view (from hardware products) of a BoM, but clearly shows that by extending that BoM idea into the software world, we can add more information to make an SBoM even more powerful and useful. In the blog he treats two software use cases: “Recreating a release” and “Escrow account”. In some industries you must maintain releases made months, years or even decades ago. When there is a problem, the customer wants you to fix the bug, the whole bug - and *nothing but the bug*. “A Bill of Materials allows Build & Release Engineers to identify the exact versions of all tools needed to recreate a particular release”. An Escrow Account allows a customer to take over and recreate a product if the vendor for some reason is no longer able to do so. However, as Bee points out a simple “list of ingredients” BoM is not sufficient, we also need information about “how to create the product from that raw material (such as your build instructions)”. In ElectricCommander (which is a build automation system that manages the build, test and release process), collecting data for the SBoM can be automated as the data “can be retrieved while the build is occurring, and can be stored with the build once it is completed”.

In the early-mid 2000s one of the authors used the concept of a SBoM for software at a major Scandinavian cell-phone producer to keep track of open source licences as well as commercial licences. Open source licences in order not to violate licences or to use licences that could mean that company source code had to be published to the public domain. Commercial licences in order to pay the correct licence fees. This activity is sometimes referred to as Software Asset Management.

In her book on Configuration management [Dart, 2000], Susan Dart does not have BoM or SBoM as an explicit index term. However, talking about builds the term is explicitly used: "Think of a build as being all the rules to create, such as executing a makefile that contains all the information about what to build and how to do it, and then all the results of executing the makefile (typically called *bill-of-materials*) such as the versions of files used in that build, the tools, the options on the tools, the libraries, the data sets, the template, the version of operating system and machine, and person responsible". And she goes on to say that: "Each build is given a unique identification so that the resultant configuration item (release, baseline) can be regenerated at any point in time given the bill-of-materials."

In the mid-late 1990s, a group of people were looking at how the idea of patterns could be applied to SCM matters. After publishing the seminal paper "Streamed Lines: Branching Patterns for Parallel Development" on branching and merging aspects, they went on to address "Software Reconstruction: Patterns for Reproducing Software Builds" [Cabrera-et-al-1999]. The most prominent pattern in that paper is called "Bill of Materials" and addresses the problem of "how you can reproduce the build if you know that more than your source code is required to build the software system". They are very clear that: "The bill of materials identifies what components you need, where they can be found, what version they are, and how to assemble them to reproduce the software system" - so, far more than a "list of ingredients". And they stress that: "the important purpose of the BOM is to identify components that are not under version control directly (e.g., environment information, compilers, linkers, et al)". Furthermore, they point out that at the time at least two of the big SCM tool providers, Continuous and ClearCase, had tools that could provide controlled, reproducible builds, ObjectMake and clearmake respectively.

In the mid-late 1990s one of the authors explicitly used the term BoM for software in the Software Configuration Management part of a course on Software Development Environments at Aalborg University, Denmark.

Digging out an old ClearCase manual from 1995 [Atria, 1995], we discovered that Bill of Material is an index term, but that it refers to Configuration Records. The main motivation of clearmake is better build avoidance than traditional make by allowing secure derived object sharing. To "guarantee that your builds in a parallel-development environment will be both correct (never reuse an object that is not appropriate) and optimal (always reuse an existing object that is appropriate)." There are the following functions connected to Configuration Records (CRs):

- catcr CR (displays the contents of the CR)
- diffcr CR1 CR2 (compares two existing CRs)

Earlier than that, David Leblang [Leblang, 1994] is very clear about what the build challenges of SCM are: "It also includes the automatic and reliable production of "bills-of-materials" (BOM) that document software system builds, and that can be used to recreate the

complete file system environment of any build." Also how detailed an SBoM should be: "After a derived object is produced, it must be possible to determine a reliable "bill-of-materials (BOM) describing all of the constituent parts of that object; ideally, the BOM should be absolutely correct and detailed enough to easily reconstruct the complete source environment at the time the object was built." And how SBoMs can be used: "In order to share derived objects, the build system must know exactly which source configuration the developer wants to build with, and have an exact bill-of-materials for all existing derived objects. Reuse of derived objects must be based on matching the developer's desired build configuration against the known configurations (BOMs) of the existing derived objects." and "Associated with each derived object is a bill-of-materials called a configuration record (config rec or CR) which clearmake can use during subsequent builds to decide whether or not the DO can be reused or shared."

In chapter 3 of his book on SCM [Babich, 1986], Wayne Babich writes: "The question "What program is this?" arises most frequently during debugging. Debugging traditionally involves an analysis of what a program does, performed by either reading the source code or observing its execution. But many times the fastest approach to finding a bug is not analysis of the program itself, but analysis of the *history* of the program - how it was created. The history of the program is called its *derivation*." Even though Babich does not explicitly call it an SBoM, his description of the concept of a derivation exactly captures that of an SBoM - and he identifies debugging as an important use case for SBoMs. Actually he states that "An ounce of derivation is worth a pound of analysis". After a careful description of how to record derivations, he goes on to treat a second use case for SBoMs: reproducibility.

In his PhD dissertation [Clemm, 1986], Geoffrey Clemm describes the Odin system which is an object manager that will respond to a request for an object by invoking the minimal number of tools necessary to produce that object. And where previously computed objects are automatically stored by the object manager for later re-use. Very simplistic said: a better and safer make, where a safe Equal function is used as a test for equality between two objects instead of unsafe timestamps. It captures the relationships between software objects which can be source code, compiled object code or test data. A parameter mechanism is used to capture that "there is a variety of additional information that can be associated with an object". It provides "a software object database" and "a tool to automate the process of object management". In substance it looks pretty much like a CMDB with configuration items that have data attached which allows the implementation of an SBoM query.

Here, in the early 1980s, we are as far back as we have had time and possibility to trace the conceptual idea of a BoM for software - and as far as we can see also earlier than people actually using the term SBoM (or BoM) in a software context.

