Vulnerability Exploitability eXchange (VEX) - Status Justifications

Publication date: June 2022

Abstract

A Vulnerability Exploitability eXchange (VEX) document must include the product's status as it relates to a particular vulnerability. Consequently, VEX documents may contain a justification statement of why the VEX document creator chose to assert that the product's status is NOT AFFECTED. This document provides the recommended NOT AFFECTED status justifications of a VEX document and offers the reader examples of when the different status justifications might be used. This document is part of a series of descriptions and guidance documents for VEX.

1.0 Introduction

Vulnerability Exploitability eXchange (VEX) is a form of a security advisory, similar to those already issued by mature product security teams.[1] The goal of VEX is to allow a software supplier or other parties to assert the status of specific vulnerabilities in a particular product. VEX documents allow both suppliers and users to focus on vulnerabilities that pose the most immediate risk, while not investing time in searching for or patching vulnerabilities that are not exploitable and therefore have no impact. VEX product statuses are not intended to be a discussion-ending declaration but a way to empower consumers to make informed decisions.

This document is meant to give guidance on what the *optional* status justifications are and when may be the proper time to use them in a VEX document. This document will not address chained attacks involving future or unknown risks as it will be considered out of scope.

The definitions, examples and diagrams below clarify when an organization may want to use a particular VEX document status justification. This document does not explicitly offer recommendations about the optimal organization. Suppliers are encouraged to work with their customers and the vendor community to identify the approach that meets respective needs.

[1] Additional information on SBOM and VEX can be found at https://www.cisa.gov/sbom and https://www.ntia.gov/SBOM.

2.0 Status Justifications Overview

VEX documents must contain the following information: VEX metadata, product details, vulnerability details, and product status. Product status details include status information about a vulnerability in a product and must be included in the document as one of the following: NOT AFFECTED, AFFECTED, FIXED, and UNDER INVESTIGATION[2].

A VEX document's product status details may also include a status justification in an optional, machine-readable field. A status justification will help consumers understand the circumstances under which an assertion was made that a product is NOT AFFECTED.[3]

Here is a list of the NOT AFFECTED status justifications:

- Component_not_present
- Vulnerable_code_not_present
- Vulnerable_code_cannot_be_controlled_by_adversary
- Vulnerable_code_not_in_execute_path
- Inline_mitigations_already_exist

Status justifications fields, like all fields in VEX documents at this stage, are assertions made by the author of the document, either the product supplier or any other party. The attestation is designed to be machine readable. Like all assertions, the consumer of the VEX document may choose to accept it or not. The goal of these status justifications is to offer more context and detail about these claims, in a machine-readable fashion, to facilitate trust and allow automated policy-based actions by the user.

VEX documents and status justifications are based on specific *known vulnerabilities.* Many attackers use chains of vulnerabilities to exploit a system. The assertions in a VEX document, including the status justifications laid out in this overview, are built around known risks to software and components. New vulnerabilities may emerge, which could affect a product.

