Chapter 27. Vulnerability Management

Part of any good SSDL process is a well-defined pipeline for obtaining, triaging, and resolving vulnerabilities found in a web application. We covered methods of discovering vulnerabilities in Chapter 26, and prior to that we covered methods of integrating SSDL into your architecture and development phases to reduce the number of outstanding vulnerabilities found.

Vulnerabilities in a large application will be found in all of these phases, from the architecture phase to production code. Vulnerabilities noted in the architecture phase can be defensively coded against, and countermeasures can be developed before any code is written. Vulnerabilities found any time after the architecture phase need to be properly managed so they can eventually be fixed and any affected environment patched with the fix. This is where a vulnerability management pipeline comes into play.

Reproducing Vulnerabilities

After a vulnerability report, the first step to manage it should be reproducing the vulnerability in a production-like environment. This has multiple benefits. First off, it allows you to determine if the vulnerability is indeed a vulnerability. Sometimes user-defined configuration errors can look like a vulnerability. For example, a user "accidentally" makes an image on your photo-hosting app "public" when they usually set their photos to "private."

To reproduce vulnerabilities efficiently, you need to establish a *staging* environment that mimics your production environment as closely as pos-

sible. Because setting up a staging environment can be difficult, the process should be fully automated.

Prior to releasing a new feature, it should be available in a build of your application that is only accessible via the internal network or secured via some type of encrypted login.

Your staging environment, while mimicking a real production environment, does not need real users or customers. However, it should contain mock users and mock objects in order to both visually and logically represent the function of your application in production mode.

By reproducing each vulnerability that is reported, you can safely avoid wasting engineering hours on false positives. Additionally, vulnerabilities reported externally through a paid program like a bug bounty program should be reproduced so that a bounty is not paid for a false positive vulnerability.

Finally, reproducing vulnerabilities gives you deeper insight as to what could have caused the vulnerability in your codebase and is an essential first step for resolving the vulnerability. You should reproduce right away and log the results of your reproduction.

Ranking Vulnerability Severity

After reproducing a vulnerability, you should have gained enough context into the function of the exploit to understand the mechanism by which the payload is delivered, and what type of risk (data, assets, etc.) your application is vulnerable to as a result. With this context in mind, you should begin ranking vulnerabilities based on severity.

To properly rank vulnerabilities, you need a well-defined and easy-to-follow scoring system that is robust enough to accurately compare two vulnerabilities, but flexible enough to apply to uncommon forms of vulnerability as well. The most commonly used method of scoring vulnerabilities is the Common Vulnerability Scoring System.

Common Vulnerability Scoring System

The Common Vulnerability Scoring System (CVSS) is a no-cost and widely available system for ranking vulnerabilities based on how easy they are to exploit and what type of data or processes can be compromised as a result of a successful exploitation (see Figure 27-1). CVSS is a fantastic starting point for organizations with a limited budget or lack of dedicated security engineers.

