WIZ Research

PEACH

A Tenant Isolation Framework for Cloud Applications

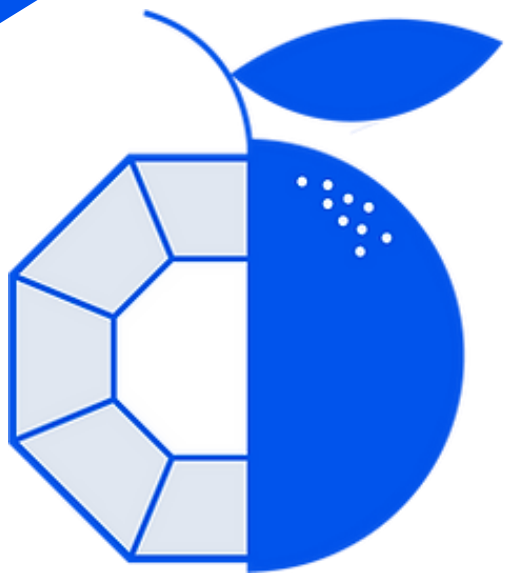


Table of Contents

Executive Summary

Isolation between different tenants in multi-tenant cloud applications is a foundational principle of modern cloud security. However, building for tenant isolation and balancing it against operational considerations is a complex endeavor, and several cases have been discovered in recent years where **design and implementation mistakes enabled unauthorized cross-tenant access**. These flaws included both bugs in customer-facing interfaces as well as insufficient hardening of the security boundaries meant to separate tenants from one another.

We therefore introduce the **PEACH framework** for managing the risk of isolation escape in cloud SaaS and PaaS applications, composed of two parts:

- **Modelling tenant isolation** – We propose the inclusion of an isolation review process as part of a broader threat modelling effort concerning a multi-tenant application. The resulting model can serve as the basis for promoting greater transparency from application vendors in regard to isolation assurance, by allowing them to disclose an abstract description of their tenant isolation measures without having to reveal sensitive implementation details. This process involves analyzing the risks associated with the application's customer-facing interfaces, by determining:
    1. The complexity of the interface as a predictor of vulnerability
    2. Whether the interface is shared or duplicated per tenant
    3. The security boundaries in place
    4. How strongly these have been implemented according to five parameters, abbreviated as **P.E.A.C.H.** (hardening of Privilege, Encryption, Authentication and Connectivity, and ensuring proper Hygiene)
- **Improving tenant isolation** – Once the risk of isolation escape has been properly assessed, vendors can take three preventive approaches if they so choose, all while accounting for operational context such as budget constraints, compliance requirements, and expected use-case characteristics of the service:
    1. Reducing interface complexity
    2. Improving tenant separation
    3. Increasing interface duplication

PEACH is based on the **lessons we've learned in the course of our cloud vulnerability research**, and we've been using it internally at Wiz as part of our own product design review process. We would be thrilled to receive your feedback so we can improve the framework and make it as useful as possible for cloud application developers.

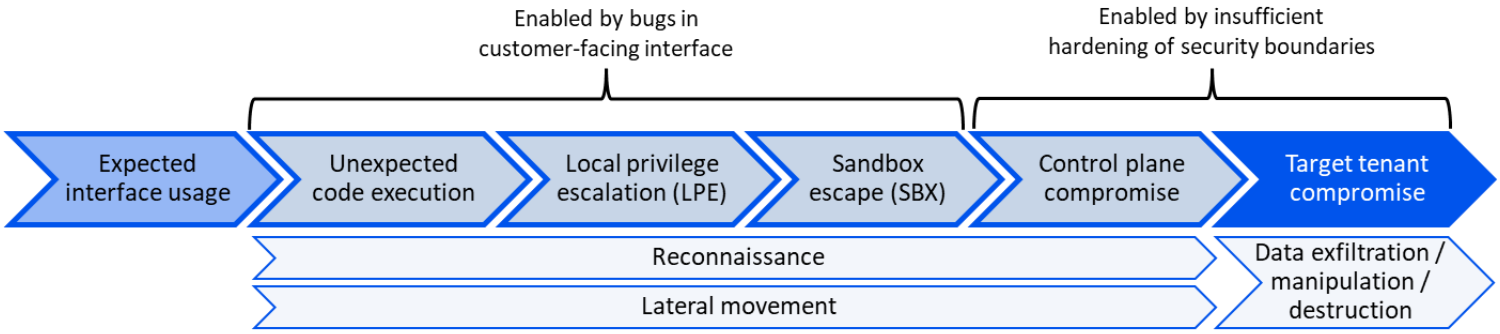
Acknowledgements

We would like to extend our gratitude to Christophe Parisel (Senior Cloud Security Architect, Société Générale), Cfir Cohen (Staff Software Engineer, Google), Kat Traxler (Principal Security Researcher, VectraAI), Srinath Kuruvadi (Head of Cloud Security, Netflix), Joseph Kjar (Senior Cloud Security Engineer, Netflix), Mike Kuhn (Managing Principal, Coalfire), Daniel Pittner (Software Architect, IBM Cloud), and Adam Callis (Information Security Architect, Cisco) for sharing constructive input throughout the development of this framework.

We would also like to thank AWS for their review of our whitepaper and the valuable feedback they provided. We highly appreciate their willingness to help us identify tenant isolation best practices and their commitment to improving security transparency for cloud customers.

