KonvergeAI

CCT 310 - 3 Application Security

UNIT 3 Web Application Security - II

Securing Modern Web Applications

The first step in writing a well-fortified web application starts prior to any software actually being written. This is the architecture phase. In the architecture phase of any new product or feature, deep attention to detail should be spent on the data that flows throughout the application.

- It could be argued that most of software engineering is efficiently moving data from point A to point B. Similarly, most of security engineering is efficiently securing data in transit from point A to point B and wherever it may rest before, after, or during that transit.
- It is much easier to catch and resolve deep architectural security flaws before actually writing and deploying the software. After an application has been adopted by users, the depth at which a re-architecture can be performed to resolve a security gap is often limited.
- This is especially true in any type of web application that consumers build upon. Web applications that allow users to open their own stores, run their own code, and so on can be extremely costly to re-architect because deep re-architecture may require customers to redo many time-significant manual processes.

Comprehensive Code Reviews

During the process of actually writing a web application that has already been evaluated as a secure architecture, the next step is carefully evaluating each commit prior to release into the codebase. Most companies have already adopted mandatory code review processes to improve quality assurance, reduce technical debt, and eliminate easy-to-find programming mistakes.

Code reviews are also a crucial step in ensuring that released code meets security standards. In order to reduce conflict of interest, commits to source code version control should not only be reviewed by members of the committer's team, but also by an unrelated team (especially in regard to security).

Catching security holes at the code review level on a per-commit basis is actually easier than one would think. The major points to look out for are:

- ✔ How is data being transmitted from point A to point B (typically over a network, and in a specific format)?
- ✔ How is data being stored?
- ✔ When data gets to the client, how is it presented to the user?
- ✔ When data gets to the server, what operations occur on it and how is it persisted?

Vulnerability Discovery

Assuming your organization and/or codebase has already undertaken steps to evaluate security before writing code (architecture) and during the development process (code reviews), the next step is finding vulnerabilities in the code that occur as a result of bugs that are not easily identifiable (or missed) in the code review process. Vulnerabilities are found in a number of ways, and some of these ways will damage your business/reputation, while others will not.

The old-fashioned way of finding vulnerabilities is either by customer notification or (worst case) widespread public disclosure. Unfortunately, some companies still rely on this as their only means of finding vulnerabilities to fix in their web applications.

More modern methods for finding vulnerabilities exist, and can save your product from a wave of bad PR, lawsuits, and loss of customers. Today's most security-conscious companies use a combination of the following:

- ✔ Bug bounty programs
- ✔ Internal red/blue teams
- ✔ Third-party penetration testers
- ✔ Corporate incentives for engineers to log known vulnerabilities

By making use of one or more of these techniques to find vulnerabilities before your customers or the public do, a large corporation can save huge amounts of money with a little bit of expense upfront.

Vulnerability Management

After assessing the risk of a vulnerability, and prioritizing it based on the factors listed, a fix must be tracked through to completion. Such fixes should be completed in a timely manner, with deadlines determined based off of the risk assessment. Furthermore, customer contracts should be analyzed in response to an assessed vulnerability to determine if any agreements have been violated.

- Also, during this time frame if the vulnerability can be recorded, additional logging should be put in place to ensure that no hacker attempts to take advantage of the vulnerability while the fix is being developed. Lack of logging for known vulnerabilities has led to the demise of several companies that were not aware a vulnerability was being abused while they waited for resolution from their engineering teams.
- Managing vulnerabilities is an ongoing process. Your vulnerability management process should be carefully planned out and written down so that your progress can be recorded. This should result in more accurate timelines as time goes on and time-to-fix burn rates can be averaged.

Regression Testing

After deploying a fix that resolves a vulnerability, the next step is to write a regression test that will assert that the fix is valid and the vulnerability no longer exists. This is a best practice that is not being used by as many companies as it should be. A large percentage of vulnerabilities are regressions—either directly reopened bugs or variations of an original bug. A security engineer from a large software company (10,000+ employees) once told that approximately 25% of their security vulnerabilities were a result of previously closed bugs regressing to open.