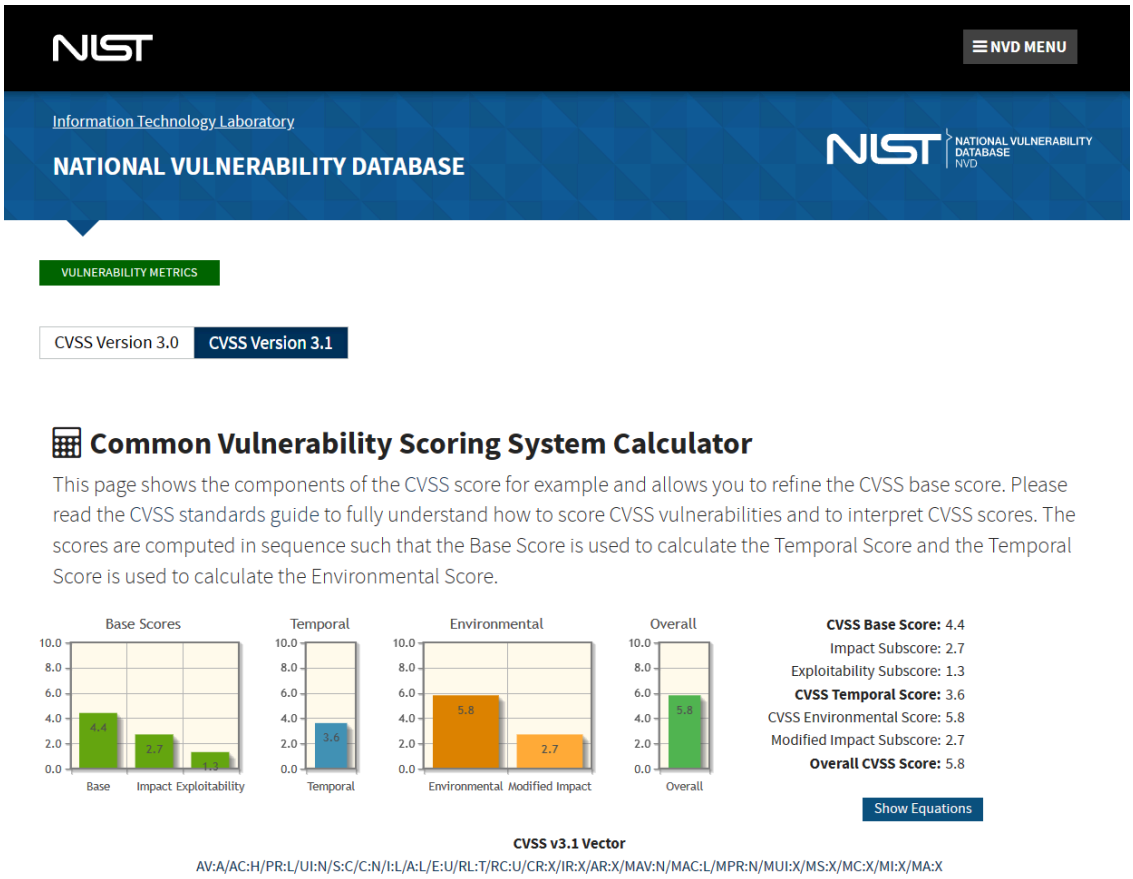

Figure 27-1. CVSS is a time-tested vulnerability scoring system that is widely available on the web and well documented

CVSS is intended as a general-purpose vulnerability scoring system, and as a result, it is often criticized for not being able to accurately score all types of systems or rare, unique, or chained vulnerabilities. That being said, as a general-purpose vulnerability scoring system for common (OWASP top 10) vulnerabilities, this open vulnerability scoring framework does a good job.

The CVSS system is on version 3.1 at the time of this writing. It breaks down vulnerability scoring into a few important subsections:

- Base—scoring the vulnerability itself
- Temporal—scoring the severity of a vulnerability over time
- Environmental—scoring a vulnerability based on the environment it exists in

Most commonly, the CVSS base score is used, and the temporal and environmental scores are used only in more advanced cases. Let's look at each of these scores in a bit more depth.

CVSS: Base Scoring

The CVSS v3.1 base scoring algorithm requires eight inputs (see Figure 27-2):

- Attack Vector (AV)
- Attack Complexity (AC)
- Privileges Required (PR)
- User Interaction (UI)
- Scope (S)
- Confidentiality Impact (C)
- Integrity Impact (I)
- Availability Impact (A)