Introduction

Isolation between different tenants in multi-tenant cloud applications is a foundational principle of modern cloud security: while hardware may be shared among multiple customers, each customer's data is expected to be kept at least as private and secure as if it were located on-premises. However, **designing for tenant isolation poses a significant engineering challenge**. In recent years, several cross-tenant/isolation escape vulnerabilities have been discovered in various multi-tenant cloud services which could have potentially enabled malicious tenants to access data belonging to other customers[1]. We can identify a common root cause in all these cases in the form of bugs in customer-facing interfaces (such as an insecure API permitting tenants to execute code in unexpected ways or escalate their local privileges within their own environment), compounded by **i**nsufficient hardening of security boundaries (such as a virtual machine with an overly permissive firewall).



Typical vulnerability-enabled cross-tenant attack sequence
(depending on circumstances, some intermediate steps can be skipped)

In light of these security implications and based on the lessons we've learned over the course of our own cloud vulnerability research – which we believe are applicable to both PaaS and SaaS – we propose herein a **practical framework for reasoning about tenant isolation in cloud applications**.

Our goals are to pave the way for eventually setting **common industry standards** for tenant isolation, and to improve vendor transparency with regard to isolation assurance. We believe that widespread adoption of this framework will empower cloud customers and security researchers by clarifying what questions they should be asking vendors about tenant isolation, and by making it easier to compare isolation (and resilience to cross-tenant vulnerabilities) between different cloud service offerings.

While isolation cannot prevent interface vulnerabilities or reduce their likelihood, it can introduce tolerance for them into the system by **minimizing their "blast radius".** For example, a directory traversal vulnerability in a customer-facing interface could be mitigated by duplicating the affected component per tenant, so that each customer would only be able to interact with their own dedicated endpoint. Thus, the relevant bug would still exist, but it could only be abused by a single tenant to traverse their own data, thus nullifying the impact. In other words, an attacker could no longer use such a vulnerability to gain initial cross-tenant access to a target environment. The vulnerability would still be problematic, but arguably less so (for example, an attacker with existing access to a target network could theoretically exploit such a vulnerability to create a backdoor that might be undetectable by the target tenant).

In other words, isolation can be used to mitigate interface vulnerabilities by way of compensating controls. Thus, in our view, **isolation is complementary to interface security in multi-tenant cloud services.**

[1] e.g., ExtraReplica, SuperGlue, AzureScape, CloudFormation credential leak, AutoWarp, ChaosDB, SynLapse, IDOR in AWS Lake Formation, AttachMe and Hell's Keychain

This framework attempts to build on prior work by cloud service providers[2] and government agencies[3] on the subject of tenant isolation. It is not meant to serve as an alternative to existing prescriptive cloud architecture guidance on security[4], and neither should it be viewed as a replacement for prevailing general threat modelling frameworks[5]. Instead, this framework serves to specifically model the threat of unauthorized cross-tenant access and should therefore be used as part of a broader threat modelling process. We have chosen to focus our attention on **isolating complex customer-facing interfaces** in an attempt to dismantle the larger problem of a full architecture analysis. Moreover, this "interface-oriented" approach allows us to take on the perspective of a would-be attacker observing the environment from the outside in, while "following the untrusted data", so to speak.

Having said that, cloud service design is complex, and security is only one of many design considerations such as functionality and cost. Therefore, each of the principles outlined in this framework should be considered alongside other factors, and it is ultimately up to the vendor to decide what steps to take on the basis of the conclusions of the review process. In other words, this framework is primarily intended to serve as a **descriptive (non-prescriptive) foundation for risk analysis** by surfacing potential weak points in isolation and suggesting possible solutions.

Part One – Modeling Tenant Isolation

1. External Interfaces

We define a given cloud service's **external interfaces** as the set of "closest" tenant-facing applications that process user input (in other words, mechanisms that expose attack surface by which a customer can change the state of the service backend in some way, such as a database client).

Because they allow for the ingestion of **user-controlled and untrusted data** (in the form of strings, source code, binaries, etc.), each interface must be properly secured to prevent abuse by a malicious tenant. An effective method of doing so – and the subject of this framework – involves **isolating** at least some of the interface's components.

A **tenant-isolated component** is defined here as one that has been shifted outside the vendor's **trust boundary**, then **duplicated** per tenant (i.e., made to be dedicated rather than shared; whether logically, virtually, or physically), and **separated** from all others of its kind via **security boundaries** (such as virtualization or network segmentation)[6]. We can then define **isolation level** as measured by the number and quality of such boundaries that contribute to this separation, forming an end-to-end **"isolation scheme"**.

We can generally define **interface complexity** as measured by the degree of control granted to would-be attackers[7] (such as in the case of arbitrary code execution environments found in some PaaS offerings). We offer the following

---

[2] e.g., AWS's SaaS Tenant Isolation Strategies, Azure's guidance for secure isolation, IBM's article on handling multiple tenants in their public cloud, and Oracle's "Isolate Resources and Control Access" section in their Best Practices Framework

[3] Such as guidance by the UK's National Cyber Security Centre (NCSC); in particular, "Technically enforced separation in the cloud", *Cloud security guidance*, ncsc.gov.uk and the principle of "Separation between customers"

[4] e.g., the Security pillar of the AWS Well-Architected Framework, the Security, privacy, and compliance category of Google's Cloud Architecture Framework, the Secure phase of Microsoft's Cloud Adoption Framework for Azure, and the Best practices framework for Oracle Cloud Infrastructure