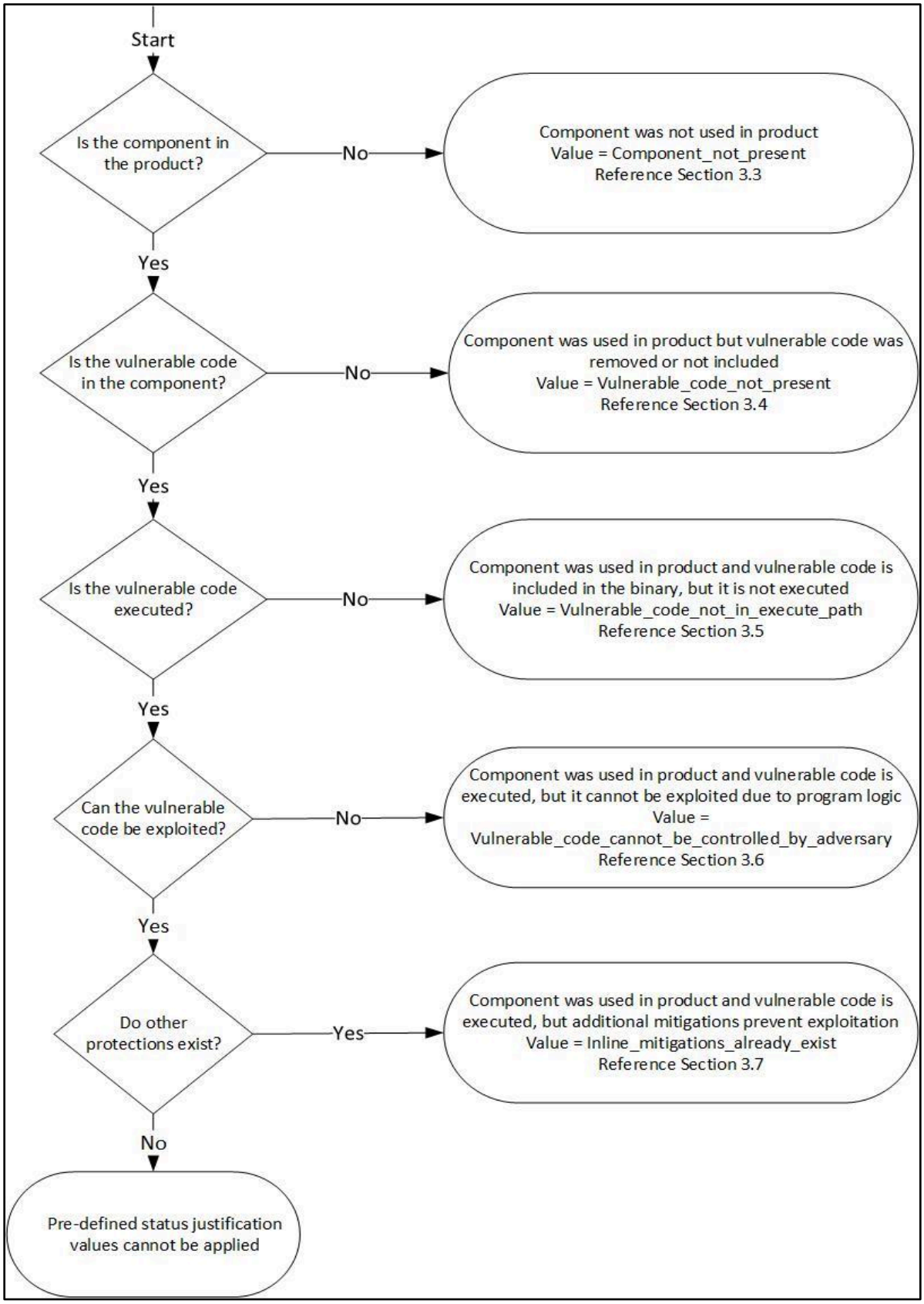
[2] The VEX product statuses are defined in the VEX Use Cases document found at https://www.cisa.gov/sites/default/files/publications/VEX_Use_Cases_Aprill2022.pdf.

[3] Information about other optional, machine-readable fields to justify additional product statuses (i.e., AFFECTED, FIXED, and UNDER INVESTIGATION) will be the topic of future VEX Working Group documents.

