How to reduce the time it takes to patch vulnerabilities?

Author: Vedang Joshi
Published: October 12, 2023 · 5 min read

"Patch for CVE-2023-12345 that affected py-cool library from version v1.3 to v2.6 was released - v2.7" mentioned the release note of py-cool library on GitHub.

This statement can have two distinct reactions by all the people using py-cool. For a few teams, it happens to be a sigh of relief, as they can now see their CI colored green, while for a few other teams, it could just be a blip in the ocean - one more patch, one more vulnerability. What differentiates both these teams? which one is the better state to be in? and how can we go from being the second team to the first? Let's explore this.

A lot of what I am going to share in this article is just the elaboration of the following 2 words: PROACTIVE MAINTAINCE. Yes, that's it. If you already understand where we are headed, you can go back and sip your sangria.

Using third-party libraries is essential to software development. The important point is to always consider the tradeoff between the functionality the library offers vs. its maintenance overhead. This is our first key point:

- Choose Dependencies Wisely:
  - **Verify the Source**: Ensure that the source of your dependencies is reputable and reliable. Trustworthy sources are less likely to introduce vulnerabilities or abandon their projects.
  - **Verify Maintainability**: Assess the project's maintainability by checking the frequency of updates and the responsiveness of maintainers to issues and pull requests.
  - **Consider Overhead**: Every dependency adds complexity and potential maintenance overhead. Whenever possible, avoid dependencies that aren't critical to your project's functionality.

Once you have decided to go ahead and use the library. You need to constantly listen to what the maintainers have to say.

- Communicate with the Community
  - **Engage with the Community**: Dependencies with active communities are often more reliable. Interact with community members, ask questions, and provide feedback.
  - **Subscribe to Releases**: Stay informed about updates by subscribing to dependency releases. Regular updates can include critical security fixes and new features.
  - **Changelogs**: Pay attention to changelogs to understand what changes have been made in each update. This information can help you assess the impact of updates on your project.
  - **Direct Communication**: Don't hesitate to communicate directly with dependency maintainers. Building a rapport can be invaluable when troubleshooting issues or requesting assistance.

Now that you know who is writing the code you use. You need to stay updated with the new features and patches they are releasing.

- Plan Regular Dependency Reviews
  - **Set a Schedule**: Establish a routine for dependency reviews, aiming to conduct them at least every two weeks. Consistency is key to managing dependencies effectively.
  - **Update Direct Dependencies**: Prioritize updating direct dependencies as they directly impact your project's functionality. Pay attention to deprecation warnings or compatibility issues.
  - **Address Backlogs**: Don't let dependency updates accumulate. Tackling a backlog can be time-consuming and compounding, leading to increased risk.

While these are libraries that you introduced intentionally, there are tons of libraries working in the background. Docker, as we all know, does a great job and packaging these all the way to the kernel level. This ability of docker can be of great help, it allows us to use just what is required.

- Reduce Clutter in Docker Images
  - **Minimal Golden Images**: Start with minimal golden images as the base for your containers. These lightweight images minimize the attack surface.
  - **Trim Dependencies**: Modify your golden images to include only the dependencies required by your application. Reducing the number of dependencies can enhance security and resource efficiency. For example, use `--no-install-recommends` with `apt-get` in Debian-based images to avoid installing unnecessary packages.

While you do your best, there would still be a good chance that you have to keep support for outdated software. For example - You may be supporting a new version of a database but other microservices are still on the old versions, thus you can't update. The best thing to do here is to mark all the code that would be deprecated and assign alternatives to each.

- Separate Outdated Code from Your Core
  - **Modular Approach**: Maintain outdated or deprecated code as separate modules or packages. This allows you to update or discard them more easily when dependencies require changes.

Finally, testing is an investment you need to make

- End-to-End Tests are Your Best Friends
  - **Test Coverage**: Develop E2E tests that cover critical aspects of your application. These tests act as a safety net to catch issues caused by dependency updates.
  - **Automate Testing**: Automate your E2E tests and integrate them into your continuous integration (CI) pipeline. Automated testing ensures that tests are consistently executed.

In conclusion, We need to understand that software engineering is not a technical problem, it rather is a socio-technical problem. As developers, we can't afford to overlook the social aspect. People leaving the organization, new people joining, dependencies being maintained, dependencies being archived - These are not rare events, we need to make our products resilient to them.

The above steps are elementary and an initiation of good processes. Adhering to them can surely help us obviate vulnerabilities and reduce the time to patch them. Feel free to share what you think about the same, and as always

Thanks for reading!

Security Python DevOps

© 2025 Github · Powered by Zola · Theme Zola-Paper