Building and implementing a vulnerability regression management framework is simple. Adding test cases to that framework should take a small fraction of the time that an actual fix took. Vulnerability regression tests cost very little upfront but can save huge amounts of time and money in the long run.

Mitigation Strategies

Finally, an overall best practice for any security-friendly company is to actively make a good effort to mitigate the risk of a vulnerability occurring in the application codebase. This is a practice that happens all the way from the architecture phase to the regression testing phase.

- Mitigation strategies should be widespread, like a net trying to catch as many fish as possible. In crucial areas of an application, mitigation should also run deep.
- Mitigation comes in the form of secure coding best practices, secure application architecture, regression testing frameworks, secure software development life cycle (SSDL), and secure-by-default developer mindset and development frameworks. Practicing all of the preceding steps will greatly enhance the security of any codebase you work on. It will eliminate huge amounts of risk from your organization, and save your organization large amounts of money while protecting you from huge amounts of brand damage that would occur otherwise in due time.

Secure Application Architecture

When building a product, a cross-functional team of software engineers and product managers usually collaborate to find a technical model that will serve a very specific business goal in an efficient manner. In software engineering, the role of an architect is to design modules at a high level and evaluate the best ways for modules to communicate with each other. This can be extended to determining the best ways to store data, what third-party dependencies to rely on, what programming paradigm should be predominant throughout the codebase, etc.

Similarly to a building architect, software architecture is a delicate process that carries a large amount of risk because re-architecture and refactor are expensive processes once an application has already been built. Security architecture includes a similar risk profile to software or building architecture. Often, vulnerabilities can be prevented easily in the architecture phase with careful planning and evaluation. However, too little planning, and application code must be re-architected and re-factored—often at a large cost to the business.

The NIST has claimed, based on a study of popular web applications, that “The cost of removing an application security vulnerability during the design phase ranges from 30–60 times less than if removed during production.” Hence solidifying any doubts we have regarding the importance of the architecture phase.

Analyzing Feature Requirements

The first step in ensuring that a product or feature is architected securely is collecting all of the business requirements that the product or feature is expected to implement. Business requirements can be evaluated for risk prior to their integration in a web application even being considered.

Consider this business case: after cleaning up multiple security holes in its codebase, MegaBank has decided to capitalize on its newly found popularity by beginning its own merchandising brand. MegaBank's new merchandising brand, MegaMerch, will offer a collection of high-quality cotton T-shirts, comfortable cotton/elastic sweatpants, and men's and women's swimwear with the MegaMerch (MM) logo.

In order to distribute merchandise under the new MegaMerch brand, MegaBank would like to set up an ecommerce application that meets the following requirements:

Users can create accounts and sign in.

User accounts contain the user's full name, address, and date of birth.

Users can access the front page of the store that shows items.

Users can search for specific items.

A high-level analysis of these requirements tells us a few important tidbits of information:

We are storing credentials.
We are storing personal identifier information.
Users have elevated privileges compared to guests.
Users can search through existing items.
We are storing financial data.

These points, while not out of the ordinary, allow us to derive an initial analysis of what potential risks this application could encounter if not architected correctly. A few of the risk areas derived from this analysis are as follows:

Authentication and authorization: How do we handle sessions, logins, and cookies?

Personal data: Is it handled differently than other data? Do laws affect how we should handle this data?

Search engine: How is the search engine implemented? Does it draw from the primary database as its single source of truth or use a separate cached database?

Each of these risks brings up many questions about implementation details, which provide surface area for a security engineer to assist in developing the application in a more secure direction.

Authentication and Authorization

Because we are storing credentials and offering a different user experience to guests and registered users, we know we have both an authentication and an authorization system. This means we must allow users to log in, as well as be able to differentiate among different tiers of users when determining what actions these users are allowed.

Furthermore, because we are storing credentials and support a login flow, we know there are going to be credentials sent over the network. These credentials must also be stored in a database, otherwise the authentication flow will break down.

This means we have to consider the following risks:

How do we handle data in transit?

How do we handle the storage of credentials?

How do we handle various authorization levels of users?