3.0 Status Justification Definitions, Diagrams & Examples

3.1 Status Justification Introduction

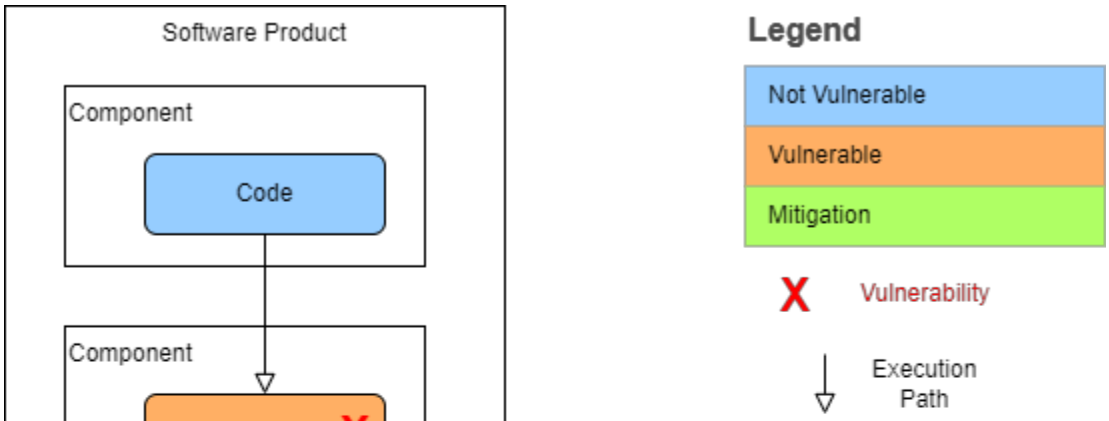
The following sections illustrate the pre-defined status justification field values in detail. Status justification values and use cases may be difficult for non-developers to understand, so it may be helpful to review the flowchart below to determine which of the use cases applies to a given status justification assertion. The flowchart illustrates the key factors that contribute to status justifications and suggests the correct value that applies. If none of these factors are present, the flowchart identifies when the pre-defined status justification values cannot be applied.



VEX Status Justification Decision Flowchart

3.2 Status Justification Diagrams

Readers may find the status justification diagrams throughout the document helpful in visually understanding the status justification definitions for NOT AFFECTED product status assertions. The diagram below is an example of an AFFECTED software product and shows the reader the general layout of all diagrams and includes a color-coded legend for understanding whether a component is vulnerable or not vulnerable and whether there is a mitigation present within the execution path.



Affected Software Product Diagram Example

3.3 Component_not_present

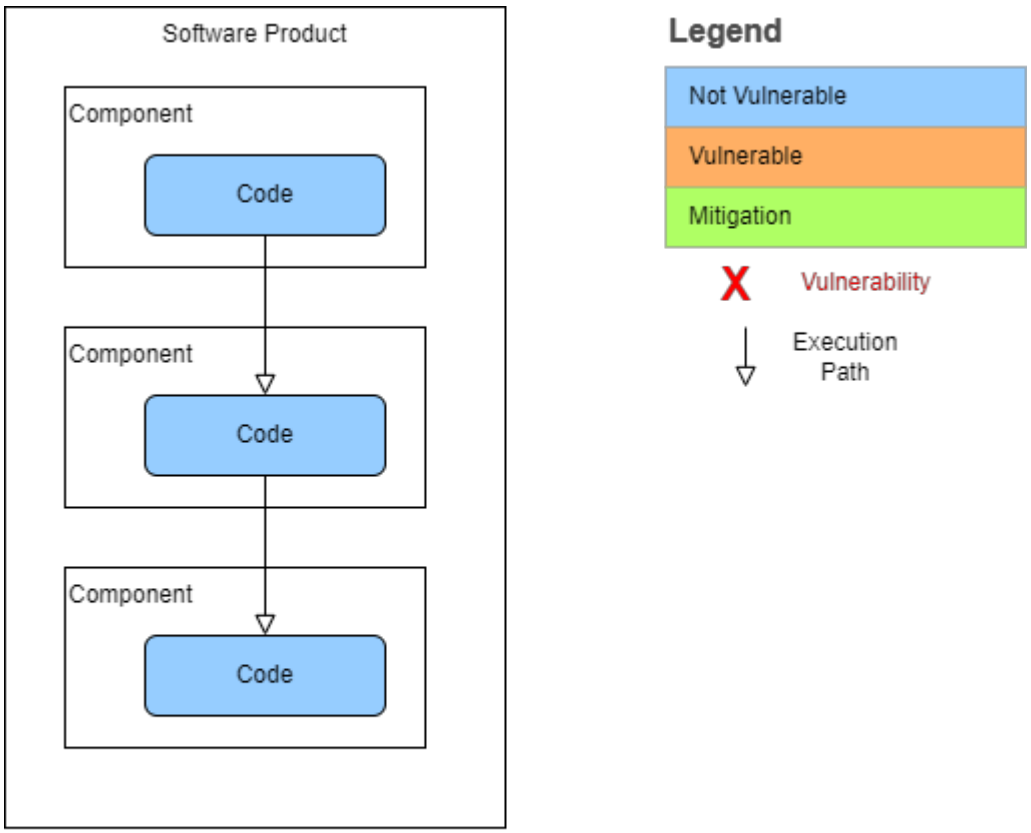
This status justification may be used when the product is not affected by the vulnerability because the component is not included in the product. The status justification may be used to preemptively inform product users who are seeking to understand a vulnerability that is widespread, receiving a lot of attention, or is in similar products.