[5] Such as PASTA, Microsoft's STRIDE and Carnegie Melon's Hybrid Threat Modeling Method; see also AWS's "How to approach threat modeling", OWASP's Threat Modeling Cheat Sheet and the principles outlined in the Threat Modeling Manifesto

[6] Another definition for this term can be found in Fehling C., Leymann F., Retter R., Schupeck W., & Arbitter P. (Springer, 2014). "Cloud Computing Patterns" (*"A component shared between tenants avoids influences between tenants regarding assured performance, available storage capacity, and accessibility of functionality and data."*), but our proposed definition expands the scope of isolation to include both shared and non-shared underlying IaaS workloads (such as a shared virtual machine hosting a multi-tenant database; and a collection of single-tenant virtual machines each running a dedicated database, respectively).

[7] More specifically, we can postulate that a given software component's degree of Turing-completeness determines its potential for abuse by a malicious user aiming to escalate their privileges in the environment. For a more precise definition of interface complexity, we refer the reader to prior work on this subject, such as Kumari, U. & Upadhyaya, S. (2011). "An Interface Complexity Measure for Component-based Software Systems"

examples of **interface types**[8] and their estimated complexity levels[9] (the reader may however choose to use a different system of estimation or extrapolate from these examples to measure other cases not listed here).

In complex interfaces, legitimate code execution permissions could be abused to escalate privileges or escape a sandbox (as we have demonstrated in our own prior work on this subject[10]). In this framework, we assume that **complexity correlates positively with likelihood of vulnerability**[11], thus justifying stronger isolation of the interface to some degree. In other words, tenant isolation serves to compensate for interface complexity by containing it and the risks associated with it.

Interface type	Typical input (example)	Typical process	Complexity level
Arbitrary code execution environment	Arbitrary	Execution	High
Database client	SQL query	Database operation	High
Arbitrary file scanner	Arbitrary	Parsing	Medium
Binary data parsing	Protobuf	Parsing	Medium
Web crawler	JavaScript	Rendering	Medium
Port scanner	Metadata	Parsing	Low
Reverse proxy	Arbitrary	Proxy	Low
Queue message upload	Arbitrary	Proxy	Low
Data entry form	String	Parsing	Low
Bucket file upload	Arbitrary	Storage	Low

Examples of interface types and their typical inputs, typical processes, and estimated complexity levels

2. Security Boundaries

Choice of boundary technologies used to facilitate tenant isolation (i.e., the IaaS or PaaS offerings that form the basic building blocks of an application's architecture) should generally be limited to **well-known, mature mechanisms** that have been industry-vetted to prevent code execution from leading to isolation escape[12].

An individual security boundary's effective isolation level is dependent on both its **type** and **hardening** (as defined in this section and the following one). While type determines the boundary's potential (or maximum isolation level), hardening determines the realization of this potential. In other words, if implemented permissively, an otherwise strong boundary becomes weaker, and conversely, a strictly implemented weak boundary cannot be made stronger than dictated by its potential.

Combining multiple independent boundaries (i.e., of different types) increases isolation level, and also serves to compensate for boundaries that have been permissively configured, whether by mistake or by design (such as to allow multiple virtual machines or database instances to be able to freely communicate with one another). Conversely, unmitigated **"weak links"** in design or implementation decrease isolation level (such as network connectivity between instances belonging to different tenants).

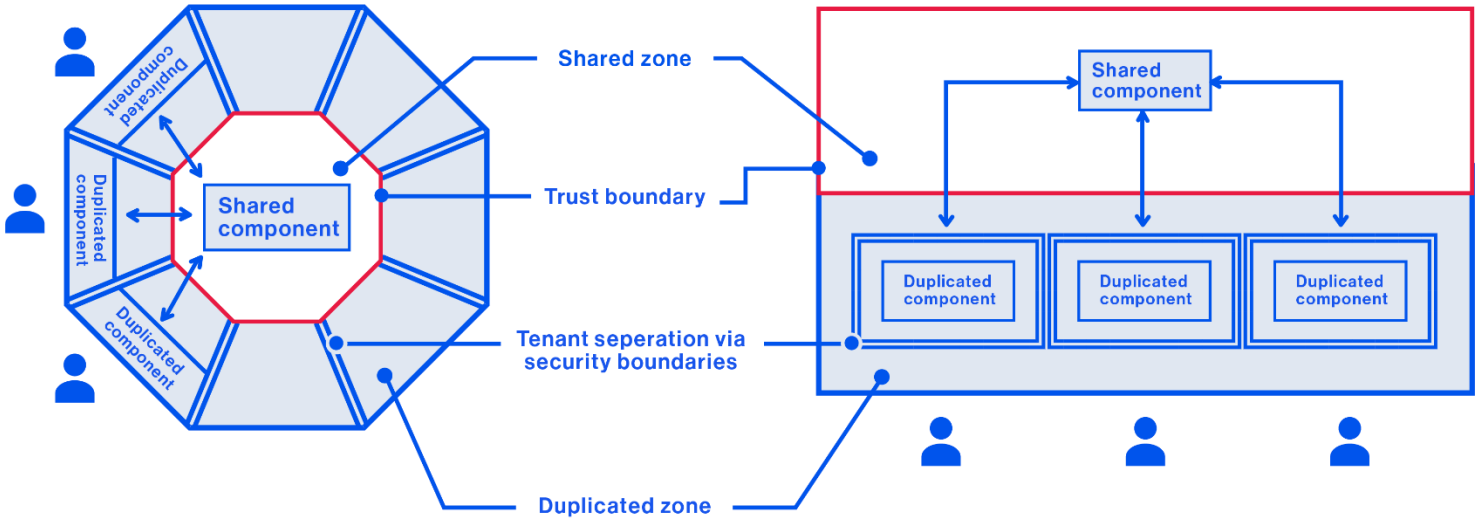
[8] Each of these interfaces may be expressed as a GUI, CLI, REST API (accepting structured data such as JSON), etc.