3. Use case categories

In this chapter, we will present the use cases for SBoM that were found through literature, interviews with practitioners and our own experience. We want to show you the breadth and wealth of what SBoMs can be used for - to provide you with a catalogue of SBoM use case examples to be used verbatim or for inspiration.

After the available literature and the interviews with practitioners had been mined for use cases, we ended up with literally hundreds of more or less different use cases. This was clearly unmanageable and we started to look for ways to structure and organise the use cases for simplification and better overview. We looked for commonalities between groups of use cases and for ways to abstract away irrelevant details of similar use cases.

We identified three significant *aspects* of SBoMs: Materials, Process and Information. The *Materials* aspect is the actual contents of the object described by the SBoM - also called the "list of ingredients" by the NTIA. The *Process* aspect is what describes how an object was manufactured - or how to create a tasty pizza from the list of ingredients. The *Information* aspect captures all the data that can tell something interesting about an object (like supplier and price of the ingredients for the pizza) - or everything you always wanted to know about an object but were never able to ask.

For each of the three aspects, the relevant specific use cases have been grouped into one or more use case *categories* that capture the general characteristics of those specific use cases. A use case category will have a name to distinguish it. Furthermore, there will be a description motivating the category, a short definition of the category, a partial list of specific use cases that belong to the category, the intended users of the category and finally a sketch of the input data and the functionality that is needed to provide the desired output or result. Even though there is a distinction in the focus of the categories, there will also be certain overlaps between them and depending on the context there will be examples of specific use cases that could be placed in one category or another.

In the following, we will treat each of these three aspects - Materials, Process and Information - and for each of them present the use case categories they contain.

3.1 The Materials aspect

The first aspect of an SBoM is the Materials aspect - what are the single parts that make up the object covered by the SBoM. Very much like an IKEA packing list, that is probably the first thing you check when you open the box to see if they got it right - our experience is that they do. This use is quite similar to the traditional use of a BoM in the hardware business - and not just for computer hardware, but also if your product is a front end loader, a car, a dumper, a windmill or a (nuclear) power plant.

We will be interested in using the SBoM to search for the presence of a certain object in it. For that a unique identification of all objects that are parts of any SBoM is needed. Here we have to be a little careful about using the hardware metaphor also for software. In hardware identification is usually done by using part numbers for an “as designed BoM”, whereas serial or batch numbers are used in the “as built BoM” which means that many cars will have identical “as designed BoMs”, but most cars will have unique “as built BoMs”. In software

the “as designed SBoM” could be a baseline of the source code and the “as built SBoM” is the SBoM for the binary. In a software context part, serial and batch numbers do not make sense and for any given version of an application all copies of that application will have identical SBoMs. There can be various reasons for establishing the presence - or absence - of a given object in an SBoM - vulnerability scan being only one of the reasons.

So, in short, the “defining” characteristics for the Materials category is “searching for the presence of a specific object by its UID in the SBoM for a product” - where the product could be a third-party library.

For the Materials aspect there is only one use case category: “Search for an object by UID”.

Use case category 1: “Search for an object by UID”

Motivation:
Most often we think of an application or a library as one single entity - a “black box”. However, from time to time we could be interested in going into the “black box” and look at the details - all the different parts that make up the product.

In software development it is very common to use externally provided libraries as well as internally developed components that can be shared between several products and versions. When a “problem” is discovered with a library or shared component we need to figure out in which products and versions that library or component is used. It could be the simple case of scanning a product’s SBoM for the presence of “log4j” - either as a consumer of the product to be warned about a newly discovered vulnerability or as the producer of the product to be able to notify our customers and fix the problem. A scan of the UIDs of all the binary objects that are contained in the product of interest will give us the answer. As a consumer of one single product, you will scan the associated SBoM to ascertain whether the product is affected by a certain vulnerability or not. As a producer of products (and several versions of a product) you would often be interested in knowing where a certain object is used. For this you would have to systematically scan the SBoMs for all the versions of all your products. In both cases you could be affected by “blind spots” in case the SBoM for a product is not complete or sufficiently detailed. For instance, if there is not provided an SBoM for a third-party library that is used in a product.

It does not always have to be vulnerabilities that we want to scan for. It could also be operations that wonder if a certain patch has already been applied to the system in production. Or more in general aspects related to software composition analysis (SCA). The purpose of SCA is to get an overview of and insight into what components are used within a software application - both third-party components and internally developed components. It can be used as a dedicated step in the software quality process to verify that the components (and versions) development promised to deliver are also included in the release before it moves on to test and quality assurance.

If the SBoM is hierarchically structured, the search could also indicate the “level of depth” that we want to search. For a more detailed discussion of how SBoMs could be structured, see chapter 4.

Examples:
Risk management - evaluation

- Risk verification - when a new risk is discovered to answer questions like “Are we potentially affected?” and “Where is this piece of software used?”

Risk management - monitoring

- To monitor components for vulnerabilities

Transparency/communication

- To provide an SBoM to customers

Operations

- An SBoM allows an organization to understand what components are active on its systems and networks (from the operations perspective)

Intended users:
(Cyber) Security, SCA, Operations, developers.

Input data:

- Unique identification of object
- A structured or unstructured SBoM

Functionality:

- searchObject(UID, SBoM, depth)

Output result:

- searchObject returns a list of SBoMs where the UID is found - void if not found

3.2 The Process aspect

The second aspect of an SBoM is the *Process* aspect - how the object or product covered by the SBoM was built. This aspect actually also has a parallel in the traditional BoM that has been used in the hardware industry for ages. There they talk about things like “fixtures and tools”, “process consumables” and “manufacturing documentation” being part of the BoM data [Watts, 2000]. Very much like the “assembly instructions” from your IKEA package. For the production of software, it is even more important to have precise and detailed process information since even the tiniest little difference can sometimes make things go wrong. Even if it is not optimal, you can hammer a screw into a plank of wood, but you would get problems if you mixed binaries compiled with and without debug option. So, we should know all the details about how our binaries were created (from which source code and how) and how they were assembled into an object or product.