3.3.1 “Component_not_present” Examples

Examples of this status justification include:

- The product is written in Python and Elixir and does not have any Java code present; therefore, the product could not be affected by the Log4j vulnerability.
- Because Java JAR files often contain other JAR files, automation may incorrectly flag a component as existing in a JAR file without verifying it as actually present.
- Base layer container images often contain unused packages. A later layer could remove one or more of these packages. A container image may be expected to contain ‘sudo’ but it was removed prior to deployment.

3.3.2 “Component_not_present” Diagram



Component_not_present Diagram Example

3.4 Vulnerable_code_not_present

This status justification may be used when the product software is not affected because the code underlying the vulnerability is not present in the product. Unlike the

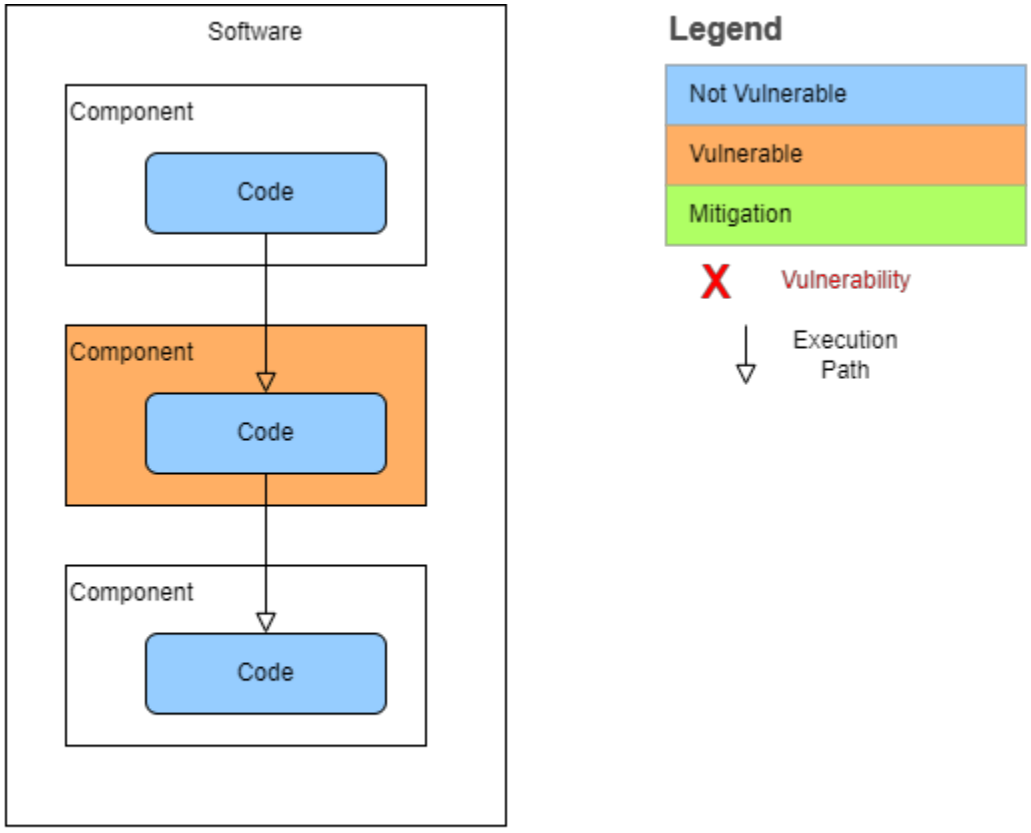
“Component_not_present" status justification, the component in question is present, but the specific code causing the vulnerability is not present in the component.

3.4.1 “Vulnerable_code_not_present” Examples

Examples of this status justification include:

- The compiler has stripped out the relevant function.
- The vulnerable code may have been added in a later version of a component and is not present in the version shipped in the product.