[9] Note that interfaces originally designed for single-tenant use (e.g., PostgreSQL database clients) and adapted for use in cloud environments should be assumed to be of high complexity unless convincingly proven otherwise. Such software might contain vulnerabilities that would normally be locally scoped and therefore of lower concern, such as host-level arbitrary code execution and local privilege escalation, but the same vulnerabilities would have a much larger blast radius in multi-tenant environments, as they could enable an attacker to access other customers' data (see also "The cloud has an isolation problem: PostgreSQL vulnerabilities affect multiple cloud vendors", Wiz).

[10] See *"The cloud has an isolation problem: PostgreSQL vulnerabilities affect multiple cloud vendors"*, Wiz

[11] See the methodology used by the Cyber Independent Testing Lab (CITL) as well as Alenezi, M. & Zarour, M. (2020). "On the Relationship between Software Complexity and Security"

[12] Precisely defining the industry vetting of boundary technologies is outside the scope of this paper, but this could include formal testing and certification processes. For example, Cook, B., Khazem, K., Kroening, D., Tasiran, S., Tautschnig, M., Tuttle, M.R. (2018). "Model Checking Boot Code from AWS Data Centers" and "Using the Kani Rust Verifier on a Firecracker Example", Kani.



Simplified representation of a multi-tenant cloud service environment, illustrating interface isolation and potential lateral movement directions

The mechanisms outlined as follows are considered to be security boundary types in the context of this framework[13], though others may qualify as well[14]. We have divided these into two groups: **primary boundaries**, which can be utilized on their own (achieving varying degrees of isolation) and **secondary boundaries**, which can be utilized in conjunction with primary types to increase the isolation level.

It is important to clarify that vendors should not be discouraged of using boundary types estimated to have low isolation levels – many of these have significant operational benefits that drastically outweigh security considerations in production environments. However, vendors should invest in compensating for such drawbacks by other means (as explained in part two).

Primary Boundaries

Hardware Separation

Wherein instances belonging to different tenants run on separate physical machines (IaaS such as bare-metal servers or dedicated physical datacenters). This is the highest practical level of isolation but is not considered cost-effective at scale, nor is the added level of isolation significant in comparison to the use of industry-vetted hardware virtualization (this becomes especially apparent where hardware separation has been insufficiently hardened).

Hardware Virtualization[15]

Wherein instances belonging to different tenants run in separate virtual machines running on shared hardware[16]. This is generally considered a high level of isolation.

Containerization

Wherein instances belonging to different tenants run in separate containers[17] running on shared hardware (or nested within a shared virtual machine) via operating-system-level virtualization. Containers are not considered an especially effective security boundary on their own, and therefore this should generally be treated as a low level of isolation.

---

[13] These could also be referred to as "separation primitives"; see Pfeiffer, M., Rossberg, M., Buttgereit, S., and Schaefer, G. (2019). "Strong Tenant Separation in Cloud Computing Platforms"

[14] Note that "zero isolation" is defined as storing and processing data belonging to different tenants in an unencrypted form (or encrypted via the same key) on a shared hardware component, without any form of virtualization or segmentation.

[15] Equivalent to MITRE ATT&CK Enterprise M1048 (Application Isolation and Sandboxing)

[16] Examples include Linux KVM, Hyper-V, Nitro Hypervisor and Firecracker

[17] Examples include Docker or containerd containers as well as Drawbridge picoprocesses and V8 isolates as utilized in Cloudflare Workers (see also "Security model", *Cloudflare Docs*)

When using containerization as a security boundary, vendors should enforce permission restrictions on deployed workloads[18], via technologies such as SECCOMP[19], AppArmor[20] and gVisor[21], thereby limiting the capacity of a compromised container from affecting its host. Additional mitigations may include implementing a read-only filesystem, limiting the privileges of user accounts, disabling the root user, and minimizing the number of running highly privileged processes. Moreover, vendors should generally prefer the use of hardened operating systems[22].

Data Segmentation
Wherein data belonging to different tenants is stored and/or processed on a shared storage component (virtual or physical), but unique keys are used to encrypt and/or authorize access to each tenant's data[23]. This is generally considered a low level of isolation, as it's highly dependent on the implementation of other aspects of the architecture (such as key storage security).

Secondary Boundaries

Network Segmentation[24]
Wherein instances belonging to different tenants run on separate machines (virtual or physical) in the same network, which is segmented into single tenant VPNs[25] or subnets (logically separated via encapsulation, encryption, and firewalling[26]). This is generally considered a high level of isolation.

Identity Segmentation
Wherein roles are assigned to different tenants (whether or not customers can access them directly under normal circumstances) which are limited by deny-by-default policies, such that each tenant cannot perform any actions affecting resources belonging to any other tenant[27]. This is generally considered a high level of isolation.