SSL Certificate

Let's consider three cases where a hacker gets access to MegaMerch's databases:

Case #1

Passwords stored in plain text

Result

All passwords compromised

Case #2

Passwords hashed with MD5 algorithm

Result

Hacker can crack some of the passwords using rainbow tables (a pre-computed table of hash→password; weaker hashing algorithms are susceptible to these)

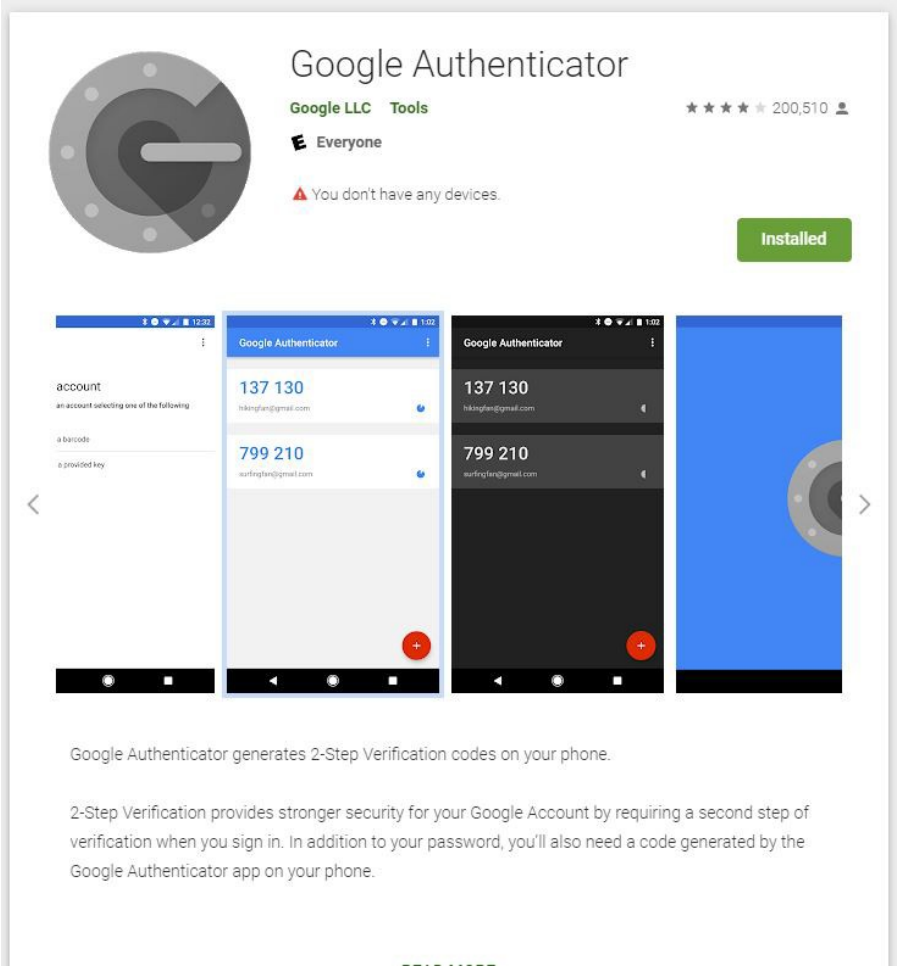
Case #3

Passwords hashed with BCrypt

Result

It is unlikely any passwords will be cracked

BCrypt is a hashing function that derives its name from two developments: the "B" comes from Blowfish Cipher, a symmetric-key block cipher developed in 1993 by Bruce Schneier, designed as a general purpose and open source encryption algorithm. "Crypt" is the name of the default hashing function that shipped with Unix OSs.



2FA

PII and Financial Data

When we store personally identifiable information (PII) on a user, we need to ensure that such storage is legal in the countries we are operating in, and that we are following any applicable laws for PII storage in those countries. Beyond that, we want to ensure that in the case of a database breach or server compromise, the PII is not exposed in a format that makes it easily abusable. Similar rules to PII apply to financial data, such as credit card numbers (also included under PII laws in some countries).
A smaller company might find that rather than storing PII and financial details on its own, a more effective strategy could be to outsource the storage of such data to a compliant business that specializes in data storage of that type.

Customers may be relying on insecure functionality, hence causing you to build secure equivalent functionality and provide them with a migration plan so that downtime is not encountered.