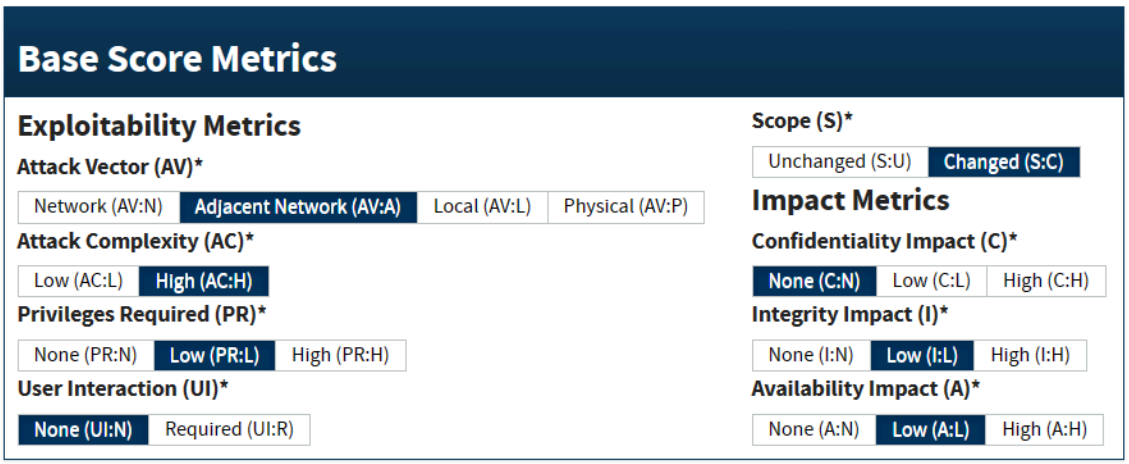

Figure 27-2. CVSS base score is the core component of the CVSS algorithm, which scores a vulnerability based on severity

Each of these inputs accepts one of several options, leading to the generation of a base score:

Attack Vector option

Attack Vector accepts Network, Adjacent, Local, and Physical options. Each option describes the method by which an attacker can deliver the vulnerability payload. Network is the most severe, whereas physical is the least severe due to increased difficulty of exploitation.

Attack Complexity option

Attack Complexity accepts two options, Low or High. The Attack Complexity input option refers to the difficulty of exploitation, which can be described as the number of steps (recon, setup) required prior to delivering an exploit as well as the number of variables outside of a hacker's control.

An attack that could be repeated over and over again with no setup would be Low, whereas one that required a specific user to be logged in at a specific time and on a specific page would be High.

Privileges Required option

Privileges Required describes the level of authorization a hacker needs to pull off the attack: None, Low, and High. A High privilege attack could only be initiated by an admin, while Low might refer to a normal user, and None would be a guest.

User Interaction option

The User Interaction option has only two potential inputs, None and Required. This option details if user interaction (clicking a link) is required for the attack to be successful.

Scope option

Scope suggests the range of impact successful exploitation would have. "Unchanged" scope refers to an attack that can only affect a local system, such as an attack against a database affecting that database. "Changed" scope refers to attacks that can spread outside of the functionality where the attack payload is delivered, such as

an attack against a database that can affect the operating system or filesystem as well.

Confidentiality option

Confidentiality takes one of three possible inputs: None, Low, and High. Each input suggests the type of data compromised based on its impact to the organization. The severity derived from confidentiality is likely based on your application's business model, as some businesses (health care, for example) store much more confidential data than others.

Integrity option

Integrity also takes one of three possible inputs: None, Low, and High. The None option refers to an attack that does not change application state, while Low changes some application state in limited scope, and High allows for the changing of all or most application state. Application state is generally used when referring to the data stored on a server, but could also be used in regard to local client-side stores in a web application (local storage, session storage, IndexedDB).

Availability option

Availability takes one of three possible options: None, Low, and High. It refers to the availability of the application to legitimate users. This option is important for DoS attacks that interrupt or stop the application from being used by legitimate users, or code execution attacks that intercept intended functionality.

Entering each of these scores into the CVSS v3.1 algorithm will result in a number between 0 and 10. This number is the severity score of the vulnerability, which can be used for prioritizing resources and timelines for fixes. It can also help determine how much risk your application is exposed to as a result of the vulnerability being exploited.

CVSS scores can be easily mapped to other vulnerability scoring frameworks that don't use numerical scoring:

- 0.1–4: low severity
- 4.1–6.9: medium severity
- 7–8.9: high severity
- 9+: critical severity

By using the CVSS v3.1 algorithm, or one of the many web-based CVSS calculators, you can begin scoring your found vulnerabilities in order to aid your organization in prioritizing and resolving risk in an effective manner.

CVSS: Temporal Scoring

Temporal scoring in CVSS is simple, but due to complicated wording, it can sound daunting. Temporal scores show you how well equipped your organization is to deal with a vulnerability, given the state of the vulnerability at the time of reporting (see Figure 27-3).