3. Hardening Factors

As mentioned above, determining a given security boundary's efficacy must account for the weakest link in its design or implementation. For example, a virtual machine running on a fully patched hypervisor should be considered strongly isolated only if the virtual machine is also hardened from a network and identity perspective, as any network connectivity between hosts may theoretically allow an attacker access to other tenants' virtual machines unless guardrails are set in place to control this behavior. Note that in the context of this framework, we assume that all underlying IaaS is secure.

Thus, to reach their full potential – which is usually only necessary and/or feasible in the case of complex interfaces – security boundaries can be hardened so as to reduce the probability of weak links (whether obvious or obscure). The following measures – abbreviated as **P.E.A.C.H.** – can therefore be taken to raise the isolation level of a given interface, by methodically reducing attack surface and increasing defense-in-depth. These measures correspond to the types of weaknesses that we've identified and successfully exploited in various complex PaaS and SaaS offerings.

---

[18] Equivalent to MITRE ATT&CK Enterprise M1028 (Operating System Configuration)
[19] "Restrict a Container's Syscalls with seccomp", *Kubernetes | Docs*
[20] "AppArmor – Linux kernel security module", apparmor.net
[21] "What is gVisor", gvisor.dev
[22] Such as Google's Container-Optimized OS (COS)
[23] Examples include RLS (see also "Shipping multi-tenant SaaS using Postgres Row-Level Security", Nile) as well as Kubernetes namespaces using role-based access control (RBAC)
[24] Equivalent to MITRE ATT&CK Enterprise M1030 (Network Segmentation)
[25] Examples include Amazon VPC and Azure Virtual Network (Vnet)
[26] Such as AWS's Network ACLs
[27] Examples include AWS IAM and Azure Active Directory

In the context of this framework, we can use each of these hardening factors as parameters by which to **measure the strength of an isolation scheme:**

Privilege Hardening[28]

1. Tenants and hosts generally have minimal permissions in the service environment, thereby adhering to the principle of least privilege.
2. In particular, each tenant is not authorized to read or write to other tenants' data unless explicitly approved by the tenants involved, and each host does not have read or write access to other hosts.
3. Privileges are verified prior to execution of operations.

Encryption Hardening

1. Data owned by and related to each tenant – at-rest and in-transit – is encrypted with a key unique to that tenant, regardless of architecture.
2. Logs pertaining to each tenant's activity are encrypted by a secret shared only between the tenant and the control plane[29].

Authentication Hardening

1. Communications between each tenant and the control plane (in both directions) use authentication with a key or certificate unique to each tenant.
2. Authentication keys are validated, and self-signed keys are blocked.

Connectivity Hardening[30]

1. All inter-host connectivity is blocked by default unless explicitly approved by the tenants involved (so as to facilitate database replication, for example); hosts cannot connect to other hosts in the service environment, except those used by the control plane (i.e., a hub-and-spoke configuration).
2. Hosts do not accept incoming connection requests from other hosts unless explicitly approved by the tenants involved, except those used by the control plane (in anticipation of scenarios in which an attacker manages to overcome connectivity limitations on their own compromised host).
3. Tenants cannot arbitrarily access any external resource (both within the service environment and on the Internet) and are limited to communicating with only pre-approved resources or those explicitly approved by the tenant.

Hygiene

Unnecessary data scattered throughout the environment might serve as clues or quick wins for malicious actors, particularly those that have managed to breach one or more security boundaries, which in turn enable further reconnaissance and lateral movement. Therefore, vendors can eliminate the following types of data at the design level and also regularly scan for forgotten artifacts:

1. **Secrets** – The interface and datastore (and underlying host, in the case of virtualization or containerization) do not contain any keys or credentials that would allow authentication to other tenants' environments or decryption of other tenants' backend communications or logs.

[28] Somewhat equivalent to MITRE ATT&CK Enterprise M1052 (User Account Control)

[29] We use the term "control plane" here to mean a collection of backend components in charge of policy establishment (such as resource orchestration and provisioning of access), as opposed to the "data plane" which is made up of components in charge of policy execution (such as processing of user input in accordance with the established policies).

[30] Note that connectivity hardening does not apply to data segmentation.

2. **Software** – The instance and datastore (and underlying host) do not contain any built-in software or source code that could enable reconnaissance or lateral movement.
3. **Logs** – Each tenant's logs are hidden from and inaccessible by other tenants; logs accessible to each tenant do not contain any information pertaining to other tenants' activity.