Deep architecture-level security flaws may require rewriting a significant number of modules, in addition to the insecure module.

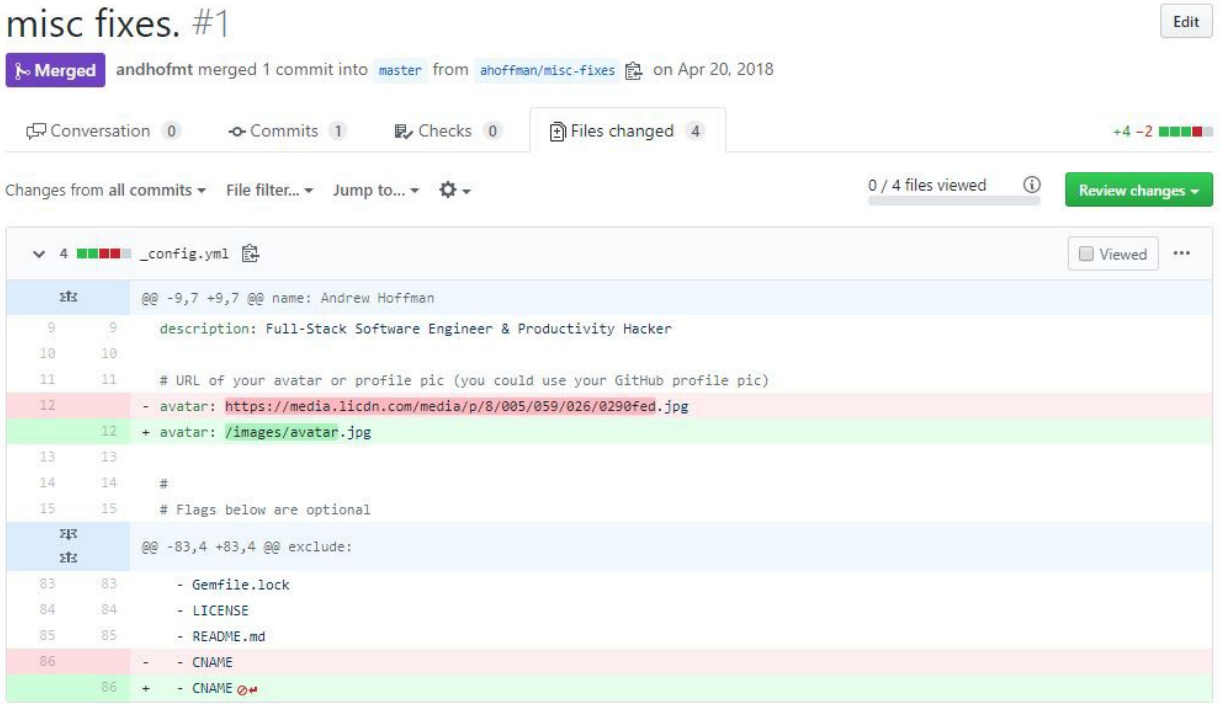
The security flaw may have been exploited, costing the business actual money in addition to engineering time.

The security flaw may be published, bringing bad PR against the affected web application, costing the business in lost engagements and customers who will choose to leave.

Reviewing Code For Security

This has two benefits. The first and most obvious benefit is that of security, but having an additional reviewer who typically is viewing the code from outside the immediate development team has its own merits as well. This provides the developer with an unbiased pair of eyes that may catch otherwise unknown bugs and architecture flaws.

As such, the code security review phase is vital for both application functionality as well as application security. Code security reviews should be implemented as an additional step in organizations that only have functional reviews. Doing so will dramatically reduce the number of high-impact security bugs that would otherwise be released into a production environment.



How to Start a Code Review

A first step in reviewing code for security is to pull the branch in question down to a local development machine.

Here is a common local review flow that can be done from the terminal:

Check out master with git checkout master.

Fetch and merge the latest master with git pull origin master.

Check out the feature branch with git checkout <username>/feature.

Run a diff against the master with git diff origin/master...

