US Patent & Trademark Office
Patent Public Search | Text View

CYBER ATTRIBUTION OF SOFTWARE CONTAINERS

Abstract

A system and method for software containers attribution are provided. The method includes de-compiling a configuration file of a container image of a software container; identifying at least one candidate build file in the de-compiled configuration file, wherein the at least one candidate build potentially formed the container image; determining if at least one matching condition is satisfied between each of the at least one candidate build file and the de-compiled configuration file; associating the configuration file with each candidate build file satisfying the matching condition; and updating an inventory with the associated configuration file and the respective candidate build, wherein such association provides a direct mapping between the container image and the build files formed the container image.

Inventors: YARON; Eshel (Amsterdam, NL), SCHWARTZ; Tomer (Tel Aviv, IL), RESNIANSKI; Pavel (Tel Aviv, IL)

Applicant: Wiz, Inc. (New York, NY)

Family ID: 88194391

Assignee: Wiz, Inc. (New York, NY)

Appl. No.: 19/193142

Filed: April 29, 2025

Related U.S. Application Data

Publication Classification

Background/Summary

CROSS-REFERENCE TO RELATED APPLICATIONS [0001] This application is a continuation of U.S. patent application Ser. No. 18/917,410 filed on Oct. 16, 2024, now pending, which is a continuation of U.S. patent application Ser. No. 17/656,914 filed on Mar. 29, 2022, now U.S. Pat. No. 12,204,651. [0002] The contents of the above-referenced applications are hereby incorporated by reference.

TECHNICAL FIELD
[0003] The present disclosure relates generally to attribution of software containers in virtualized execution environments.
BACKGROUND
[0004] Cyber (or security) attribution is the process by which security analysts collect evidence, build timelines, and try to piece together evidence in the wake of a cyber-attack, to determine what caused the breach. That is, understanding what and who caused the breach can be a valuable source of intelligence for cyber-breach victims. For example, attribution of a detected malware can identify the type of resource utilized to run the malware (e.g., a software agent running a Linux(R) agent), the network resources that the malware communicated with, local resources that have been exploited, and so on. The attribution may not necessarily lead to a hacker who maliciously exploited the vulnerability but also to a programmer who accidentally caused the vulnerability.
[0005] Attribution as performed today requires security researchers (human) to investigate at least the source of the vulnerability. As such, cyber attribution usually requires significant amounts of time and resources. Furthermore, in many cases, the process involves a significant amount of educated guesswork, meaning results are not deterministic and can often be subject to debate.
[0006] Attribution of software containers is a very complex problem due, in part, to the structure of containers and how they are formed. A software container, such as build by Docker®, is a standard unit of software that packages code and all its dependencies to allow applications to run from one computing environment to another. A software container includes a container image which is a lightweight, standalone, executable package of software that includes all resources to run an application including code, runtime, system tools, system libraries and settings. The build file (e.g., Dockerfile) contains all the commands to assemble and create a container image. Using a build file, users can create an automated build that executes several command-line instructions in succession.
[0007] Attribution of software containers would allow for determination of the root cause of vulnerabilities detected in containers. Current techniques disclosed in the related art allows for identification of vulnerabilities in software containers either as a static container image or as an executable instance. However, current container security techniques cannot identify the root cause of detected vulnerabilities. This is a limitation as the source of the vulnerabilities cannot be corrected quickly without having significant down time of the application running a vulnerable container. Further, identifying the root cause of the vulnerabilities can determine who (e.g., a programmer) caused any detected vulnerability and whether the source of the vulnerability is associated with other container images. That is, attribution of software containers would allow for further investigation of reasons for vulnerabilities and prevent similar vulnerabilities in the future, and to further allow for fixing such vulnerabilities at minimum delay.

[0008] It would therefore be advantageous to provide a solution that would overcome the challenges noted above.

SUMMARY

[0009] A summary of several example embodiments of the disclosure follows. This summary is provided for the convenience of the reader to provide a basic understanding of such embodiments and does not wholly define the breadth of the disclosure. This summary is not an extensive overview of all contemplated embodiments, and is intended to neither identify key or critical elements of all embodiments nor to delineate the scope of any or all aspects. Its sole purpose is to present some concepts of one or more embodiments in a simplified form as a prelude to the more detailed description that is presented later. For convenience, the term "some embodiments" or "certain embodiments" may be used herein to refer to a single embodiment or multiple embodiments of the disclosure.