3.4.2 “Vulnerable_code_not_present” Diagram



Vulnerable_code_not_present Diagram Example

3.5 Vulnerable_code_not_in_execute_path

This status justification may be used when the vulnerable code can never be executed in the context of the application. This situation can occur when third-party libraries are included but where the application only uses a fraction of the library functions directly or indirectly. A single

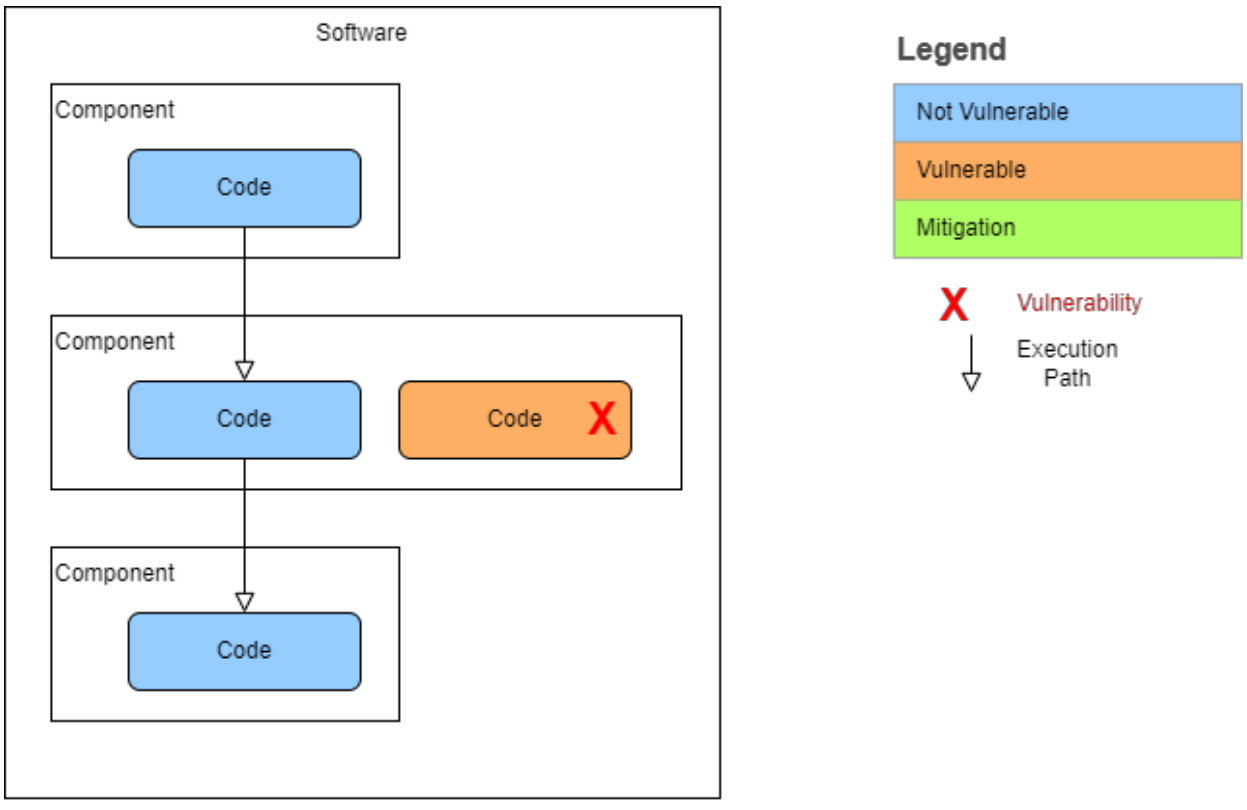
path of execution should not be assumed since the attacker may be able to divert the path of execution.

3.5.1 “Vulnerable_code_not_in_execute_path” Examples

Examples of this status justification include:

- An old library ships with the distribution but is not invoked by the software.
- A vulnerability in a binary has been patched, but the unpatched binary remains for “roll-back” purposes.
- In a container image, there will be operating system utilities that are not used by the application deployed inside the image. Those utilities cannot be removed from the container image and are not used by the deployed application in any way.
- Included libraries often have many functions. A vulnerability in a library in a function that is never called directly or indirectly also has this status. Applications that only used OpenSSL for secure random function were not vulnerable to the Heartbleed protocol vulnerability which was never called from secure random.
- The code is only executed by a hardware module that is not installed.