4. Isolation Review

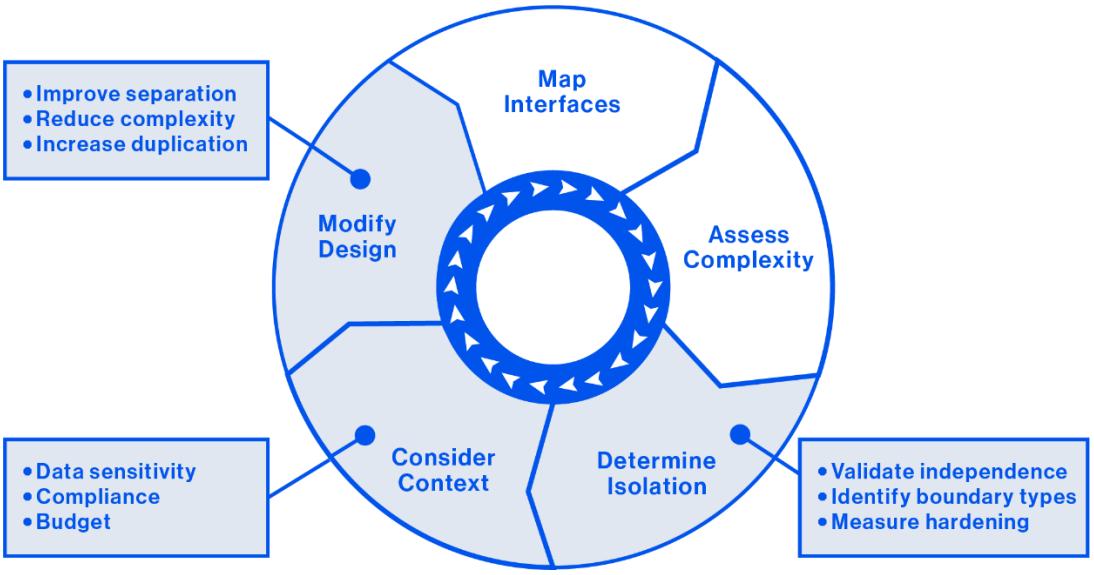
Review Procedure

To put all of the above into practice, we propose conducting an **isolation design review** by decomposing the service into a set of customer-facing interfaces, estimating the complexity and isolation levels of each interface component, and modifying the design as necessary (as outlined in part two of this document).

The process of designating complexity and isolation levels is not algorithmic or deterministic, and the results are not quantitative (as mentioned earlier, the review process is only meant to identify weak points and suggest possible solutions). Instead, the reviewer should use **qualitative assessments** based on the guidance provided in this section. We have provided our own recommended assessments of various interfaces and boundary types (see previous sections), but reviewers may choose to use a different system of estimation.

In greater detail, the review process begins by **mapping all external interfaces**, and then performing the following analysis for every individual interface:

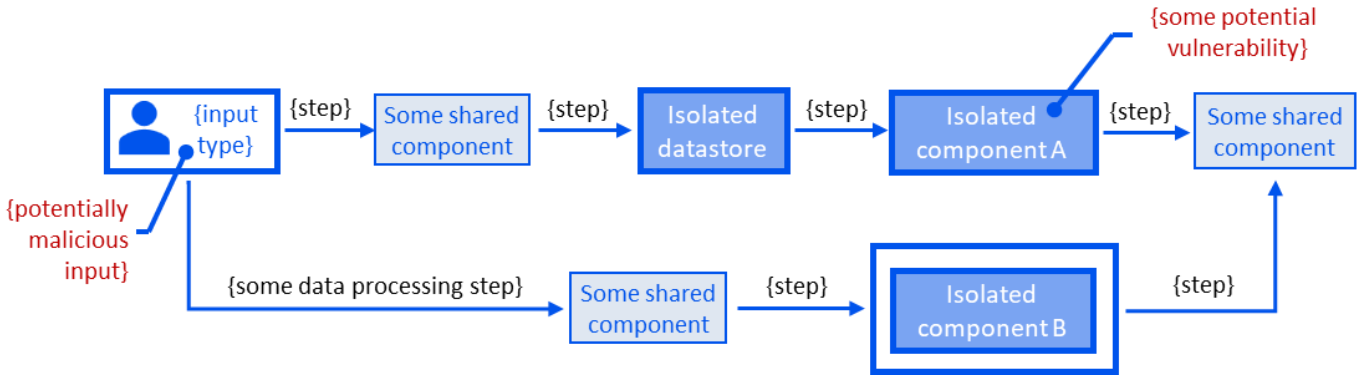
1. Identify the **type of input** (i.e., the ingested untrusted data) and the various **components** of the interface.
2. Sketch a **design diagram** of the interface, including the following elements (see suggested format below):
    a. Note the user input type, which might be malformed in order to abuse the interface.
    b. Trace the flow of data from the user input through each component that stores or processes it.
    c. Differentiate between shared and duplicated components.
    d. Add security boundaries separating each duplicated component from other tenants.
3. Assess the **complexity level** of each component. (Refer to section 1)
4. Determine the interface's **isolation level** according to the following guidelines:
    a. Which security boundary types are in place? (Refer to section 2)
    b. Are they independent of one another? (i.e., of different types)
    c. How have they been hardened? (Refer to section 3)
5. Summarize your findings in the **implementation table** (see example below).
6. Identify **potential vulnerabilities** that may arise at each stage of the dataflow, by asking:
    a. What is the component's attack surface?
    b. What sort of malicious user input might lead to its abuse?
7. Note **potential issues** stemming from the design and add them as callouts in the design diagram.
8. If isolation level is determined to be insufficient with regard to complexity and contextual factors (including compliance, data sensitivity and budget considerations), **modify the design** as necessary by reducing complexity, improving separation, and/or increasing duplication (as explained in part two).



Isolation design review procedure

Design Diagram

The isolation design diagram is a simplified data flow diagram (DFD)[31] meant to focus the review process on ensuring tenant isolation, while deprioritizing other aspects and removing interface security from scope (as mentioned earlier, we view isolation as complementary to interface security, and a means by which to compensate for interface vulnerabilities)[32].



Isolation design diagram showcasing untrusted user input processed by two distinct interfaces, with each of their various components either shared or isolated in different ways

Implementation Table

The isolation implementation table is meant to accompany the design diagram and serve as a checklist for assessing hardening factors, each of which serves to mitigate weak links in a given set of security boundaries and thereby raise the isolation level of the interface under analysis.