[0010] Certain embodiments disclosed herein include a method includes de-compiling a configuration file of a container image of a software container in order to obtain a de-compiled configuration file, wherein the de-compiled configuration file includes a first plurality of commands; identifying a candidate build file in the de-compiled configuration file, wherein the candidate build file contains a second plurality of commands; and matching the configuration file to the candidate build file based on the first plurality of commands and the second plurality of commands.

[0011] Certain embodiments disclosed herein also include a non-transitory computer readable medium having stored thereon causing a processing circuitry to execute a process, the process comprising: de-compiling a configuration file of a container image of a software container in order to obtain a de-compiled configuration file, wherein the de-compiled configuration file includes a first plurality of commands; identifying a candidate build file in the de-compiled configuration file, wherein the candidate build file contains a second plurality of commands; and matching the configuration file to the candidate build file based on the first plurality of commands and the second plurality of commands.

[0012] Certain embodiments disclosed herein also include a system software containers attribution, comprising: a processing circuitry; and a memory, the memory containing instructions that, when executed by the processing circuitry, configure the system to: de-compile a configuration file of a container image of a software container in order to obtain a de-compiled configuration file, wherein the de-compiled configuration file includes a first plurality of commands; identify a candidate build file in the de-compiled configuration file, wherein the candidate build file contains a second plurality of commands; and match the configuration file to the candidate build file based on the first plurality of commands and the second plurality of commands.

Description

BRIEF DESCRIPTION OF THE DRAWINGS

[0013] The subject matter disclosed herein is particularly pointed out and distinctly claimed in the claims at the conclusion of the specification. The foregoing and other objects, features, and advantages of the disclosed embodiments will be apparent from the following detailed description taken in conjunction with the accompanying drawings.

[0014] FIG. **1** is a system network diagram utilized to describe various disclosed embodiments.

[0015] FIG. **2** is a flowchart illustrating a method for software containers attribution according to an embodiment.

[0016] FIG. **3** is a flowchart illustrating a method for hierarchical matching according to an embodiment.

[0017] FIG. **4** is an example of diagram of hierarchical representation generating from container

images and build files according to an embodiment.

[0018] FIG. **5** is a schematic diagram of an attribution system configured according to an embodiment.

DETAILED DESCRIPTION

[0019] It is important to note that the embodiments disclosed herein are only examples of the many advantageous uses of the innovative teachings herein. In general, statements made in the specification of the present application do not necessarily limit any of the various claimed embodiments. Moreover, some statements may apply to some inventive features but not to others. In general, unless otherwise indicated, singular elements may be in plural and vice versa with no loss of generality. In the drawings, like numerals refer to like parts through several views.

[0020] It has been a challenge to secure software applications executing software containers due to the way containers are built and executed. It has been identified that providing attribution for software containers would allow for immediate identification of any vulnerability exist or detected in a container, and for immediate correction of such vulnerability.

[0021] To this end, the disclosed embodiments provide techniques for attribution of software containers. Specifically, the disclosed embodiment provides a deterministic mapping between a container image and a build file utilized to create the image. The disclosed embodiments can also provide deterministic mapping between a specific layer in the container image and a code line in the build file from which the layer is created. Therefore, when a vulnerability is detected or reported in a specific container or layer, the build file and code line responsible for creation of the infected layer can be immediately identified, and the vulnerability can be mitigated by altering the identified responsible code and building a non-vulnerable container from the fixed build file. Accordingly, the disclosed embodiments provide techniques for improving security of the software infrastructure.