The git diff command should return two things:

A list of files that differ on master and the current branch

A list of changes in those files between master and the current branch

A code functionality review checks code to ensure it meets a feature spec and does not contain usability bugs. A code security review checks for common vulnerabilities such as XSS, CSRF, injection, and so on, but more importantly checks for logic-level vulnerabilities that require deep context into the purpose of the code and cannot be easily found by automated tools or scanners.

In order to find vulnerabilities that arise from logic bugs, we need to first have context in regard to the goal of the feature. This means we need to understand the users of the feature, the functionality of the feature, and the business impact of the feature.

Here we run into some differences in what we have primarily discussed throughout the book when we talk about vulnerabilities. Most of the vulnerabilities we have investigated are common archetypes of well-known vulnerabilities. But it is just as possible that an application with a very specific use case has vulnerabilities that cannot be listed in a book designed for general education on software security.

Consider the following context regarding a new social media feature to be integrated into MegaBank—MegaChat:

We are building a social media portal that allows registered users to apply for membership.

Membership is approved by moderators based on a review of the user’s activity prior to membership.

Users have limited functionality, but when upgraded to members they have increased functionality.

Moderators are automatically given member functionality, plus additional moderation capabilities.

Unlike users, who can only post text media, members can upload games, videos, and artwork.

We gate the membership because hosting this type of media is expensive, and we wish to reduce the amount of low-quality content as well as protect ourselves from bot accounts and freeloaders who are only looking to host their content.

From this we can gather:

Users and roles
The users are MegaBank customers.

The users are split into three roles: user (default), member, and moderator.

Each user role has different permissions and functionality.

Feature functionality
Users, members, and moderators can post text.

Members and moderators can post video, games, and images.

Moderators can use moderation features, including upgrading users to members.

Business impact
The cost of hosting video, games, and images is high.

Membership comes at the risk of freeloading (storage/bandwidth cost) and bots (storage/bandwidth cost).

An archetypical vulnerability would be an XSS in a post made by a user. A custom logic vulnerability would be a specific API endpoint that is coding improperly and allows a user to send up a payload with isMember: true in order to post videos, although they have not been granted the member functionality by a moderator.

The code review is where we will look for archetypal vulnerabilities, but also try to find custom logic vulnerabilities that require deep application context.

An effective way of determining what code to review in a security review of a web application is as follows:

Evaluate the client-side code to gain understanding of the business logic and understand what functionality users will be capable of using.

Using knowledge gained from the client review, begin evaluating the API layer, in particular, the APIs you found via the client review. In doing this, you should be able to get a good understanding of what dependencies the API layer relies on to function.

Trace the dependencies in the API layer, carefully reviewing databases, helper libraries, logging functions, etc. In doing this, you will get close to having covered the majority of user-facing functionality.

Using the knowledge of the structure of the client-linked APIs, attempt to find any public-facing APIs that may be unintentionally exposed or intended for future feature releases. Review these as you find them.

Continue on throughout the remainder of the codebase. This should actually be pretty easy because you will already be familiar with the codebase having read through it in an organic method versus trying to brute force an understanding of the application architecture.

Vulnerability Management

Part of any good secure software development life cycle (SSDL) process is a well-defined pipeline for obtaining, triaging, and resolving vulnerabilities found in a web application.

Reproducing Vulnarabilites

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.”