The Process aspect could capture things like build scripts, compilers, compilation options, operating systems, automated tests and other important information that were used to produce an object. A consequence of this is that old versions of scripts, compilers, operating systems, tests and other things need to be kept in case we want to use the SBoM to be able to exactly recreate an object at a later date. This means that just like the traditional BoM, the SBoM can also have a prescriptive use (*how to*) and not just be descriptive (*what*).

The purpose of the Process aspect is to be able to understand how a software object was built (and from what), how it was tested and verified, and also possibly recreate the software object long after it was originally released.

So, in short, the "defining" characteristics for the Process category is that the SBoM should also describe the process properties of an object. How it was built and from what to the level of detail that is of interest for our specific context. The purpose is to be able to decide if two objects are identical (not just for contents but also for manufacturing process) and if not to be able to tell what the difference is.

For the Process aspect we have identified four use case categories: Use, Troubleshooting, Build and Build audit.

Use case category 2a: "use of objects"

Motivation:

If someone brings a bookshelf they have made to our house, we simply mount it on the living room wall and then it is "integrated" in the house. For software things are not that easy. If there is a library or a component that we want to use and integrate in our code, we have to ask the compiler to rebuild "the house" to see if the "bookshelf and the other pieces" fit together and have someone (a tester) make sure that everything still works. Building your software from scratch every time a small change has been made can be quite time consuming. So, we need something smarter in order to get fast feedback on whether it works or not.

When we have made a change to a component then we need to compile only that component and all the components that depend on it. For the remaining components we can use the binaries we may have from a previous compilation. That is basically what make [Feldman, 1979] helps us do - to compile as little as possible and use as many already existing components as possible so we can have an executable as fast as possible. However, since make does not keep any data of its own and relies on timestamps for sources and binaries, it is not always an entirely safe approach. Sometimes timestamps may be unreliable, we might have changed the version of the compiler, or we might use different options when compiling this time. This is why developers do a complete and clean build from time to time. So, we need more data about the binary than a timestamp to be able to decide if it can be safely used or we should create a new binary. That data could be integrated in the SBoM for the binary object.

In a slightly different scenario developer B updates the workspace with the changes done by developer A. This means that for all the changed components the binary in developer B's workspace will become "out of date" and would have to be reproduced. However, developer A most probably compiled the whole system and will have up-to-date binaries for the components that were changed. If it were possible to "tap into" these binaries we would avoid that all other developers had to compile these components. They could simply use the binaries already produced by developer A. However, before the binaries can be used, we have to make sure that they have been produced exactly the way that *we* want them produced. For that we need an SBoM that describes exactly how the component has been derived to the level of detail that we are interested in. This is basically what is provided by clearmake [Atria, 1995] and other similar advanced build tools from that time. Not everyone is comfortable with using binaries produced by others as it might open up for security holes. So, this requires that we have secure, trustable SBoMs for the binaries or that we are in a "friendly" environment where we can be more relaxed.

When we go into the process details of how binaries have been manufactured, we also enter a situation where the SBoM will contain language and technology specific information. This will make it more complex to create a common, standard format for exchanging SBoMs. However, since the majority of Process use cases are for "internal" use only, it should not create problems if we tailor the format to our specific needs.

We considered naming this use case category "build avoidance" but settled on "use". However, in many cases it is build avoidance in the same sense as a hardware shop will try to avoid building its own nuts and bolts if it can buy them already built.

Examples:
Build re-use, build avoidance (effect of re-use), build in parallel and distributed builds, sharing within a project

- To decide that an existing artefact matches the requirements for a new one, the production of which may thus be avoided

Uniquely identifiable

- Uniquely identifying a build

Libraries (perhaps of third-party developer), sharing between different projects

- The build of a subsystem made in the context of one project should be re-usable as such in the context of any other project

Intended users:
Developers (in particular if they use TDD or CI/CD and don't have the patience to wait for a complete build to finish), testers (if they need to build in order to test), "intelligent/safe" build tools (so they can safely optimise the build process).

Input data:

- Precise description of the object that is needed in form of an SBoM

Functionality:

- find(SBoM_A)

Output result:

- An object that matches the SBoM_A, void if not found

Use case category 2b: "high-level troubleshooting"

Motivation:
In software development we don't always get things right the first time and when there is a problem, we initiate a process of troubleshooting to discover where the problem is and what caused it. If the problem is syntax or semantic errors, the compiler will tell us where the problem is and what we did wrong. If there is a problem with the logic, then there should be test cases that can tell us where the problem is and what goes wrong.

These are the easy problems. What if your application crashes immediately when it starts? The compiler does not see any problems and we don't get as far as being able to run the test cases. Situations like that and similar cases are not rare in software development. Without the help from the compiler and test cases it leaves us with no other choice than to "start in the upper left corner and read through 5 million lines of code" - not a pleasant choice.

What we would be interested in knowing is the difference between what is not working today and what was working yesterday. Since the tiniest little things can create problems in software, we will be interested in knowing even the minute details of these differences. We are not only interested in the differences in the source code used, but also differences in *how* the application was manufactured - and maybe even differences in the run-time environment. If it worked yesterday, but not today, then these differences seem like the obvious places to start hunting for the problem. Troubleshooting at a much higher level than just focusing on the source code - or as Wayne Babich put it "an ounce of [SBoM], is worth a pound of analysis" [Babich, 1986].

Examples:

- To answer the question "What program is this?" during debugging - the history of the program - how it was created - is called its derivation.
- "Analyzing the derivation is often a far faster way to find a bug than analyzing the code: An ounce of derivation is worth a pound of analysis" - so we should be able to work with - and diff - SBoMs.
- The build audit can be used to compare builds, allowing you to see what versions of what files are different between two builds or even what compiler options may have changed
- Reduce the overall amount of separately distinguishable artefacts and making real differences emerge

Intended users:
Developers

Input data:

- Two SBoMs

Functionality:

- diff(SBoM_A, SBoM_B)

Output result:

- Showing the differences between SBoM_A and SBoM_B

Use case category 2c: "building an object"

Motivation:
In the software world we are in the fortunate situation that when another client comes and buys one of our products, we don't have to go to the factory floor and instruct our employees to build the product. We can simply copy the executable that we sold to the previous client or it may be available for download. However, even in the software world things are not always as simple as that.

