To: Cybersecurity and Infrastructure Security Agency (CISA) 11 December 2023

Re: Software Identification Ecosystem Option Analysis (RFC)
Docket number: CISA–2023–0026

We are pleased to submit our comments on behalf of the Open Source Security Foundation. We have thoroughly reviewed the Cybersecurity and Infrastructure Security Agency (CISA) Request for Comment (RFC) on its Software Identification Ecosystem Analysis white paper and have crafted our comments to effectively address the needs and challenges specified. We believe that our unique skill set and deep understanding of the domain makes us an ideal partner for this endeavor.

Point of Contact (Primary): Omkhar Arasaratnam, OpenSSF General Manager
Website: https://openssf.org/
Email Address: omkhar@linuxfoundation.org
Phone Number: +1 646 321 6202

DocuSigned by:
Arun Gupta
8AAAEFF7FCCA496...
Arun Gupta
Vice President and General Manager for Open Ecosystem, Intel
OpenSSF Governing Board Chair

DocuSigned by:
Omkhar Arasaratnam
39419DF60AD6478...
Omkhar Arasaratnam
OpenSSF General Manager

DocuSigned by:
7059F609FFDE46B...
Brian Fox
Chief Technology Officer, Sonatype
OpenSSF Governing Board Member
Chair, OpenSSF CISA-RFC Committee

DocuSigned by:
David A. Wheeler
2D9352DEF41B456...
David A. Wheeler, PhD
Director of Open Source Supply Chain Security, Linux Foundation
Facilitator, OpenSSF CISA-RFC Committee

OpenSSF Comments to the CISA Request for Comment (RFC) on Software Identification Ecosystem Analysis White Paper

The Open Source Security Foundation (OpenSSF) appreciates the opportunity to provide comments in response to the Cybersecurity and Infrastructure Security Agency (CISA) Request for Comment (RFC) on its Software Identification Ecosystem Analysis White paper

OpenSSF is a cross-industry organization that brings together the industry's most important open source security initiatives and the individuals and companies that support them. The OpenSSF is committed to collaborating and working upstream and with existing communities to advance open source software (OSS) security for all. We have over 100 members, including some of the largest tech companies, financial services, and other organizations involved in developing OSS. Below are our responses. We welcome the opportunity to continue the conversation.

We agree that having a global approach to identifying software is an important problem to address. As these comments show, the issues are complex. We hope our recommendations will help the white paper and the world to move forward toward solving these problems.

Evolve the problem statement, underlying requirements, and use cases

This is an important and difficult issue to address. To make progress on this issue, it is critical to more clearly identify the use cases with specificity.

Along with use cases, there are other key considerations in addition to those called out in the paper's section 1.3. Consumers need software to automate important tasks. However, few if any

directly need a software identifier, so saying there are “requirements” on software identifiers directly is missing a critical step.

Software identifiers are important not because they are *directly* useful, but because there are *other indirect* tasks that are greatly enabled by identifiers. By clearly laying out the key use cases for software identifiers, the path to identifying the tasks enabled by software identifiers helps more clearly answer the approaches to best answer the question “how should the world identify software?” For example, a potential use case for software identifiers is hinted at in this paper’s section 1.3 by phrases like “vulnerability scanner”. However, a vulnerability scanner tool in and of itself is not a use case. By defining a list of key use cases up front, we think this paper could make even more progress toward solutions to this hard problem.

Below are examples of use cases we believe are important when considering how to structure Software Identify Schemes (this is not a complete list):

- Be able to use a software composition analysis (SCA) tool (in this document called an “inventory tool”) on a particular program (including a package or container) and report the software identity of every one of its components (transitively).
    - Then, given a set of software identities, determine what known vulnerabilities are in those components, as a way to estimate risk of use.
    - Similarly, given a set of software identities, determine the licenses, as a way to identify potential licensing issues.
- Given a set of files on a computer, which of those files are known software and what are their sources? This is a common digital forensics task (e.g., for law enforcement), often implemented by supporting databases like the National Software Reference Library (NSRL), which fundamentally uses cryptographic hashes for identification of software files (specifically MD5, SHA-1, and SHA-256).
- On a network, be able to identify the software installed on systems, and identify them at a larger level to determine license compliance (as part of a license compliance audit), if they’re up-to-date, and if there are known vulnerabilities in those larger components.
- Given a vulnerability report to a public database (e.g., CVE or similar), be able to report which software (by software ID) that has this vulnerability. Note that this will typically be a set with version range(s). This allows analysts, researchers, and others to estimate the likely impact of a vulnerability without knowing about specific organizations.
- Given a well-known vulnerability (e.g., by CVE id), be able to identify all software in an organization that contains that vulnerability (transitively) and thus is at higher risk of exploit.
- Be able to detect when version ids are reused (e.g. determine that “version 1.0 this week is different from version 1.0 from last week”). Unfortunately, it’s relatively easy to release new versions of source code and/or compiled results (including packages and containers) while reusing a version identifier. In many systems this is the default behavior, as a developer has to remember to set a new version identifier to use it. Attackers, of course, may want to release a malicious version of a “believed-good” version. Those acquiring the software thus can sometimes end up with unexpected

> “new” software with a previously-acceptable version identifier, unless their process prevents it.

