Secure Defaults: Cloud Native 8

Author: Pushkar Joglekar
Contributors (alphabetical order): Andres Vega, Emily Fox, Faraz Angabini, Robert Clark, Robert Van Voorhees
Closed for public comment on Oct 31, 2021

1. Make security a design requirement

Rationale: Security should be the fourth pillar in designing a cloud native system in conjunction with consistency, availability and partition tolerance. This approach provides cost-savings in time and effort when compared to moving a system from an insecure state to secure state as an afterthought. This approach is known as *secure by design*.

How: Utilize threat modeling, red/purple/blue team reports of prototypes and existing systems, as well as amplified feedback loops to generate design requirements to mitigate potential attacks. These should be integrated in the form of corresponding tests into the existing software development process and pipeline.

Example: Docker runtime ships with built-in seccomp policy, blocks unneeded linux capabilities and runs on a local unix socket by default which reduces the attack surface area at kernel and network layer.

2. Applying secure configuration has the best user experience

Rationale: Secure defaults should be part of the initial configuration in setup and are transparent to the operator. It must be the easiest option to get things done and not degrade system behavior significantly.

How: Review the existing end user facing guides and APIs to determine where the security configurations exist and ask users of the project these questions: Are the instructions clear? Is the API documentation explicit? Does it require the reader or user to be an expert or make guesses? Does it increase the cognitive load of the reader or user to apply successfully? If applying secure configuration takes the same or longer than the traditional setup, documentation is missing, or is difficult to follow, revisit the configuration to reduce time by 20% with each version release eventually making it part of project setup and installation.

Example: Running Kubernetes with Mutual TLS enabled for communication between control plane components, has the best user experience because of built-in certificate management capability.

3. Selecting insecure configuration is a conscious decision

Rationale: Sometimes, insecure configuration is applied to drive business outcomes with increased risk. In that case, this selection should be easy to understand and would need explicit choice from the user of the system to both understand the risk and consciously accept it.

How: Alert the operator when running in an insecure mode and ensure accompanying documentation clearly captures those. List security implications of insecure configurations with links to the preferred secure configuration guidance.

Example: Running a pod as *privileged* in Kubernetes, needs an explicit addition of privileged: true in the security context of a pod specification thus making the user choose to run a pod as privileged.

4. Transition from insecure to secure state is possible

Rationale: When security is not considered as a design requirement and the project gains wider adoption, moving to more secure defaults may get traction and attention of the maintainers. Suddenly switching to secure defaults can lead to backwards compatibility issues. Such a move, if gradual, simple and reversible can give users confidence to move to these secure defaults.

How: Engage with end users after each release to learn about their experience in adopting and embracing secure defaults. Dedicate engineering time each sprint to reduce any difficulty end users experience through streamlining or automating the configurations in upgrades and migrations — the goal is to balance backwards compatibility, user experience, and new feature development.

Example: Seccomp Filter support in Kubernetes with feature gates and ability to audit for syscall failures gives users a way to transition to this secure default one cluster at a time. In an event this transition fails, it is possible to revert the cluster and namespace to an earlier default state.

5. Secure defaults are inherited

Rationale: Secure defaults of an underlying system can be inherited by a system that runs on top of that system. This allows higher level abstractions with loosely coupled systems that saves redundant efforts across the different layers.

How: Provide a shared responsibility model focused on secure defaults the underlying system can provide with links to guidance, instruction, and settings to ensure that they may be inherited as part of defense in depth. Inherited control should document where hybrid controls become the responsibility of the consumer.

Example: Using TLS protocol for inter-service communication, allows systems that utilize TLS to inherit the security properties of TLS protocol and its implementation without worrying about secure key exchange, in-transit data protection and two party authentication.

6. Exception lists have first class support

Rationale: Secure defaults can be too restrictive for certain workflows. In those cases, exceptions should be allowed, logged and tracked by a policy engine. Proper implementation of exception lists should result in only those privileges being granted that are needed.

How: Exceptions lists are treated as an essential feature and considered as part of the security design of the software. When working on the design requirements, establish a user story for exception lists with equal consideration to other requirements.

Example: Policy enforcement admission controllers like Pod Security Admission (PSP replacement) have built-in support to add a list of namespaces where the pod security policies are not enforced to allow running privileged workloads in those namespaces.

7. Secure defaults protect against pervasive vulnerability exploits

Rationale: Secure defaults should create value for the user of a system by protecting them from pervasive and common vulnerability exploits that have an insecure precondition for a successful exploit.

How: As part of the pipeline, integrate tooling and code reviews that identify the following common opportunities: remote code execution (RCE), arbitrary code execution, arbitrary file read, path traversal, exposed credentials, and privilege escalation. Actively mitigate and correct any findings prior to release.

Example: Running as non-root user or with SELinux enforcing mode enabled has allowed protection against several vulnerabilities detected in container runtimes and Kubernetes.

8. Security limitations of a system are explainable

Rationale: In spite of the best intentions of a system designer, some security controls can not be applied to a system without fundamentally changing the nature of the system. In those scenarios, these limitations should be documented and discoverable, with clear, easy-to-understand reasons behind these tradeoffs and alternative options listed.

How: As part of the design review and code reviews, annotate areas where a secure option negatively alters the behavior of the software. Document this decision to exclude the secure option within the code and design documents and copy this information into a more end-user friendly ‘Design considerations’ guide.

Example: Container runtimes, do not provide hypervisor isolation since containers are not Virtual Machines. In those situations, possible alternatives like kata containers that provide hardware virtualization while allowing to manage workloads like a container are documented.

Note: Please keep examples limited to open source projects, that are related to cloud native ecosystem

Notes

- Message sent to CNCF TAG Security: https://lists.cncf.io/g/cncf-tag-security/message/71
- This guideline format and intent is influenced from: PEP 8 -- Style Guide for Python Code
- Featured in CNCF blog: https://www.cncf.io/blog/2021/10/12/cloud-native-security-microsurvey-more-than-80-of-organizations-want-to-build-modern-security-systems-with-open-source-software/
- Github issue tracker: https://github.com/cncf/tag-security/issues/734