[0022] Further, it has been identified that, even if users (e.g., security analysts) are theoretically capable of identifying such attribution of the root causes of a vulnerable container image or layer within an image, the manual process for identifying root causes by a user relies on subjective evaluations of connections based on the operator's understanding of the structure of the containers. This could lead to some guesswork, and thus results that are not deterministic. Moreover, in a typical organization, there are thousands of software containers implemented that can be programmed by programmers from inside and/or outside the organization. During a lifetime of a container, the build file can be modified multiple times, thus having multiple versions of the same build files may be associated with the container image. The inconsistency in the development process and registry of containers may lead to manual analysis of potential mappings, thereby leaving vulnerabilities untreated and exposed. Additionally, the results of such manual analysis can vary between individuals, thus causing confusion and results in inconsistent interpretations of reported cybersecurity events and vulnerabilities.

[0023] The various disclosed embodiments include methods and systems for attribution of software containers by analyzing manifest (or "config. File") associated with container images and build files, mapping each container image to a single build file as well as mapping each layer in the container image to a specific code line. The method further includes populating an inventory based on the mappings (attribution) and enriching the inventory with any detected or reported vulnerability in a container. This allows for tracing the build tree back to the exact code line which triggered the vulnerability.

[0024] FIG. **1** shows an example network diagram **100** utilized to describe the various disclosed embodiments. The example network diagram **100** illustrates a cloud computing platform **110** including servers **111** configured to execute applications through an instantiation of one or more software containers. The images of the software containers are stored in a container image repository (or simply a container repository) **120** and their build files are stored in a code repository **130**. The software containers can be built and executed in any containerized environment, such as

Kubernetes®, Docker®, and the like. The servers **111** may operate in a production environment. The cloud computing platform **110** may be a private cloud, a public cloud, and/or a hybrid cloud.

[0025] A software container, such as built by Docker® is a standard unit of software that packages code and all its dependencies to allow applications to run from one computing environment to another. A software container includes a container image which is a lightweight, standalone, executable package of software that includes all resources to run an application including code, runtime, system tools, system libraries and settings. The build file (e.g., Dockerfile) contains all the commands to assemble and create a container image. The build file allows an automated build of container images. A container image includes several layers, where each layer corresponds to certain instructions in the build file. For example, a base layer would be a version of an operating system (e.g., Ubuntu).

[0026] The container repository **120** maintains a collection of container images used to provide different versions of an application. The container repository **120** may be an integral element of development and deployment practices that make use of containers and PaaS environments. DevOps teams make use of software containers, and most teams pull a variety of container images from numerous sources, such as an open source or a source proprietary to the organization. DevOps team may build custom container images, optionally using existing ones as layers, and share their source code or container images. An organization will use a container repository to share container images with its team or with the broader repository platform community. It should be noted that the repository **120** may be a public repository or a private repository that allows an organization to keep its images private within the development team.

[0027] Each container image stored in the container repository **120** contains a manifest file, which is a text-based file (e.g., JSON). The manifest (e.g., Docker manifest) includes information about a container image, such as layers, size, and digest. The docker manifest command also gives users additional information, such as the operating system (OS) and architecture an image was built for. A manifest list (.config file) is a list of image layers that is created by specifying one or more (ideally more than one) image names. The manifest may define metadata designating, for example, build information, license information, or any customized data. The data metadata is set by a programmer and may not be accurate or updated. According to an embodiment, a.config layer (hereinafter "configuration file") is retrieved from the manifest for analysis.

[0028] The code repository **130** includes the build files of the container images in the container repository **120**. The code repository **130** may be a private or public repository. Examples for the code repository **130** includes Git®, Mercurial, Subversion, Perforce, and the like.

[0029] Also, shown in FIG. **1**, is a detector **140** configured to detect vulnerabilities or misconfigurations in software containers executed in the application server **111** or container images stored in the container repository **120**. The detector **140** is a third-party system, and the operation of which is outside the scope of the disclosed embodiments.

[0030] According to the disclosed embodiments, an attribution system **150** is operable to provide attribution of container images stored in the container repository **120** and enrich the determined attributions with vulnerabilities reported by the detector **140**. The attribution system **150** is configured to communicate, access information, or pull data stored in the container repository **120**, the code repository **130**, and the detector **140**.

[0031] In an embodiment, the attribution system **150** is configured to retrieve build files from the code repository **130** and configuration files metadata of container images in the container repository **120**. The retrieved configuration files and build files are of containers executed in servers for software applications of a specific organization. Then, for each configuration file (or a container image), the attribution system **150** is configured to find a corresponding build file that created the image. That is, for example, if a container image "A" was created using a "build_file_1", the attribution system **150** would attempt to map these to two entities.