As mentioned above, we don't always get things right, in the software world and elsewhere. So, we will have to carry out maintenance of our products: corrective, adaptive, perfective and preventive maintenance. In many cases it is quite straightforward, you change the source code to fix a bug or add a new feature, you build the system using the latest version of the source code (possibly taking advantage of the functionality from Use case category 2a), and if everything tests ok you are done.

However, in some cases things are not as simple as that. If you have a long-term maintenance contract you might have to work with software for a petro-chemical plant or military equipment that is 20-30 years old. When you build the system after a maintenance change, you might not want to use the latest version of the source code since the customer is not interested in other changes than what they have requested. You might not want to use the latest compiler and development environment to build the system. The produced executable will have to run on very old hardware, and even if it runs, changes to the build process or environment might introduce subtle errors as explained for Use case category 2b above.

Another situation might be that a client using an important system is not sure that you will be able to maintain that system "forever". They could ask you to deposit the source code with a third-party agent such that someone else could take over the maintenance if you are no longer able to. This is known as source code escrow. In case that another development organisation has to take over the source code and maintain it, they would probably appreciate that there was also process and environment information about how to build the binaries and the executable from the source code.

Finally, precise and concise information about how a system was built and from what can also be used to avoid saving the binaries. This, however, requires that we have implemented reproducible builds which can be a hard problem to solve. The binaries can be re-created from the information in the SBoM and the source code from the repository. Depending on the level of detail and precision we want for the reproducible builds, we may also have to conserve old tools and environments that have been used as part of an SBoM. Even if disk space is cheap these days, we may not want to consider the binaries and executables as Configuration Items to be stored in the CMDB. The source code is already in the CMDB and an SBoM is very concise, prescriptive information about how to re-create a binary, so it could be considered as a CI that will be kept in place of the binary in the CMDB (or a binary artefact repository).

Examples:

- To reproduce an identical copy of any given program - for debugging or other purposes.
- Consistency and repeatability of builds
- Parallel and distributed builds - by understanding the build order dependency graph
- A record to make builds repeatable

Intended users:
Developers, testers (if they don't get a binary to test), build meisters

Input data:

- SBoM for an object to build

Functionality:

- build(SBoM_A)

Output result:

- the built object for the SBoM

Use case category 2d: “build audit”

Motivation:

When you have to carry out a build audit, you would like to base it on a correct and detailed account of what is in the built object. Does it contain what it should - no more, no less? However, a good and complete build audit should care about more than just the contents of an object. For instance, it is important to know what version of the compiler has been used and which options were used, to verify that the object has been built in a consistent way. Furthermore, it is important to know what source code was used to build the object - and the version of that source code. The amount and the level of detail of the information needed for the build audit depends on the context we are in. Is it software for a life-support machine or a small app we have developed to keep track of who should clean the stairway in our apartment building. This use case category has things in common with use case category 2b (troubleshooting) except that it has a different purpose and different intended users.

We believe that the SBoM for an object would be the perfect source of information for a build audit. The SBoM not only contains information about the contents of the object, but, as we have seen above, it also contains process information about how the object was constructed and from what. Furthermore, if we have automatically created SBoMs as part of the build process (see also chapter 4), we will have reliable and trustable information that will allow us to verify what *actually* happened and not what we *think* happened (or what *should have* happened). This way we have effectively eliminated the human factor - both for accuracy and for speed.

Examples:

- Build Audit - Often the traceability between the derived objects and the versions of the source used to produce them is lost, making it impossible to reproduce builds. A record is kept of who built each derived object, when it was built, on what platform it was built and, most important, what files and what versions of those files were referenced during the build. This information is called a *configuration record*. All the configuration records for all derived objects that are part of a build make up the ***build audit***. ClearCase provides the ability to generate a **bill of materials** for any build
- The benefit of clearmake is that you now have an audit trail telling you where you went wrong

Intended users:

QA, SCA, developers, testers

Input data:

- an SBoM to “build audit”

Functionality:

- possibilities to analyse the SBoM

Output result:

- None (apart from the “results” from the analysis)

3.3 The Information aspect

The third aspect of an SBoM is the *Information* aspect - what is the information that is interesting about the object covered by the SBoM and its constituent parts. Based on what it is that we want to know about an object, the SBoM for the object will have to contain data that can provide the knowledge we are looking for. Just like the other two aspects also the Information aspect has a parallel from the traditional BoM from hardware. There the questions that they want to have answered are things like: “What is the combined weight of an assembly?” and “Hours to produce?” [Watts, 2000]. Even if these questions do not carry over one-to-one to a software context, there are still plenty of interesting and useful questions that we could ask about software objects (besides “Does it contain log4j?”). Legal disputes may require an organisation to prove that the software was built and tested in a certain way and what were the results of the testing. When you release a new version of your software, you want it to have release notes detailing what is new and what has changed in this version.

So, in short, the “defining” characteristics for the Information categories are that attributes in the SBoM describe certain properties about the object. These properties can be searched/queried for in all the objects contained in a product - or the properties of an object can be summarised in a report.

Since there is so much diversity for this aspect, even within the same category, we have decided to leave out information about examples, intended users, input data, functionality and output result. From a specific context and use case it is straight-forward to derive that information.

For the Information aspect we have identified five use case categories: Licence tracking, Export control, Test-related matters, Legal aspects and Information sharing - but depending on your context (and curiosity) there could be more.

Use case category 3a: “licence tracking”

Motivation:

In modern software development it is not uncommon to use a lot of externally developed code. That external code comes with a licence that you have to agree to and respect both for free open source and for commercial third-party code. So, it becomes essential for a company to keep track of the licences it uses in its products. In the case of commercial licences, we don’t want to pay more than necessary and neither do we want to pay too little. In the case of open source licences, we do not want to violate the licence and we may not want our own code to become public domain.