Additionally in section 2.2.1, other “requirements” are discussed:

1. The same identifier must not be assigned to different software (“identifier reuse”)
2. Multiple software identifiers are never created for a single piece of software (“overidentification”)

These aren’t the same kind of requirement as the use cases above. But these are merely derived “requirements”; they are not the real requirements. What really matters are the mostly-unstated use cases, and ensure that key tasks can be done well and efficiently.

In fact, there is a need in some use cases for a single identifier to refer to a set of software within modern software development (e.g., “I’m paying for foo 1.0.X, any X will do”) or “install the current version of bar” or “install the current version of bar in its 1.0 series”. The reality is that software already has multiple software identifiers. In many cases you do want to ensure that a given software identifier only refers to one specific software instance, and typically in those cases what’s important is #1 (avoid identifier reuse) and not #2 (avoid overidentification).

We’re unlikely to get to a world where there is one identifier for any one software instance; in most cases we will want #1 and we’re willing to drop #2. By insisting, without evidence, that there must always “only be one” the paper almost ensures a solution can’t be found.

Needed: A recommended working solution

This paper lists various options, but we believe the paper should provide a specific proposal to solve the problem (once the requirements are more clearly defined as discussed earlier).

We don’t think there will be a single software identifier that will be used in all cases, across all technologies, projects, development languages, and ecosystems. Instead, the world needs to know when it’s preferable to use which system to help solve various real-world use cases. Ideally, software identifiers would solve these problems, or at least identify the contour of the solution. To the extent practical they need to be consistent, scalable, reproducible and international.

Importance of a consistent ID

For the SCA & SBOM use cases discussed above, it becomes important that given a piece of software, a person or system can walk backwards to determine a common name. Those names need to be human-understandable and align to other data (including, but *not* limited to, vulnerability databases).

An analogy can be made with identifying chemical compounds. The International Union of Pure and Applied Chemistry (IUPAC) defines a widely-used nomenclature for chemical compounds. While we typically use the term “Water” to refer to the chemical H2O, the bottoms-up name that can be derived from the molecular observation is Dihydrogen Oxide. It is true for any compound, that you can follow a convention to work backwards from a chemical formula to a name that can be commonly understood.

In practice, we expect multiple systems to be in use simultaneously for some given software, for example between open source software and closed source software, the conventions might be slightly different. Here, chemistry provides a similar analogy in that they have different naming systems for organic vs. inorganic compounds.

Scaling is a challenge with a single naming authority

As noted in the white paper, there are substantial challenges when an identification ecosystem is managed by a central authority. An example of such a system is CPE (Common Platform Enumerator) used for CVEs, which is now having many struggles due to scale. Having a small number of organizations responsible for creating identifiers and maintaining identifier databases creates bottlenecks in a space that is moving and evolving very quickly. Even systems such as CVE that try to manage federation with a centralized model have proven to struggle within the bounds of the current software environment.

Given how vast the software ecosystem is across all dimensions of open and closed source, it is impossible to imagine a centralized authority being effective. Many domain identity assignments such as DNS have had to become hierarchical and distributed to scale. Every modern company is a software producer and thus the scale of the problem is very similar to DNS (Domain Name System.

Whatever guidance is produced, it needs to focus on an identification system that is open and distributed. Just as open source has consumed software development, the open source model, including distributed and openness as well as international cooperation, should be a central theme to any identification system.

Leveraging DNS as a global namespace for software

It may be that a combination of more carefully-defined convention, combined with some social pressure could move society in a better direction. The only naming system that has scaled to cover the global technology space is DNS, it is essentially *the* namespace for modern life. Through DNS, it is possible to find essentially every software producer’s corporate website quickly. This provides opportunities to leverage this pre-existing namespace as the basis for a

consistent way to identify software. The DNS system is also inherently internationally supported and geographically distributed, which are key requirements for any identification scheme.

Maven Central and Java have long used the reverse of the fully qualified domain name (FQDN) for naming classes and java packages. This provides a mechanism to validate ownership/control of a domain for authentication purposes (e.g. to publish) which has to date been very effective at avoiding many typosquatting malware problems observed in other ecosystems. The Maven developers encourage repositories to internally use DNS (specifically reverse-DNS) for full names, to counter many typosquatting and dependency confusion attacks.