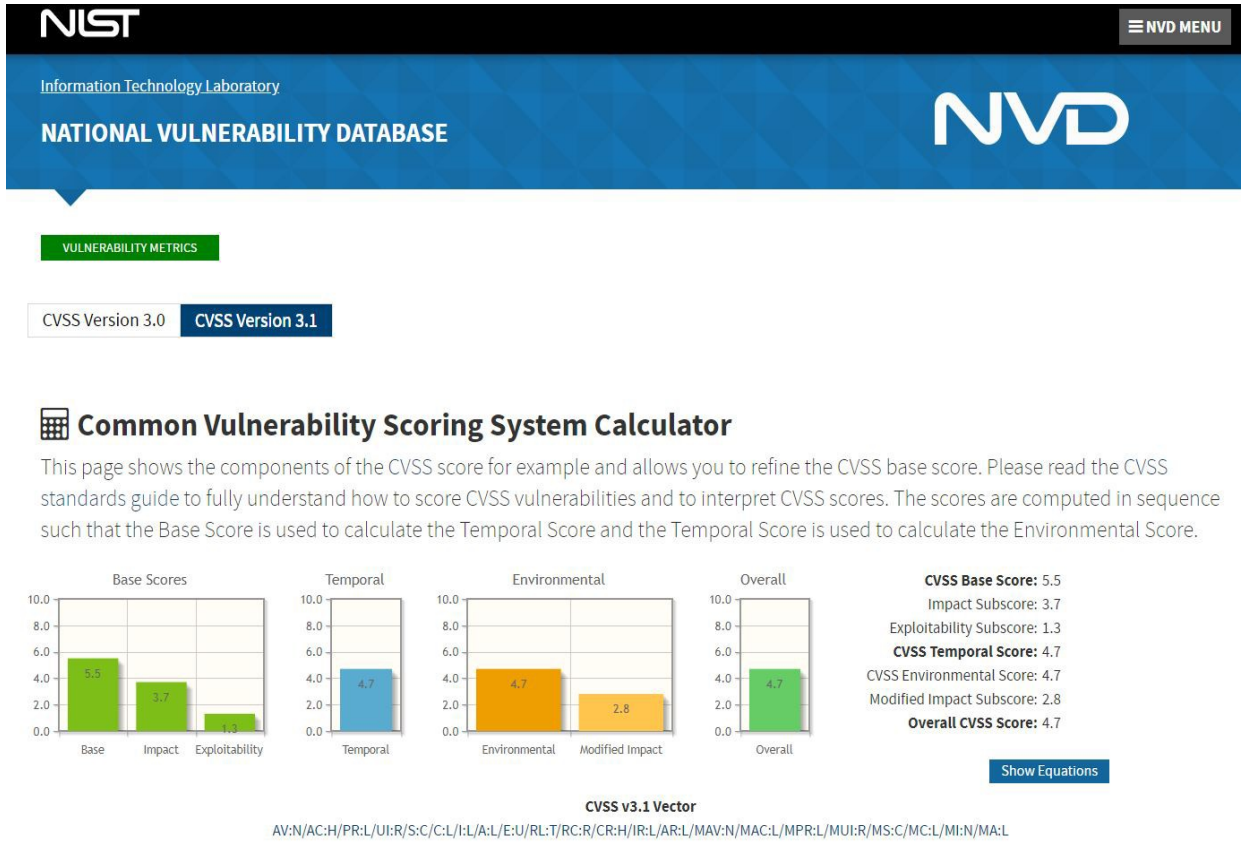
To reproduce vulnerabilities efficiently, you need to establish a staging environment that mimics your production environment as closely as possible. Because setting up a staging environment can be difficult, the process should be fully automated.

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.

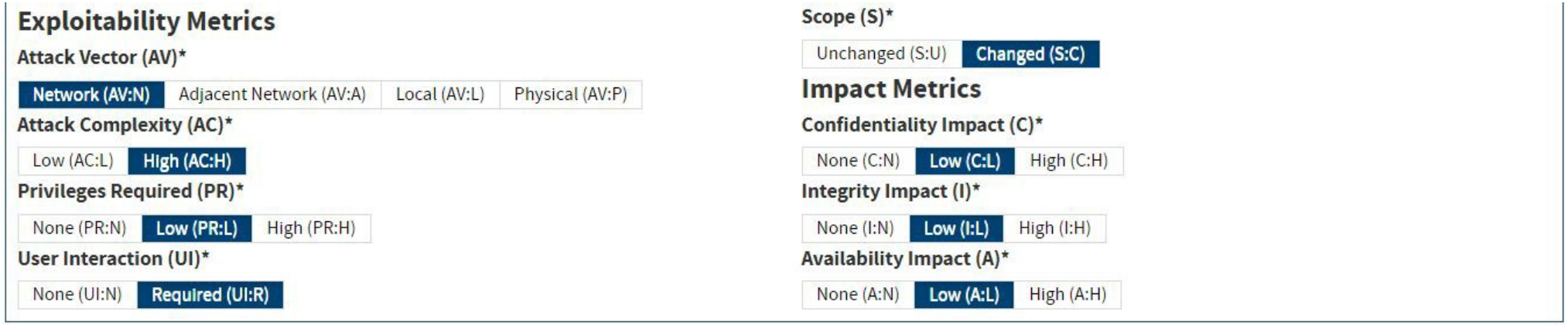
Finally, reproducing vulnerabilities gives you deeper insight as to what could have caused the vulnerability in your codebase and is an essential first step for resolving the vulnerability. You should reproduce right away and log the results of your reproduction.