The easiest way to keep track of what licences are actually in your product is to make licence information part of the SBoM for all relevant objects. So, the SBoM for a third-party library - open source or commercial - will have an entry called “licence” that contains information about its type. Then as part of the functionality that is provided for SBoMs, there should be a query that could search a given SBoM (and its objects) for the presence of a specific type of licence. So, through the SBoM we will be able to know what licence an object is covered by, but compliance with that licence is not part of an SBoM’s responsibility (though information about that could be captured in the SBoM).

Use case category 3b: “export control”

Motivation:

This use category is quite similar to the previous “licence tracking”. There is, however, one important difference and that is that the result from a query about export control might change as time passed. This means that the licence status for a certain library in version 3 will never change, it will remain *static*. There might be a change in status when moving to version 4 of the library, but that will not influence version 3. For the export control status, on the other hand, the situation will be more *dynamic* since how restrictive a certain export clause is might change due to “international events”. So, a library in a given version may be allowed for export to certain countries today, but not so tomorrow. In that respect, it looks very similar to the situation for “vulnerability scan”. That a certain object can be safe today but may not be safe tomorrow if a vulnerability is discovered. This means that when we deal with dynamic data, we cannot bake this data into the SBoM but will have to make it the result of a query and that the result has validity only at the point in time that the query is performed. See more about static and dynamic information in chapter 4.

Use case category 3c: “test related matters”

Motivation:

In many different contexts and situations, we would be interested in knowing the test status of a given object. Has testing been performed on this object? What version of the test cases and with what result? Who has performed the testing and signed it off? In what environment has the testing been performed? Given that we already have an SBoM that contains useful information about the object it seems obvious to also include test related information in the SBoM.

Use case category 3d: “legal aspects”

Motivation:

In case of a legal dispute, it might be useful to be able to prove down to minute details everything relevant about a software product. If a car has an accident or if medical equipment fails, it can be important for a software provider to be able to demonstrate that the problem was not due to an error in the software or the development process. To have a record that demonstrates that the software has been tested and the prescribed processes followed. Besides the “test related matters” from the use case category above, such a scenario will also have to address that the SBoM can describe the state at several points in time. Until now we have intended an SBoM to capture what is constructed at build-time. However, when we install a software package it is often parameterised and sometimes the end user also has possibilities to configure the software at run-time. To be able to win a legal dispute a run-time SBoM will most probably weigh heavier as evidence in court than a build-time SBoM. This seems to indicate that in certain contexts we will have to handle not only “as built SBoM”, but also “as installed SBoM” and even “as operated SBoM”.

Use case category 3e: “information sharing”

Motivation:

In general, there seems to be no limit to what characteristics or properties we could be interested in knowing something about for our objects. It could be the release notes for an application, whether or not an object conforms to certain standards, what is the object’s status in the development process and much more. Often the data or information will be created by one category of people and later used by another category of people. In the following we will treat a couple of properties in more detail.

In a hierarchically structured SBoM (aka SBoM of SBoMs) there will be information about what other objects are contained in this object and therefore used by this object. This is the dependency information that development environments can provide developers. However, the dependency information that is very useful when making changes to an object (or more precisely the object's source code) is where this object is used. This information could be obtained by searching all SBoMs for the presence of the object in question's unique ID in the uses attribute. This way changes can be propagated and their consequences analysed.

A more complex case could be a logon module that is used across several of our products and versions and where we discover an issue with version 2.1 of the module - now where exactly is that version used. If we are using the logon module in a binary format, we can scan all products and all versions of a product to look for where the unique ID for the binary is used. However, if we use the source code of the logon module, we will not be able to use a unique ID for a binary object. Every time the source code is built, the binary will get a new unique ID even if it is not built in a different way and the two binaries will be identical. This will make a simple scan for one single unique ID impossible. Instead, the scan in this case will have to be for the unique ID for the source code in version 2.1, which will be present as the source code attribute in the SBoM for each binary object.

In general, we do not want to bake vulnerability information into the SBoM. What is considered safe today may not be so tomorrow, so vulnerability status should be the result of comparing the object's ID with a list of known vulnerable IDs. However, in some cases we want to leave some vulnerability information in the SBoM. If we know that a certain library has a vulnerability, but we have analysed our use of the library and established that our use is not affected by the vulnerability, then we want to document that in the SBoM data for others to use. So, vulnerability scan is a little more complex in real life than just "looking for log4j".

When developers or architects are looking for libraries or code to use for their project, they can also take advantage of the information shared by the entries in the SBoM. From the interviews it came out that there could be a huge difference in the number of direct and in-direct dependencies you would get from using a library. There are also differences in how often and how quickly different libraries are maintained and patched. Likewise, information about End-of-Life can be useful when browsing for libraries or components to use. If such information is shared in the SBoM for a library or component it will make finding the information much easier as there is one single place to look for it.

4. General considerations when implementing SBoMs

During the interviews and studies of the literature, other aspects of the SBoM were found that were generic for all the use cases. Once an organisation decides to implement SBoM, there are things to consider in general. The ambition with this chapter is not to provide solutions for every aspect of the SBoM. Instead, this chapter should be used as a guide for "things" to consider when implementing an SBoM.

SBoM organisation
An SBoM can be a flat list of contained objects or a composite SBoM of SBoMs represented as a hierarchically structured list of contained objects. The list can be manifested in many ways from a text file (make, XML, JSON) to entries in a database.

Level of detail in the SBoM
An SBoM can contain a lot of information that needs to be captured and maintained. The level of detail, or level of abstraction, required in the SBoM needs to be decided upon for each specific use case. Depending on the level of detail needed for the SBoM, the structure of the SBoM can be flat or structured.