[0032] The attribution system **150** may be configured to retrieve configuration files and not the

container images so the amount of data being transferred to and processed by the attribution system **150** is limited to the textual files of the container images and not the entire the data of the images. It should be appreciated that when needed, the entire container image can be downloaded from the repository as well.

[0033] The attribution system **150** is configured to analyze each build file and configuration file to provide a deterministic mapping between a container image and a build file.

[0034] However, the information retrieved from the repositories **120** and **130** does not always allow for deterministic mappings. Thus, the attribution system **150** is configured to implement and utilize techniques and heuristics to enable such mappings. Specifically, a number of build files, at the outset, can be potentially mapped to a single container image. Further, a container image can be an outcome of multiple build files and single build file can be programmed to create multiple containers.

[0035] To this end, the attribution system **150** is configured to de-compile the container images and check if one or more matching conditions between the de-compiled image and the build files are met. A de-compiled image is the configuration file retrieved from the manifest of the respective image and is processed to remove any information that cannot contribute to the matching. Such information may include, for example, hash values of the various images in the container image.

[0036] In an embodiment, to determine if a configuration file (de-compiled) matches a build file, all commands from the build file are matched to all commands in the configuration file, excluding, for example, FROM or LABEL commands (hereinafter "excluded commands or command"). To this end, a configuration file may be searched for all commands starting from the last command in the file until the first excluded command is encountered. The first matching attempt may result with a number of candidate build files for a single configuration file. When there is more than one candidate build files, such files are checked against one or more matching conditions to determine which of the candidate files can be associated with a container image. If only one build file is found to match the configuration file, then the mapping is deterministic.

[0037] In an embodiment, the matching conditions include, but are not limited to, similarity; equal files; creation time; number of commands; and hierarchical matching. The various different matching conditions are required as the configuration file, in most cases, does not exactly match the image layer.

[0038] In the similarity matching condition, it is checked if all candidate build files are the same. This may happen when the same build file is saved in different locations (e.g., folders) by the same user or different users.

[0039] In an embodiment, the creation time matching condition includes comparing the image container's creation time to the creation date of a candidate build file, and if the former date is later, then this build file was not the source of the image. For example, if a creation data of "build_file_1" is Jan. 1, 2021, and the creation of the container image "A" is Jan. 1, 2020, then "build_file_1" was not utilized to create the container image "A."

[0040] In another embodiment, the number of commands matching includes comparing the number of commands in the build file to the number of commands (without the excluded commands) in the de-compiled configuration file. If the number of commands are the same, the build file can be associated with the container image. In an embodiment, if the number of commands are not the same, then the build file that includes the maximum number of "matching" commands is determined to be the matching file. For example, there are 2 candidates, "build_file_1" and "build_file_2", to match to a configuration file A' of a container image "A". "Build_file_1" includes 10 commands matching to 10 commands in the configuration file A' and "build_file_2" includes 20 commands matching to 20 commands in the configuration file A'. Thus, the container image "A" is mapped to the "build_file_2."

[0041] In another embodiment, hierarchical matching is performed when a container image is formed from multiple build files. In this case, an attempt to map a single build file would not

provide the proper matching. Thus, in this embodiment, a dependencies tree is created to represent the relationship between build files and a container image. The dependencies tree is created by listing all commands starting from the last command until the first excluded command (or any equivalent command for image creation) is encountered. This is performed for each candidate build file. This list of commands is matched to the configuration file which should all include all the commands in the list. Every candidate build file, having a set of commands mapped to the configuration file, is associated with the respective container image. The hierarchical matching will be discussed in more detail below with reference to FIG. **4**.

[0042] The attribution system **150** is further configured to map a code line in the build file to the respective layer in the associated container image. That is, when a container image is associated with a build file (or files) based on the deterministic matching, the build file is further analyzed to map the code lines that form layers in the container image.