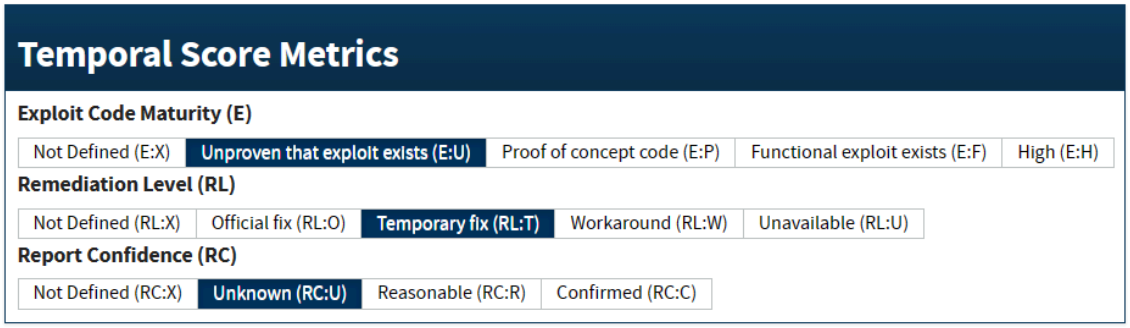

Figure 27-3. The CVSS temporal score scores a vulnerability based on the maturity of security mechanisms in your codebase

The temporal score has three categories:

Exploitability

Accepts a value from “unproven” to “high.” This metric attempts to determine if a reported vulnerability is simply a theory or proof of concept (something that would require iteration to turn into an actual usable vulnerability), or if the vulnerability can be deployed and used as is (working vulnerability).

Remediation Level

The Remediation Level takes a value suggesting the level of mitigations available. A reported vulnerability with a working tested fix being delivered would be a “O” for “Official Fix,” while a vulnerability with no known solution would be a “U” for “Unavailable.”

Report Confidence

The Report Confidence metric helps determine the quality of the vulnerability report. A theoretical report with no reproduction code or understanding of how to begin the reproduction process would appear as an “Unknown” confidence, while a well-written report with a reproduction and description would be a “Confirmed” report confidence.

The temporal score follows the same scoring range (0–10), but instead of measuring the vulnerability itself, it measures the mitigations in place and the quality and reliability of the vulnerability report.

CVSS: Environmental Scoring

CVSS environmental scores detail your particular environment (specific to your application) in order to understand what data or operations would present the most risk to your organization if a hacker were to exploit them (see Figure 27-4).

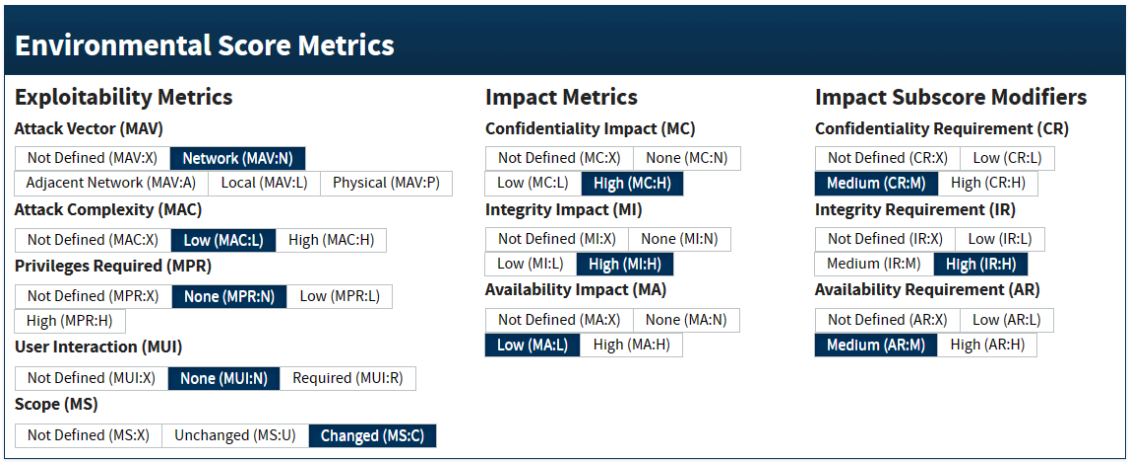

Figure 27-4. The CVSS environmental score measures a vulnerability based on the context (environment) in which it would be exploited

The environmental scoring algorithm takes all of the base score inputs, but adds to three requirements that detail the importance of confidential-

ity, integrity, and availability to your application.