This predictable scheme also allows a consumer to understand where they might begin to find a given package (eg org.apache / org.eclipse / org.openssf for corresponding software), or importantly how to construct an identifier for a package if it wasn't already known.

A known weakness in leveraging DNS as the basis of software identity is that companies change names, rebrand products, merge and/or close down and their domains follow along. Thankfully this is a well-understood problem and there are widely-known solutions that are already often applied today.

Since domains are so critical in the modern era, it's fairly common for companies to hold on to older domains and ensure they are redirected to the latest correct name. As an example, sun.com still redirects to Oracle, many years later after their acquisition. This reduces the likelihood that someone could come along and hijack the name and for example use it to publish fake components.

It is also possible to monitor when DNS changes and repositories can use this to ensure that should a domain becomes abandoned, and pre-existing validation for those names becomes invalidated. Should a domain be legitimately acquired later, steps can be taken to ensure that the existing subsections of the name space are still protected from future tampering by making them unavailable to the new owner. While DNS has challenges, the problems are known, generally understood, and organizations already have mechanisms on how to deal with them by reserving and keeping domains.

In short, DNS can be helpful when nothing else provides information and this can become a key building block in a convention to arrive at a given piece of software's identity.

Hash-based identifiers: Helpful, but not by themselves

In practice, software is often identified by cryptographic hashes (whether through commit or other digital signing). These are often found for their packages (especially before installation) or of individual executable files (post installation). The git version control system is widely used, and specific versions are typically identified by their commit ID (a cryptographic hash currently

based on SHA-1). These algorithms include MD5 (in spite of its known flaws), SHA-1 (ditto), SHA-256, and SHA-512. OmniBOR is also based on cryptographic hashes. The paper should discuss cryptographic hashes in general as a way to identify software as “inherent identifiers”.

The key advantage of cryptographic hashes is that they can, with great specificity, identify unique software. This provides an easy way to compare two IDs and know that if the IDs are identical that the software is identical.

Cryptographic hashes also have several disadvantages.

1. *The “same” software may have many hashes*. Changes that “don’t matter” can produce different hashes. Packages or executables that are slightly different for each user (e.g., due to an embedded license key) will produce a different cryptographic hash for each user. Executables are sometimes modified on the fly (e.g., by prelinking), causing their hashes to change over time. Developers typically refer to commit hashes instead of hashes of the result. The hash value of the results from “git archive”, such as a software archive release from GitHub for the same software, may occasionally change (GitHub promises to give 6 months’ warning before each change).
2. *They are often overprecise*. In many cases, users want to refer to a set of software, e.g., “any version 1.0.X of software foo”. This requires identifying a set cryptographic hashes, sometimes a large set, to cover simple cases.
3. *They aren’t human-readable nor do they carry semantic information*. It’s not obvious from a cryptographic hash that some software is “foo from organization bar” or what the version number is from the identifier. It’s also harder to determine “is this old?”

Cryptographic hashes are often helpful for rigorously identifying specific packages, but also have many limitations. In addition, while OmniBOR has many potential advantages (especially in providing easy mechanisms for looking up the transitive dependencies of software), it’s still relatively new and not in wide use at this time. So in practice, cryptographic hashes are helpful when supplemented with another system, such as purl, but do not by themselves solve the naming and identification problem.

In support of purl + hashes

Purl is currently in use as a de facto standard in many situations. For example:

- OpenSSF Open Source Vulnerability (OSV) schema is widely used to alert organizations/developers about vulnerable components. They have added support for purl <https://ossf.github.io/osv-schema/>. The format can separately store version lists and version ranges.
- OSSIndex is another popular vulnerability database that is built using purl in the apis..
- Both SPDX and CycloneDX, widely-used SBOM formats, support purl.

Purls are human-readable and they accomplish several things:

- they provide a human-readable name,
- they identify the source of the software,

- they often provide the information necessary to re-download the software.

Note that a software project downloaded from a package repository should be represented by a different purl than a project downloaded directly from GitHub, even if it is the “same” project and version from some viewpoints, since these are different sources and they might not be the same. Today’s software is mostly comprised of open source software (OSS) (on average 80%), and purl works very well today at providing software identifiers for this most common case.

It’s important to understand that purl includes many different “types” that leverage the innate identifiers of various ecosystems. The first part of the id is, in fact, the ecosystem such as Maven, or npm, or PyPI the package is from. In this way, what follows can be interpreted using the conventions of that ecosystem. This structure also provides the ability for purl to be extended to support new, as yet known ecosystems and standards, without requiring an entirely new parsing schema.