[0043] The attribution system **150** is further configured to populate an inventory of container images and their associated build files. The inventory is enriched with the code lines in each build file that form layers with reported or known vulnerabilities. The inventory may be a data structure, such as a tree structure stored in a memory of the attribution system **150** or any other storage. In an embodiment, the generated inventory can be saved in storage or a database **160**.

[0044] It should be appreciated that the attribution of container images as stored in the inventory would allow for quick remediation and detection of vulnerability (reported by the detector **140**). A user, e.g., a programmer who programmed the build file of a vulnerable container, can be alerted to which build file and/or event in the respective code lines that the vulnerability results from. In a typical organization with thousands of container images in a production environment, this process may be a significant reduction in time and effort in providing a secure environment.

[0045] The repositories **120**, **130**, the detector **140**, the attribution system **150**, and database **160** may be connected to a network **170**. The network **170** may be a local area network (LAN), a wide area network (WAN), the Internet, and the like. Further, the repositories **120**, **130**, the detector **140**, the attribution system **150**, and/or the database **160** may be hosted in a cloud environment such as the cloud computing platform **110** or a different platform. In some configurations, the attribution system **150** can be integrated in the detector **140**.

[0046] It should be understood that the embodiments described herein are not limited to the specific architecture illustrated in FIG. **1**, and other architectures or deployment may be equally used without departing from the scope of the disclosed embodiments.

[0047] FIG. **2** is an example flowchart **200** illustrating a method for software container attribution according to an embodiment.

[0048] At S**210**, build files of container images are retrieved from a code repository. A build file (e.g., a Dockerfile), when executed, causes the creation of a container image of a software image. The build files of the entire development history is retrieved.

[0049] At S**215**, configuration files of container images are retrieved from a container repository. A configuration file includes the information on layers of a container image of a software container. The configuration file, for example, may include a first layer of a container manifest. In an embodiment, the build files and configuration files are of container images of respective software containers executed in a production environment of an organization. Examples for the code and image repositories are provided above. It should be noted that S**210** and S**215** can be executed in parallel or in a reverse order.

[0050] At S**220**, a configuration file from the retrieved configuration files is selected and de-compiled. The de-compilation of a configuration file includes removing all information that cannot be utilized for the matching and association of build files. For example, all hash values are removed from the selected configuration file.

[0051] At S**230**, at least one candidate build file is identified for the selected (and de-compiled) configuration file. A candidate build file may be identified by matching the commands in the build

files to a selected de-compiled configuration file. For example, if a "build_file_1" includes the following commands:

TABLE-US-00001 build_file_1:   FROM scratch   RUN yum install -y httpd   CMD ["echo", "Web Image created"] And a selected configuration file A' includes the following commands: RUN yum install -y httpd   CMD ["echo", "Web Image created"]

Then, the "build_file_1" is a candidate as the commands (RUN, CMD) in both files are the same.

[0052] Multiple candidates can be identified for numerous reasons. For example, if the user (e.g., programmer) saved the same build file in multiple locations (folders), the command and their variables do not exactly match due to compilation changes or when a container image is composed using a number of build files.

[0053] In an embodiment, the identification of candidate build files is performed based on one more or text search techniques. Such techniques may include using regulation expression, string matches, indexing, wild-card searches, and the like. It should be noted that all commands from the build file are matched to all commands in the selected configuration file, excluding the excluded commands. To this end, a configuration file may be searched for all commands starting from the last command in the file until the first excluded command is encountered.

[0054] At S**235**, it is checked if there is only one candidate build file, if so at S**270**, the only matching build file is determined to match the selected de-compiled configuration file, and the two files (build and configuration) are associated with each other. If S**235** results with a No answer (there are multiple candidate build files), execution continues with S**240**. At S**240**, all candidate build files are checked to determine if they meet at least one

[0055] matching condition. As noted above, the matching condition includes, but are not limited to, similarity; creation time; number of commands; and hierarchical matching. The various matching conditions discussed in detail above.

[0056] At S**250**, it is checked if at least one matching condition is satisfied; and if so, execution continues with S**270** where the build file is determined to match the selected (de-compiled) configuration file, and the two files (build and configuration) are associated with each other. Otherwise, at S**255**, an error message is sent that no build file was found for the selected configuration file.