3.5.2 “Vulnerable_code_not_in_execute_path” Diagram



Vulnerable_code_not_in_execute_path Diagram Example

3.6 Vulnerable_code_cannot_be_controlled_by_adversary

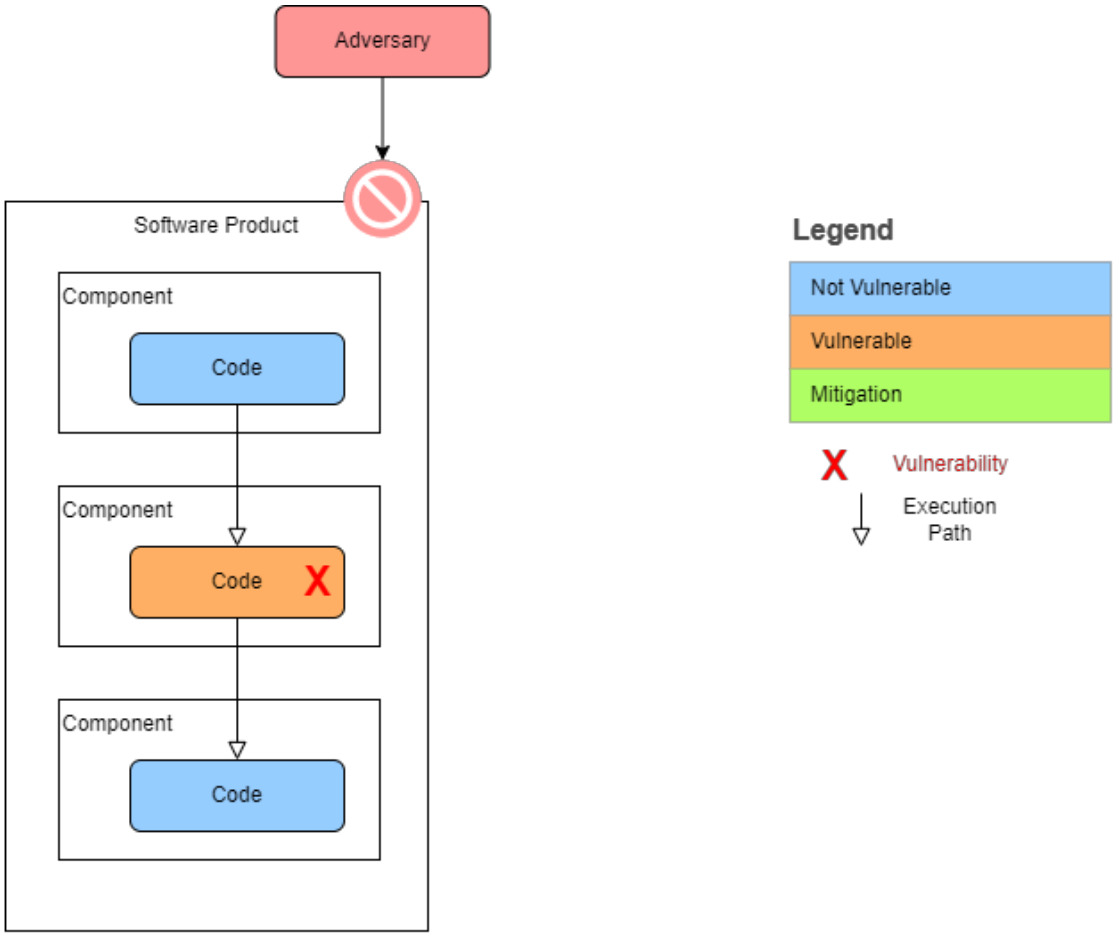
This status justification may be used when the vulnerable code cannot be controlled by an attacker to exploit the vulnerability without exploiting at least one other vulnerability. The vulnerable component is present and the component contains the vulnerable code. However, due to some other circumstances, the vulnerability in question cannot be exploited by an attacker.