The three new fields are as follows:

Confidentiality Requirement

> The level of confidentiality your application requires. Freely available public applications may score lower, whereas applications with strict contractual requirements (health care, government) would score higher.

Integrity Requirement

> The impact of application state being changed by a hacker in your organization. An application that generates test sandboxes that are designed to be thrown away would score lower than an application that stores crucial corporate tax records.

Availability Requirement

> The impact on the application as a result of downtime. An application expected to be live 24/7 would be impacted more than an application with no uptime promises.

The environmental score scores a vulnerability relative to your application's requirements, while the base score scores a vulnerability by itself in a vacuum.

Advanced Vulnerability Scoring

Using CVSS or another well-tested open scoring system as a starting point, you can begin to develop and test your own scoring system. This allows more relevant information in regard to your particular business model and application architecture.

TIP

If your web application interfaces with physical technology, you may want to develop your own scoring algorithms to include risks that come with connected web applications.

For example, a security camera controlled by a web portal would have additional implications if its systems were compromised because it could leak sensitive photos or videos of its tenants—potentially breaking the law.

Applications that connect with IoT devices, or are delivered by other means, may want to begin working on their own scoring system right out of the gate.

Any scoring system should be evaluated over time, based on its ability to prevent damage to your application, its subsystems, and your organization.

Beyond Triage and Scoring

After a vulnerability has been properly reproduced, scored, and triaged, it needs to be fixed. Scoring can be used as a metric for prioritizing fixes, but it cannot be the only metric. Other business-centric metrics must be considered as well, such as customer contracts and business relationships.

Fixing a vulnerability correctly is just as important as finding and triaging it correctly. Whenever possible, vulnerabilities should be resolved with permanent, application-wide solutions. If a vulnerability cannot (yet) be resolved in that way, a temporary fix should be added, but a new bug should be opened detailing the still-vulnerable surface area of your application.

Never ship a partial fix and close a bug (in whatever bug tracking software you use) unless another bug detailing the remaining fixes with an appropriate score is opened first. Closing a bug early could result in hours of lost reproduction and technical understanding. Plus, not all vulnerabil-

ities will be reported. And vulnerabilities can grow in risk to your organization as the features your application exposes increase (increased surface area).

Finally, every closed security bug should have a regression test shipped with it. Regression tests grow increasingly more valuable over time, as opportunities for regression increase exponentially with the size and feature set of a codebase.

Summary

Vulnerability management is a combination of very important but particular tasks.

First, a vulnerability needs to be reproduced and documented by an engineer. This allows an organization to be sure the report is valid and to understand if the impact is deeper than originally reported. This process should also give insight into the amount of effort required for resolving the vulnerability.

Next, a vulnerability should be scored based on some type of scoring system that allows your organization to determine the risk the vulnerability exposes your application to. The scoring system used for this does not matter as much as its relevance to your business model and its ability to accurately predict the damage that could be done to your application as a result of exploitation.

After properly reproducing and scoring a vulnerability (the “triage” step), a vulnerability must be resolved. Ideally, a vulnerability should be resolved with a proper fix that spans the entire application surface area and is well tested to avoid edge cases. When this is not possible, partial fixes should be deployed and additional bugs should be filed detailing still-vulnerable surface area.

Finally, as each bug is resolved, a proper security regression test should be written so that the bug cannot be accidentally reopened or reimplemented at a later date.

Successfully following these steps will dramatically reduce the risk your organization is exposed to as vulnerabilities are found and aid your organization in rapidly and efficiently resolving vulnerabilities based on the potential damage they could have in your organization.