[0057] At S**260**, it is checked if all configuration files have been checked; if so, execution continues with S**280**; otherwise, execution returns to S**220** wherein a new configuration file is selected.

[0058] At S**280**, an inventory including all configuration files and their associated build files is created. That is, the inventory provides mappings between container images and their respective build files. As noted above, the inventory may be a data structure, such as a tree structure stored in a memory, storage, or database.

[0059] At S**290**, the inventory is enriched. To this end, the container layers of each container image designated in inventory is associated with a respective code line in the build file. Such code line is a command for creating the layer. In addition, each container image is enriched with any detected or reported vulnerability on a specific container image or layer within the image.

[0060] In an example embodiment, the method discussed with reference to FIG. **2**, is performed by the attribution system **150**, FIG. **1**.

[0061] FIG. **3** is an example flowchart **300** for hierarchical matching utilized in an embodiment when a container image is composed from multiple build files. At S**310**, for each candidate file, a list of commands is extracted or listed starting from the last command in the build until encountering the first excluded command. For example, the following are two candidate build files: TABLE-US-00002   build_file_2:   FROM scratch   apt-get install curl build_file_3:   FROM A   RUN ["echo 5"]

In this example, the list of commands includes: "apt-get install curl" and "RUN ["echo 5"]".

[0062] At S**320**, it is checked if the list of commands (from S**310**) is completely included in a selected configuration file (S**220**, FIG. **1**). If so, at S**330**, the matching condition is determined to

be satisfied. Otherwise, execution continues with S**340**. It should be noted that the matching of commands can be performed using the text searching techniques discussed above.

[0063] Execution reaches S**340**, when the build files are the same except for the excluded command(s). For example, for the following are two candidate build files:

TABLE-US-00003 build_file_4: FROM A RUN ["echo 5"] build_file_5: FROM B RUN ["echo 5"]

[0064] In this example, the list of commands is the same except of the "FROM" commands. To resolve this, a hierarchical representation (a dependencies tree) of a container image (based on its configuration file) is generated. This is performed by searching for container images (based on their respective configuration files) that are completely included in other container images.

[0065] At S**350**, a build file is associated with each node in the hierarchical representation starting from the parent node. For example, if the hierarchical representation is a tree, the association of build file will start from the root. The association of a build file to a node (representing a container image) is as discussed above with respect to FIG. **2**.

[0066] FIG. **4** shows an example for the hierarchical representation **400** generated from build files **1** through **4**. The hierarchical representation **400** includes two container images **410**-A and **410**-B to be associated with a container image **420**. That is, either container image **410**-A or **410**-B are included in image **420**.

[0067] Then, the root nodes (container images **410**-A and **410**-B) are associated with their respective build files. In the example, the container image **410**-A is associated with the file "build_file_2" and image **410**-A is associated with the file "build_file_1." This allows the association of container image **420** with the file "build_file_4", as the file "build_file_2" creates its parent image.

[0068] Returning to FIG. **3**, at S**360**, the matching of candidate build files and images as determined at S**350** is retuned.

[0069] FIG. **5** is an example schematic diagram of the attribution system **150** according to an embodiment. The attribution system **150** includes a processing circuitry **510** coupled to a memory **520**, a storage **530**, and a network interface **540**. In an embodiment, the components of the attribution system **150** may be communicatively connected via a bus **550**.

[0070] The processing circuitry **510** may be realized as one or more hardware logic components and circuits. For example, and without limitation, illustrative types of hardware logic components that can be used include field programmable gate arrays (FPGAs), application-specific integrated circuits (ASICs), Application-specific standard products (ASSPs), system-on-a-chip systems (SOCs), graphics processing units (GPUs), tensor processing units (TPUs), general-purpose microprocessors, microcontrollers, digital signal processors (DSPs), and the like, or any other hardware logic components that can perform calculations or other manipulations of information.

[0071] The memory **520** may be volatile (e.g., random access memory, etc.), non-volatile (e.g., read only memory, flash memory, etc.), or a combination thereof.