3.6.1 “Vulnerable_code_cannot_be_controlled_by_adversary” Examples

Examples of this status justification include:

- A hard-coded variable that denies user-generated input for the vulnerable function.
- A logging facility that is only called as a result of hardware malfunction or during installation.
- Software is vulnerable to the EternalBlue vulnerability, but SMB ports 139 and 445 are disabled, and the user does not have access to the O/S to enable the ports. The vulnerability cannot be exploited by an attacker because the ports are unavailable. No additional action is required by the operator.

3.6.2 “Vulnerable_code_cannot_be_controlled_by_adversary” Diagram



Vulnerable_code_cannot_be_controlled_by_adversary Diagram Example

3.7 Inline_mitigations_already_exist

This status justification may be used when the product includes built-in protections that prevent exploitation of the vulnerability. These built-in protections cannot be subverted by the attacker and cannot be configured or disabled by the user. These mitigations completely prevent exploitation based on known attack vectors.

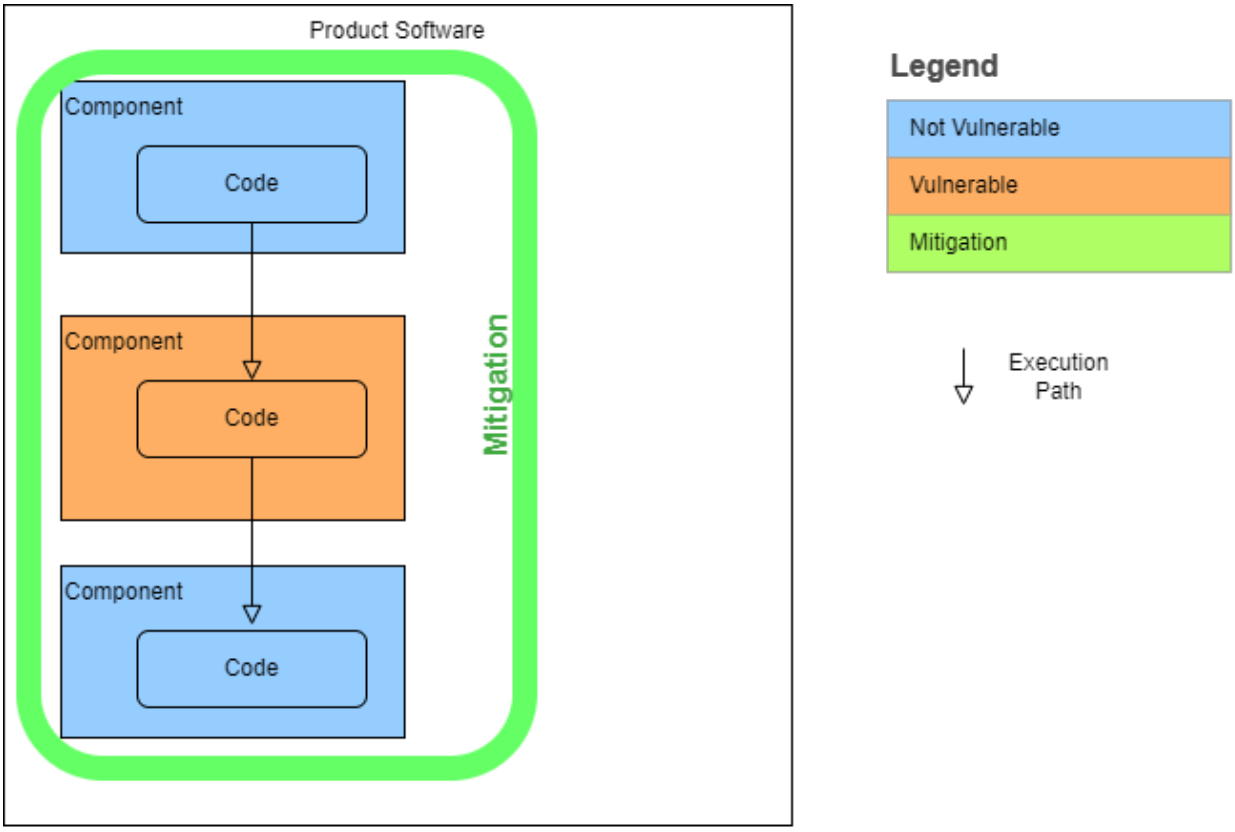
Note: Mitigations are effective only in the scenario where the threat model of the software allows it to perform its protections successfully. A mitigation which negates a vulnerability in one software/service may not necessarily be the case in another software/service.