In the simplest case, the SBoM lists the "top level" of all included components and libraries. This could be the case if we trust the objects that we include in the SBoM, for example objects delivered to us by an external supplier. If there are dependencies between objects in a flat SBoM, it is not always clear and the information dependencies may need to be explicitly stated as an attribute for an object.

A structured SBoM lists the top level of the included libraries and components as well as the SBoMs that build up the libraries and components. In a structured SBoM, dependencies between objects can become clearer, if an object changes on a low level, it could affect objects on a higher level in the structure.

A flat or structured SBoM also affects the queries we do on the SBoM. For a structured SBoM, the query can be done on different levels while a query on a flat SBoM is limited to one level.

What data is needed
When deciding on the level of detail of the information in the SBoM there are a few things to consider:

- Are there any regulatory requirements that dictates the level of details in the SBoM
- Any third-party software (libraries, APIs etc.) that need to be controlled
- Market requirements, what kind of information is relevant for the customers
- Dependencies to hardware and software.
- Are there objects that we can "trust". Complete information is perhaps not necessary for all objects in an SBoM, for example cryptographic libraries provided by a third-party developer that are bundled together with the release of our system.

The requirements on the SBoM changes over time so finding the right level of information is an evolving process.

Example:
The system developed could be bundled with a distribution of Linux. The sheer amount of packages and dependencies between packages could be too much to track and that amount of detailed information may not be necessary or useful.

Washing, hiding or filtering the data
If an external customer (other company, authorities, …) requests to get a copy of the SBoM, sometimes the information needs to be "washed". Perhaps not all information is relevant for the

external customer (rename of variable from x to y), or the information can reveal sensitive information about other customers (phone operator X have this feature implemented). Some data may be sensitive from a security perspective and may not be released to customers.

Hiding implementation details for internal customers (developers, users, …) could for example be a way to ensure the usage of the correct version of an API.

Representation format
The way that data is represented depends on the usage of the SBoM. If the SBoM is used internally only, for example during software builds, the representation format can be anything that serves the purpose. An SBoM can be in the format of a file (makefile, XML file…) or a query in a database.

If the SBoM will be used externally, as a part of a release to customers or used for certification with authorities, the SBoM may need to be in a format dictated by the receivers.

There is work ongoing to standardise the format of SBoMs, some examples are CycloneDX, SWID and SPDX. These formats are designed to be generated automatically during development and they are in a format that can be read and used both by humans and machines (e.g. JSON, XML).

Availability of the SBoM
The SBoM needs to be readily available for its users. The requirements for availability varies during the lifecycle of a product. In order for the information in the SBoM to be useful, it must also be generated and stored in such a way that the information can be trusted. The implementation of the storage solution is not a core SCM practice, but the software configuration manager should put requirements on the storage solution.

Developers need an updated SBoM several times per day to make sure that they base their development on the latest software. The SBoM should be available for usage immediately when it is produced. The representation format of these kinds of SBoMs need to be readable and usable by both machines and humans. The period of interest of these kinds of SBoMs is usually very short, when there are newly built objects, the “latest and greatest” SBoM will be used.

Once a product has been released to customers, the SBoM needs to be stored for a longer period of time, especially if there are regulatory requirements on the SBoM. The SBoM could be a document that is stored in a document handling system or the SBoM can be retrieved through queries to a database. The creation of the SBoM released to customers could be a (manual) part of the release process.

Automation - both in creation and using
During the interviews the common view of generation of the SBoM should be automated as much as possible. The information in the SBoM should also be machine readable to enable automation in using the SBoM on the consumer side.

Automation is necessary if the SBoM is updated several times per day in a Continuous Integration/Continuous Deployment pipeline while a release that happens once every year could do so with a manually generated SBoM.

Automation ensures that the SBoM is generated fast and that the data that is needed for the SBoM is captured at the same time as the data is generated. Automation also generates the SBoM in a consistent way, so it is possible to verify the correctness of the SBoM.

Dynamic and static status of data in the SBoM
The data that is a part of an SBoM is static, immutable. There are no “free floating” versions of objects or libraries.

However, the *status* of the data in an SBoM can change over time. The data in the SBoM is static but when we query the SBoM the result that we get back can have different meaning due to external circumstances. This could be, for example, a scan to check if there are any known vulnerabilities in the SBoM. When the SBoM is released, the query returns NO and that means no known vulnerabilities are in this SBoM. Running the same query against the same SBoM at a later time could return YES, meaning that now a vulnerability has been discovered in the SBoM.

The SBoM itself does not contain the status of the external references. Instead, the reference that is used to check the status of the SBoM is stored in other systems. The result of a query on the SBoM is then compared to the information in the external system. The SBoM should contain information about which external reference it should be compared to (a link, an ID or other information).

For many parts of the SBoM, the status of the data is static. Examples of data where the status is static can be compiler switches (feature on/off).

For other parts of the SBoM, the status of the data can change over time, the status is dynamic. An example: at some point in time, a library is considered secure and it can be included in the software. At a later point in time, a bug is found in the library and the library is considered insecure to use. A new release and SBoM containing a new version of the library with a fix for the bug is needed to make the software secure again.

Other examples of dynamic status of the data are export legislations, in the event of war sensitive cryptographic algorithms may not be shared with countries that were on the “whitelist” before the war started.

Other legislations can affect the status of the data in the SBoM for example in the automotive and medical industries.

Keeping the SBoM up to date
How to keep the SBoM up to date is different depending on where (or when) you are in the product’s lifecycle.

Build time
When you develop the software, the build system produces and consumes the SBoMs by keeping track of what has been built and how it was built. The build system checks the integrity of the SBoM that you are currently using. If there are differences from previous builds, your SBoM may be updated. During development, the developer can choose the strategy to pick an SBoM to base his/her development on. One strategy can be to let the build system always update the workspace to point at the latest SBoM as soon as there is a new SBoM available. Another strategy is to manually point out an SBoM to base the development on and update the workspace in a controlled way.

Install time
During the installation of a software, for example an operating system, the install program can guide you through the installation procedure. The installation program uses an SBoM from the supplier of the operating system that has been released to the customers. The install program detects the hardware that is installed and configures the software accordingly. The user may also need to answer questions about time zones, language settings etc to get the software configured properly.