[0072] In one configuration, software for implementing one or more embodiments disclosed herein may be stored in the storage **530**. In another configuration, the memory **520** is configured to store such software. Software shall be construed broadly to mean any type of instructions, whether referred to as software, firmware, middleware, microcode, hardware description language, or otherwise. Instructions may include code (e.g., in source code format, binary code format, executable code format, or any other suitable format of code). The instructions, when executed by the processing circuitry **510**, cause the processing circuitry **510** to perform the various processes described herein.

[0073] The storage **530** may be magnetic storage, optical storage, and the like, and may be realized, for example, as flash memory or other memory technology, compact disk-read only memory (CD-ROM), Digital Versatile Disks (DVDs), or any other medium which can be used to store the desired information.

[0074] The network interface **540** allows the attribution system **150** to communicate with the repositories **120** and **130**, the detector **140**, and database **160**, through the network **170** as illustrated in FIG. **1**.

[0075] It should be understood that the embodiments described herein are not limited to the specific architecture illustrated in FIG. **1**, and other architectures may be equally used without departing from the scope of the disclosed embodiments.

[0076] The various embodiments disclosed herein can be implemented as hardware, firmware, software, or any combination thereof. Moreover, the software is preferably implemented as an application program tangibly embodied on a program storage unit or computer readable medium consisting of parts, or of certain devices and/or a combination of devices. The application program may be uploaded to, and executed by, a machine comprising any suitable architecture. Preferably, the machine is implemented on a computer platform having hardware such as one or more central processing units ("CPUs"), a memory, and input/output interfaces. The computer platform may also include an operating system and microinstruction code. The various processes and functions described herein may be either part of the microinstruction code or part of the application program, or any combination thereof, which may be executed by a CPU, whether or not such a computer or processor is explicitly shown. In addition, various other peripheral units may be connected to the computer platform such as an additional data storage unit and a printing unit. Furthermore, a non-transitory computer readable medium is any computer readable medium except for a transitory propagating signal.

[0077] All examples and conditional language recited herein are intended for pedagogical purposes to aid the reader in understanding the principles of the disclosed embodiment and the concepts contributed by the inventor to furthering the art, and are to be construed as being without limitation to such specifically recited examples and conditions. Moreover, all statements herein reciting principles, aspects, and embodiments of the disclosed embodiments, as well as specific examples thereof, are intended to encompass both structural and functional equivalents thereof. Additionally, it is intended that such equivalents include both currently known equivalents as well as equivalents developed in the future, i.e., any elements developed that perform the same function, regardless of structure.

[0078] It should be understood that any reference to an element herein using a designation such as "first," "second," and so forth does not generally limit the quantity or order of those elements. Rather, these designations are generally used herein as a convenient method of distinguishing between two or more elements or instances of an element. Thus, a reference to first and second elements does not mean that only two elements may be employed there or that the first element must precede the second element in some manner. Also, unless stated otherwise, a set of elements comprises one or more elements.

[0079] As used herein, the phrase "at least one of" followed by a listing of items means that any of the listed items can be utilized individually, or any combination of two or more of the listed items can be utilized. For example, if a system is described as including "at least one of A, B, and C," the system can include A alone; B alone; C alone; 2A; 2B; 2C; 3A; A and B in combination; B and C in combination; A and C in combination; A, B, and C in combination; 2A and C in combination; A, 3B, and 2C in combination; and the like.

Claims

1. A method for software containers attribution, comprising: de-compiling a configuration file of a container image of a software container in order to obtain a de-compiled configuration file, wherein the de-compiled configuration file includes a first plurality of commands; identifying a candidate build file in the de-compiled configuration file, wherein the candidate build file contains a second plurality of commands; and matching the configuration file to the candidate build file based on the

first plurality of commands and the second plurality of commands.

2. The method of claim 1, further comprising: updating an inventory with an association between the configuration file and the candidate build file when the configuration file is matched to the candidate build file.

3. The method of claim 2, wherein the association provides a mapping between the container image and the candidate build file.

4. The method of claim 1, wherein identifying the candidate build file further comprises: searching among a plurality of build files with respect to the first plurality of commands, wherein the candidate build file is identified such that at least one command of the second plurality of commands matches at least one command of the first plurality of commands.