3.7.1 “Inline_mitigations_exist” Examples

Examples of this status justification include:

- Code sanitizes input further upstream.
- Sandboxing using memory separation techniques through software or hardware.
- Output and effects of the component that uses vulnerable code is verifiable.
- A buffer-overflow due to multiplication does not occur if one of the two arguments always evaluates to 1.
- Software is vulnerable to the EternalBlue vulnerability and the SMB port is enabled, but the device integrates a firewall configuration that prevents network access to the ports. The user does not have access to the firewall to change the configuration to allow the communication. The device firewall provides inline protection by blocking access to the port. No additional action is required by the user.

3.7.2 “Inline_mitigations_exist” Diagram



Inline_mitigations_already_exist Diagram Example

3.7.3 - Difference between “Vulnerable_code_cannot_be_controlled_by_adversary” and “Inline_mitigations_already_exist”

The status justifications “Vulnerable_code_cannot_be_controlled_by_adversary” (see section 3.6) and “Inline_mitigations_already_exist” (see section 3.7) are similar, but with some key distinctions. In both cases, the vulnerable code is in the product, and may be expected to be executed. Thus, both rely on an expert analysis of the structure of the code to determine that the product is, in fact, not affected.

However, when the code cannot be controlled by the adversary, the assertion is that there is nothing that the attacker can do to influence the execution of the code to trigger the vulnerability. Alternatively, when mitigations already exist, the adversary may influence the execution of the code to trigger the vulnerability, but some other control blocks malicious execution. Mitigations that cannot be subverted or modified by attackers or users completely prevent exploitation. As an example for section 3.6, the inputs to a function may not be user-created, while in section 3.7, the user-created string is sanitized elsewhere within the product.

4.0 Document Feedback

If you have any feedback on the contents of this paper, please send us your thoughts at SBOM@cisa.dhs.gov. Your feedback will be extremely valuable to us in making continuous improvements to the paper. VEX documents are starting to be used across the software ecosystem, and the concept will continue to be refined as implementers and adopters encounter new challenges and opportunities. Furthermore, this document does not contain an exhaustive list of status justifications and future work may contain new status justifications as demand arises.

Appendix A: About this document

This document was a product of the VEX Working Group, which grew out of the NTIA Multistakeholder Process and the Framing Working Group, initially beginning work in 2020. That work continued into 2022, facilitated by CISA.

Participants included:

Allan Friedman, CISA
Art Manion, CERT/CC
Brandon Lum, Google

Bruce Lowenthal, Oracle Corporation
Bryan Cowan, Fortress Information Security
Charlie Hart, Hitachi America Ltd.
Chris Campbell, SolarWinds
Derek Kolakowski, FoxGuard
Derek Kruszewski, aDolus Technology Inc.
Dmitry Raidman, Cybeats
Duncan Sparrell, sFractal Consulting
Ed Heierman, Abbott
Eliot Lear, Cisco Systems
Dr. Hans-Martin von Stockhausen, Siemens Healthineers
Ivana Atanasova, VMware
Jeremiah Stoddard, INL
Jim Jacobson, Siemens Healthineers
Josh Bressers, Anchore
Justin Murphy, CISA
Kate Stewart, The Linux Foundation
Matthew Paulsen, Juniper Networks, Inc.
Michael Hoover, INL
Nisha Kumar, Oracle
Rich Steenwyk, GE Healthcare
Steve Springett, OWASP
Thomas Schmidt, Federal Office for Information Security (BSI) Germany
Timothy Klett, INL
Timothy Walsh CISSP, Mayo Clinic
Tom Alrich, Tom Alrich LLC
Tony Turner, Fortress Information Security

Others participated, but do not wish to be named. Input into this document and the broader VEX effort included feedback from multiple presentations, with a particular appreciation for feedback from the Healthcare SBOM PoC community.