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 followed 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.



The CVSS system is on version 3.1 at the time of this writing, which 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

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, while 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,” while 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” (guest user), “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 file system 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 mapped to other vulnerability scoring frameworks that don’t use numerical scoring quite easily:

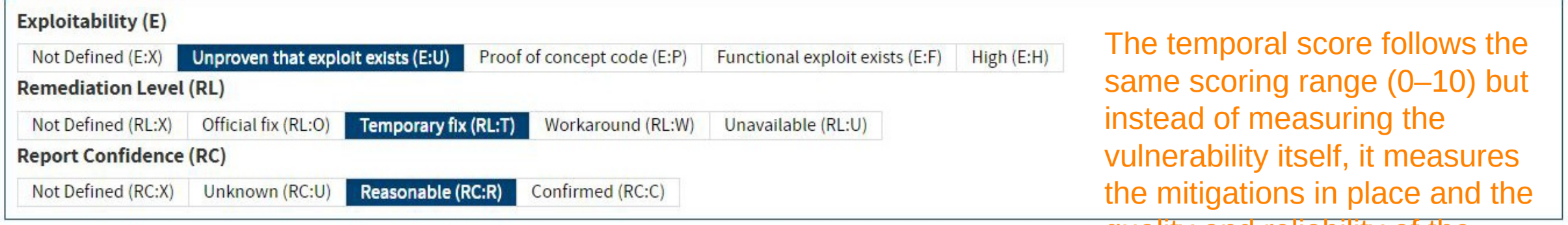
0.1–4: Low severity

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.



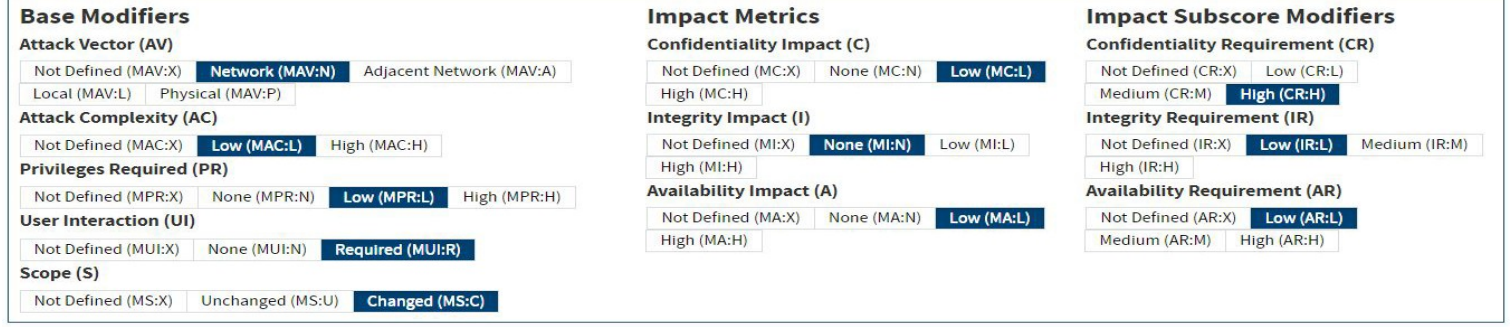
The temporal score follows the same scoring range (0–10) but instead of measuring the vulnerability itself, it measures the mitigations in place and the quality and reliability of the vulnerability report.

CVSS Temporal Score 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 “Fix 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.



CVSS Enviornmental Score The environmental scoring algorithm takes all of the base score inputs, but scores them in addition to three requirements that detail the importance of confidentiality, 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, while 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.

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 vulnerabilities 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.