The resulting configuration of the SBoM may be unique to the computer it has been installed on and the specific combination of objects and configuration parameters may never have been tested.

Note that the installer itself is not an SBoM, but that when the installer installs a new version of the object, it also has to “install” its SBoM.

Run time
The factory, or production, SBoM released to the customer will many times not be used by the customer, but an updated and customised SBoM must be used instead. Below are some examples of keeping the SBoM up to date.

During the first boot up of a device, once the device is connected to the internet, the device downloads and instals an updated SW, a new SBoM.

In the case of an operating system, the SBoM of the operating system is updated regularly with updates and bug fixes.

A product can be further configured by the customer. The customer installs different applications on his or her mobile phone or computer. The application has its own SBoM

When updating a program or the operating system on your computer the SBoM is not active until the new SBoM has been applied to your system.

SBoM for tools and environments
During our interviews, the topic of having an SBoM for the development environment came up. It is important to know what tools exist in the development environment and also what versions of the tools. Regardless of if the development environments are a server farm “on premises” or if the servers are virtual machines running in a cloud solution, the setup of the environments need to be controlled. An SBoM for the development environment could give the organisation:

- Possibility to streamline the toolchain and processes.
- Mechanisms to identify mismatches between tools, for example different versions of compilers that produce different (incompatible) output. This means that we need to be able to store the configuration settings for the environment/infrastructure in a controlled way.
- Possibility to discover if there has been any tampering of the toolchain, for example a compiler that generates executables with vulnerabilities.

The SBoM for the development environment provides information about the current state of the tools and processes used. When updates (installing new tools or maintenance of the present toolchain) are done to the development environment, a new SBoM is created. If there are problems with the update, the update can be reverted to the previous “good” SBoM.

In order to achieve an efficient use of SBoMs for development environments, both processes and the update of the tools should be automated as much as possible. An “Everything-as-Code” approach to manage the SBoM for development environments, including Infrastructure-as-Code and Configuration-as-Code, will make automatic generation of an SBoM for the environment/infrastructure possible.

Development environments are sometimes managed by a third-party company. This company may or may not respect the development organisation with regards to keeping the development environment in a known state (known SBoM). The maintainer of the development environment may update the development environment without a heads up to the developers. The consequences for this could be that tools stop working, access rights for both tools and developers are removed etc. One way of handling this could be to have a “staging environment” where the update can be tested before it is rolled out.

SBoM and microservices
Microservices are developed and deployed as separate, stand-alone units that build up a whole application or system. The services are deployed using containers and deployment can be done to one or more clusters. Since the services can be deployed to different clusters, the dependencies between them are dynamic, run-time and not static, build-time.

Tools used for creation and deployment of containers handle some parts of the SBoM, the Bill of Material and to some extent the Bill of Process. There is still a need to keep track of the Bill of Information, for example what licences are used “within the container”. The SBoM of a container is an SBoM of other SBoMs. The container can contain one or more objects, and each object also has an SBoM. Each time an object is updated (new SBoM) within a container the SBoM of the container is also updated.

There is also a need to keep track of the actual deployment scripts and tools so an “everything-as-code” would be a good approach. To manage the SBoMs for microservices, automation is key since the SBoMs are updated frequently.

Many tools that manage clusters of microservices, such as Kubernetes, have functionality to query what services are running in the cluster. There are also ways to roll back an update if the update causes problems. The SBoM that can be extracted from these tools are on a “high level” from an SBoM perspective where you can read out what services and what versions of the services that are running. The extracted information could then be compared to the SBoM of the container or cluster to identify discrepancies between what has been deployed versus what is running.

The dynamic nature of applications implemented as microservices puts new requirements on the SBoM. The application can be deployed in multiple clusters that could be spread over different locations. The dependencies between microservices are dynamic, run-time dependencies that change often.

Unique identifiers
Each object of an SBoM needs to have a unique ID to ensure that the correct object is used. It is not enough with a unique name and a version for the object since names of objects can be used by different parts of an SBoM (e.g makefiles). Names and versions of an object do not provide enough information about the object since the content of the object can be tampered with without changing the name or version. Another, globally unique, identifier based on the content of the object is needed.

One way of mitigating the risk of tampered objects is to use checksums of the files that are included in the SBoM. Once the SBoM is “frozen” a whitelist of the checksums can be created. The checksum is globally unique even if objects have the same names and versions throughout the SBoM.

There can be cases where a globally unique identifier already exists for an object, but we decide to rebuild the source ourselves. When using open source, there are often built executables of the software that are ready to use. This software already has a globally unique identifier that can be used for e.g. vulnerability scans. If we as a company decide to build the exact same software ourselves, the build result may not get the exact same identifier (based on compiler version, compiler switches etc). Then there is a possibility that the software that was built from code will not be recognized during a vulnerability scan, even if it is the same as the binary we could download.

5. Conclusions

In this white paper, we have documented that the concept of SBoM has a long history in SCM and can be used for much more than making a vulnerability scan on a list of ingredients. The NTIA has done a great job for this particular use case and in particular for establishing a format for SBoM such that they can be exchanged and shared with external stakeholders.

There is, however, a lot more to the story. There is a strong *Materials aspect* to an SBoM since it defines *what is in the object* or product. This data can be used to answer questions like "is a certain object part of this SBoM" more broadly and not just for the purpose of discovering vulnerabilities. There is also an important *Process aspect* to an SBoM since we are interested in *how the object was manufactured*. This data can, among other things, be used to see if there is an object to "buy" or if we have to build it - or to start troubleshooting at a higher level than source code. Finally, there is a rich *Information aspect* to an SBoM since there are many interesting *questions to ask about an object* and to answer those questions, we need data.