5. The method of claim 1, wherein matching the configuration file to the candidate build file further comprises: matching the candidate build file to the de-compiled configuration file based on a matching condition, wherein the matching condition is defined with respect to at least one of: similarity; creation time; number of commands; and hierarchical matching.

6. The method of claim 1, wherein the candidate build file is a first candidate build file among a plurality of candidate build files, wherein the first candidate build file is matched to the de-compiled configuration file when all candidate build files among the plurality of candidate build files match the de-compiled configuration file.

7. The method of claim 1, wherein the candidate build file is a first candidate build file among a plurality of candidate build files, wherein the first candidate build file is matched to the de-compiled configuration file when a creation time of the candidate build file is earlier than a creation time of the container image of the de-compiled configuration file.

8. The method of claim 1, wherein the candidate build file is a first candidate build file among a plurality of candidate build files, wherein the first candidate build file is matched to the de-compiled configuration file when a number of commands excluding FROM commands among the first plurality of commands is equal to a number of commands among the second plurality of commands excluding FROM commands.

9. The method of claim 1, wherein the container image is formed from a plurality of build files, further comprising: generating a hierarchical representation of the container image, wherein the hierarchical representation includes a plurality of nodes arranged in a hierarchy; and associating each build file of the plurality of build files with a respective node of the plurality of nodes, wherein the candidate build file is identified based on the association between each build file of the plurality of build files and a respective node of the plurality of nodes.

10. A non-transitory computer readable medium having stored thereon instructions for causing a processing circuitry to execute a process, the process comprising: de-compiling a configuration file of a container image of a software container, the de-compiled configuration file including a first plurality of commands; identifying a candidate build file in the de-compiled configuration file, wherein the candidate build file contains a second plurality of commands; and matching the configuration file to the candidate build file based on the first plurality of commands and the second plurality of commands.

11. A system for remediating software containers attribution, comprising: a processing circuitry; and a memory, the memory containing instructions that, when executed by the processing circuitry, configure the system to: de-compile a configuration file of a container image of a software container, the de-compiled configuration file including a first plurality of commands; identify a candidate build file in the de-compiled configuration file, wherein the candidate build file contains a second plurality of commands; and match the configuration file to the candidate build file based on the first plurality of commands and the second plurality of commands.

12. The system of claim 11, wherein the system is further configured to: update an inventory with an association between the configuration file and the candidate build file when the configuration file is matched to the candidate build file.

13. The system of claim 12, wherein the association provides a mapping between the container image and the candidate build file.

14. The system of claim 11, wherein the system is further configured to: search among a plurality of build files with respect to the first plurality of commands, wherein the candidate build file is identified such that at least one command of the second plurality of commands matches at least one command of the first plurality of commands.

15. The system of claim 11, wherein the system is further configured to: match the candidate build file to the de-compiled configuration file based on a matching condition, wherein the matching condition is defined with respect to at least one of: similarity; creation time; number of commands; and hierarchical matching.

16. The system of claim 11, wherein the candidate build file is a first candidate build file among a plurality of candidate build files, wherein the first candidate build file is matched to the de-compiled configuration file when all candidate build files among the plurality of candidate build files match the de-compiled configuration file.

17. The system of claim 11, wherein the candidate build file is a first candidate build file among a plurality of candidate build files, wherein the first candidate build file is matched to the de-compiled configuration file when a creation time of the candidate build file is earlier than a creation time of the container image of the de-compiled configuration file.

18. The system of claim 11, wherein the candidate build file is a first candidate build file among a plurality of candidate build files, wherein the first candidate build file is matched to the de-compiled configuration file when a number of commands excluding FROM commands among the first plurality of commands is equal to a number of commands among the second plurality of commands excluding FROM commands.

19. The system of claim 11, wherein the container image is formed from a plurality of build files, wherein the system is further configured to: generate a hierarchical representation of the container image, wherein the hierarchical representation includes a plurality of nodes arranged in a hierarchy; and associating each build file of the plurality of build files with a respective node of the plurality of nodes, wherein the candidate build file is identified based on the association between each build file of the plurality of build files and a respective node of the plurality of nodes.