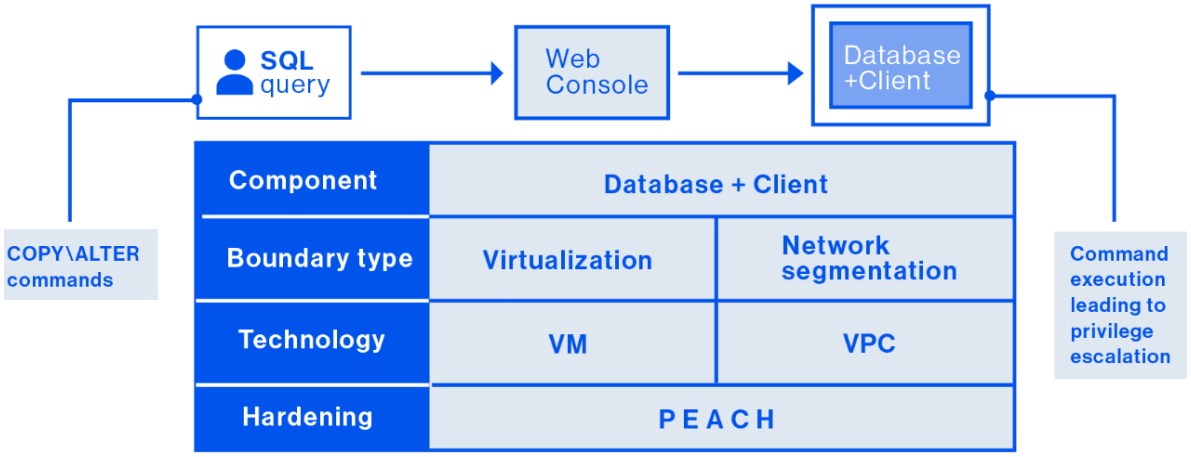
Component	datastore	component A		component B	
Boundary type	data seg.	{type}	{type}	{type}	{type}
Technology	{tech}	{tech}	{tech}	{tech}	{tech}
Hardening	PEA–H	–E–CH		PEACH	

Isolation implementation table accompanying the above design diagram, highlighting gaps in security boundary hardening (denoted by dashes in PEACH)

[31] "Data-flow diagram", Wikipedia

[32] We recommend using a whiteboard, diagramming software or threat modeling tools such as OWASP Threat Dragon.

The following is an example of an isolation design diagram and implementation table representing a highly isolated database client in a notional managed SQL database cloud service (see additional example in appendix):



Isolation design diagram and implementation table of a highly isolated database client

5. Vendor Transparency

Application vendors invest a great deal of effort and ingenuity in the security of their multi-tenant services in order to maintain the confidentiality and integrity of their users' data[33]. However, recent research has demonstrated that tenant isolation is not always properly implemented, and we have proven that this contributes to the risk of unauthorized cross-tenant access[34].

Moreover, there is a general **lack of transparency** around security controls in cloud services. On one hand, attackers would certainly be hindered by a lack of knowledge about specific architectural details or detective controls (and we should not expect vendors to make this information public). On the other hand, this supposed ignorance should not be treated as a security boundary in and of itself[35] (i.e., security by obscurity), and cloud customers are entitled to assurances about a given service's proclaimed level of tenant isolation.

Thus, we propose that vendors conduct isolation reviews in accordance with the procedure outlined above (or some other equivalent process) and **use this framework to publicly share only the most essential information about the design and implementation of their preventive controls**, without releasing full details about their service architecture. Vendors could thereby demonstrate that their architecture includes tolerances allowing for the occurrence of intrusion scenarios while limiting attackers' lateral movement options and preventing unauthorized access to customer data.

Moreover, we believe vendors should be expected to **publicly disclose significant isolation failures** discovered in their services (i.e., proven cross-tenant vulnerabilities identified in the course of internal or external research[36]) and release security advisories detailing both the vulnerabilities themselves and modifications introduced to mitigate them.

[33] For example, see Microsoft's Isolation in the Azure Public Cloud, Google's Cloud Security Overview, AWS's Security Overview of AWS Lambda and Security Overview of AWS Fargate; and MongoDB's Atlas Security whitepaper

[34] See examples in footnote 1.

[35] See the definition of open design from J. H. Saltzer and M. D. Schroeder. (1975). "The protection of information in computer systems" (*"The design should not be secret; the mechanisms should not depend on the ignorance of potential attackers."*)

[36] To this end, vendors should allow, encourage, and support external security research efforts concerning their cloud services, by operating bug bounty programs and maintaining dedicated pen-testing environments.