For each of the three aspects there are a number of different overarching use case categories which in turn each has a number of specific use cases. In chapter 3, we identified and described ten different use case categories and sketched ideas for requirements for their implementation. This can be thought of as a "catalogue" of use case categories (and specific use cases) that can be used for inspiration to get started with. The use cases can be applied during different phases of the lifecycle for a product, but it might not be all use cases that are applicable to all types of products or all contexts for software development.

Whether you want to see it as one single SBoM that has different aspects or you want to consider it to be three different types of SBoM (SBoM, SBoP and SBoI) will probably depend on your context and your needs.

In chapter 4, we provided some general considerations that have to be taken into account when working with SBoM no matter which use case category it belongs to. These considerations can be used as a checklist for what you need to address and think harder about before you start implementing SBoM in your organisation. For most of the considerations there is no single right thing to do. They will have to be adapted to your specific use case and your specific context. For some of the considerations you might even decide that they do not relate to your situation.

One consideration is the *availability of the SBoM*. It is important that the SBoM is available and accessible and optimally the SBoM should be ready to use the moment the object is created. Depending on where in the development process the SBoM is used, new objects and SBoMs can be created and ready to use several times per day while the SBoM used for production in a factory is created a few times per year. Regardless of where in the process the SBoM is used, automation of the creation of the SBoM is key. Automation will provide speed and correctness in the creation of the SBoM.

For the Process aspects it is important to also consider *environments, infrastructure and tools*. In a troubleshooting situation this is information that can be very helpful to have. If we have to rebuild old objects or products, then everything essential that has been part of the original process has to be recorded and stored. In some cases, we might use an "Infrastructure-as-

Code" approach to environments and infrastructure or we might even consider "Everything-as-Code". In such cases we should consider applying the SBoM concept also for our environments and infrastructure.

For all data in an SBoM we will have to consider that the *status of data can be static or dynamic*. The data in the SBoM is static and an SBoM should be as immutable as the object it is connected to. However, external circumstances can change the status or the result of using the data in the SBoM in a query. A product covered by SBoM "A" is considered safe. At a later point in time a critical bug is discovered in one of the objects in the product - the object covered by SBoM "A" is not considered safe anymore. The data in SBoM "A" (that is the contents) is static but the result of using that data and comparing it to an external list, changed the result of the comparison due to external events.

Even if SBoM is a long established and well-proven concept in SCM there is still "perfective" work to be done. In some cases because there are areas that have not been thoroughly explored yet, in other cases because new technologies keep coming up. So, we do not consider the concept and definition of an SBoM to be static and carved in stone, but to be something that will be adapted to different and changing situations. Runtime dependencies, that is things that a software object uses, but that are not part of the object are not covered by the SBoM by the present interpretation. In hardware, the gearbox is interfaced to the engine and to the wheels, but neither will be included in the gearbox BoM. In software we might opt to handle it differently if runtime dependencies are crucial to us. So far, the type of SBoM that we have implicitly been talking about could be termed the "as built" SBoM. Equivalently to hardware we could also have an "as designed" type of SBoM, which would very much be like a baseline for the source code. In addition, there might be situations where it could be useful to talk about an "as installed" or even an "as operated" SBoM. The use of containers for software is a relatively new thing, in particular in relation to the use of SBoM. This means that there is no "historic" information about how SBoM can be used in a container context. The interviews with practitioners revealed some experience, but it was preliminary and, in some cases, slightly contradicting. It will be interesting to follow the exploration and investigation to see how the use of SBoM will gradually settle also for containers.

What then if we have a "mixed" product consisting of both hardware and software? The BoM for the engine in a car is just an entry (BoM) in the BoM for the whole car - an SBoM is no different in nature. So, if there is a software product that controls the engine, the SBoM for that will appear as a part in the BoM for the engine - and the SBoM for the infotainment system will appear as a part in the BoM for the car.

In this white paper we have mostly talked about objects and their associated SBoMs. The objects are the artefacts of our primary interest, they are important, and we want to have as much useful information about them (the SBoM) as possible to be able to answer questions. So, what we *should* be talking about is Configuration Items and their associated data stored in a CMDB where Configuration Status Accounting queries can provide valuable answers to questions. With this view of the concept of Software Bill of Materials it should be clear that it has been brought firmly back to the Software Configuration Management domain.

We hope that the results from this white paper will help software configuration managers to revive the concept of SBoM and implement it in ways that will be useful both for customers and for the development organisation.

6. References

[NTIA]: https://www.ntia.gov/category/cybersecurity

[Watts, 2000]: Frank B. Watts: *Engineering Documentation Control Handbook - Configuration Management in Industry*, 2nd edition, 2000.

[Harrop,2021]: Mark Harrop: *The Bill of Process (BOP)*, blog, 2021.

[USDC, 2021]: *The Minimum Elements For a Software Bill of Materials (SBoM)*, https://www.ntia.gov/files/ntia/publications/sbom_minimum_elements_report.pdf

[IEEE828-2012]: *IEEE Standard for Configuration Management in Systems and Software Engineering*, 2012.

[Bee, 2014]: Electric Bee: *Why You Need a Bill of Materials*, blog, March 2014.

[Dart, 2000]: Susan Dart: *Configuration Management - The Missing Link in Web Engineering*, Artech House, 2000.

[Cabrera-et-al-1999]: Ralph Cabrera, Brad Appleton, Stephen Berczuk: *Software Reconstruction: Patterns for Reproducing Software Builds*, August 1999.

[Atria, 1995]: Atria: *ClearCase User's Manual, Unix Edition, Release 2.0.2 and later*, 1995.

[Leblang, 1994]: David B. Leblang: *The CM Challenge: Configuration Management that Works*, John Wiley & Sons, 1994.

[Babich, 1986]: Wayne A. Babich: *Software Configuration Management - Coordination for Team Productivity*, Addison-Wesley Publishing Company, 1986.

[Clemm, 1986]: Geoffrey M. Clemm: *The ODIN System: An Object Manager for Extensible Software Environments*, PhD dissertation, February 1986.

[Feldman, 1979]: Stuart I. Feldman: *Make — a program for maintaining computer programs*, April 1979.