Purl also supports additional, optional pieces of metadata called “qualifiers”. For example, a cryptographic hash may be appended to provide extra precision when required. This counters a known weakness with purls that different versions of software can in some cases use the same purl identifier. This most obviously happens when the version identifier isn’t specific, e.g., there may be many “version 1.0” of some software (1.0.0, 1.0.1, 1.0.2, and so on). This can be helpful, as sometimes users simply want to refer to the “version 1.0” series (including software not yet released). However, in other cases it’s undesirable. A more subtle challenge is that software can be re-released with the same version identifier in some situations. Some ecosystems use technical means to prevent multiple releases of the same version, but not all do. Adding a cryptographic hash for added specificity is helpful. That way, you can be certain of exactly what software is meant when there’s a match. When there is not a match, the combination of a purl and cryptographic hash provides some information (as opposed to having only a mysterious hash) along with an indicator of a potential problem. An example of this precision might look like this:

pkg:maven/org.apache.xmlgraphics/batik-anim@1.9.1
?checksum=sha1:ad9503c3e994a4f,sha256:41bf9088b3a1e6c1ef1d

Purl can easily refer to packages available through non-default repositories. Almost all purl types have a default repository, however, many types support the qualifiers “repository_url” (an extra URL for an alternative, non-default package repository or registry), “download_url” (direct download), and/or “vcs_url” (for a package version control system URL).

Many users of purl expect that the software being identified is part of a repository they can query to retrieve a compiled package (e.g., by using a package manager). However, nothing in the purl specification requires this. Software can be identified using their source code repository if it’s a common one such as GitHub or GitLab. More generally, purl includes a “generic” type, which allows an arbitrary URL (including a DNS name). This allows a software source (including a supplier) to simply assign a name without needing permission. One problem with this is that there are relatively few conventions documented at this time for this. It would be helpful if

additional conventions were defined to help the rest of the world guess ahead-of-time what the identifier for some software would likely be (given its name, version, and source).

Some purl users incorrectly expect that all software identified by a purl will be freely available to all. Again, this isn't required. An organization could use a DNS name it controls (such as using a reverse DNS name or using its DNS name in its repository_url). This would allow purls to easily identify arbitrary closed source software, using a single overall specification. Again, the problem is the lack of widely agreed-on conventions for this use case.

The combination of cryptographic hashes and a purl solve many problems. The idea is that to identify software, hashes and a purl could be used. Queries could first look for cryptographic hashes, and when that fails to match, use purl as it is a more generalized and human-readable identifier.

As noted earlier, DNS has many advantages as a basis for naming. Purl automatically takes advantage of DNS in two distinct ways:

1. *When (reverse) DNS names are used as part of a naming choice (or coordinate), purl automatically uses them*. As noted earlier, systems that use DNS-based names (typically as reverse-DNS names) provide substantial defenses against typosquatting and domain confusion attacks. These are common attacks, so automatically countering them is valuable. When such names are in use, purl automatically takes advantage of them, improving security for everyone.
2. *URLs embed DNS*. Most purl types have a default repository URL. If a user wants to use a different repository, the user must specify a different URL. In both cases the URL includes a DNS name. Again, this means that purl builds on DNS to provide a clear naming system that works internationally.

Purl is not a perfect solution; it has known weaknesses. We believe the best step would be to identify the most important weaknesses of purl, and then expand purl to fix them. The purl project is open source, and is willing to accept patches as well as public discussions about weaknesses.

Must be internationally supported

The document needs to emphasize that software development and distribution is a global issue, and therefore, solutions require international cooperation and collaboration. A solution adopted by the US that conflicts with international norms would be a great detriment.

The paper on page 13 does mention issues "if the authority is affiliated with specific national, government, or commercial entities who might be viewed unfavorably in some circles" and it uses the term "global" in a technical sense. However, that's not enough. It should instead clearly emphasize the *need* for international discussion and cooperation. Governments and industry around the world are all challenged by these problems, and we believe there *is* an interest in solving these problems. It's wonderful that the US government is showing leadership by

spending effort to analyze these challenges and working to solve them. However, it's important to ensure that the solutions are not just a dictate from any single government, but instead are part of a collaborative effort to analyze and solve these important problems.

The need for international cooperation is another reason to use solutions that don't depend on a single central name assignment system. Cryptographic hashes, DNS (including reverse DNS), and purl can all be easily supported internationally. Central assignment mechanisms, like CPE, struggle with scale and don't work as well internationally.

Conclusion

It is unlikely that there will be a single software identifier that will be used in all cases. Rather than striving for this aspirational goal, conventions need to be defined and widely shared. It will then be obvious when it's preferable to use what system to help solve various real-world use cases. In some cases, combinations will be necessary. Ideally, software identifiers would solve these problems, or at least identify the contour of the solution. To the extent practical they need to be consistent, scalable, reproducible and global.