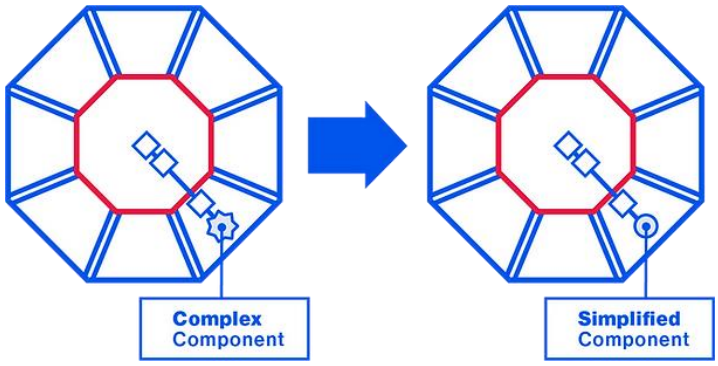
Part Two – Improving Tenant Isolation

6. Core Principle

Generally speaking, in order to improve the tenant isolation of a given cloud application, we should strive to separate trusted and untrusted data in the service environment and minimize the "surface area" between neighboring tenants, and between each tenant and the vendor control plane. We therefore propose that vendors have three primary preventive "levers" at their disposal for managing the risk of unauthorized cross-tenant access (where risk = likelihood x impact) by bringing complexity and isolation into alignment[37]: (1) **reducing complexity**, (2) **improving separation**, and (3) **increasing duplication**. We shall elaborate on each of these methods throughout this section.

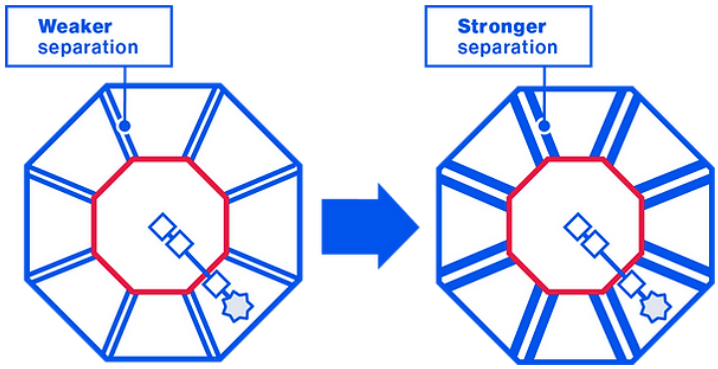
Reducing Complexity

Interface complexity and the risks associated with it can be reduced (to a point) by **right-sizing the type of interface to the type of expected input**, and by **improving API security**[38] (of both customer-facing interfaces as well as communications between frontend and backend) according to industry best practices[39]. Each of these can serve to lower complexity by applying limitations on what actions a user can perform, thus reducing a would-be attacker's aforementioned degree of control.



Improving Separation

Existing security boundaries can be **strengthened** (via hardening), **replaced** with stronger types, or **augmented** by additional types[40]. Each of these serves to lower the impact of exploitation by impeding lateral movement.



[37] These levers can be viewed as variations on the principles "make compromise difficult" and "reduce the impact of compromise" from NCSC's Cyber Security Secure Design Principles.

[38] Precisely defining a secure API is out of scope for this paper, but this would include strictly defining, validating, and sanitizing input; avoiding deserialization of untrusted data; barring complex object packing and unpacking; and preventing ingestion as input of complex buffer containing code.

[39] e.g., OWASP Top Ten or Shieldfy's API Security Checklist

[40] Note that layering multiple instances of the same boundary type (such as nested virtualization) is not cost-effective, as this would constitute a change in quantity rather than quality, and the gained security barriers would not be truly independent from one another, as a vulnerability in one would likely be just as applicable in another.

We propose that a strictly hardened IaaS-based primary security boundary (hardware separation or virtualization) is generally suitable for ensuring tenant isolation regardless of interface complexity. Likewise, an application-level primary security boundary (containerization or data segmentation) should be considered adequate for isolating sufficiently simple interfaces, such as those found in many SaaS offerings.

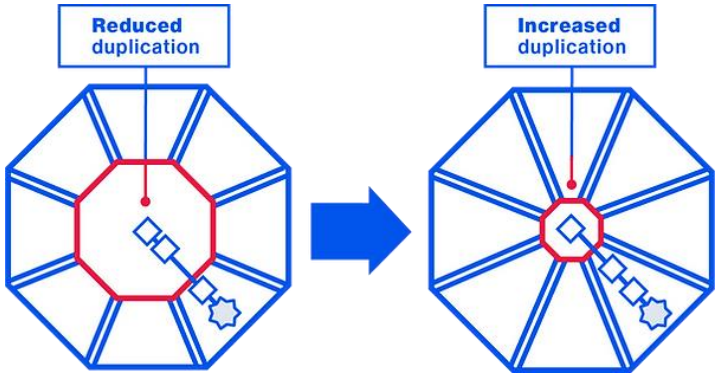
However, any single application-level security boundary encapsulating a non-simple interface should be assumed breachable – no matter how well-hardened – thus requiring an additional boundary (either IaaS or application-level)[41]. This secondary boundary must be **independent of the first (of a different type, or "oppositely polarized")**, such that being able to break either one has little bearing on an attacker's capability of breaking the other.

In other words, we should ideally consider the security model of a given cloud service to be working as intended only if the prerequisite for compromising the integrity or confidentiality of a neighboring tenant's data is the discovery and exploitation of a **0day vulnerability in an IaaS boundary** (such as VM escape), or of **multiple (≥2) completely independent 0day vulnerabilities in application-level boundaries** (such as container escape). Thus, in our view, a well-designed cloud service should be able to withstand a scenario such as a malicious tenant "merely" achieving local privilege escalation (LPE) on their guest machine, escaping their sandbox, or gaining host-level root permissions.

Note that stored user data in particular (i.e., a datastore in the form of a cache, queue, database, or specific database entries) should always be tenant-isolated by way of a strictly hardened data segmentation boundary (at minimum), whether the corresponding interface is duplicated or shared.

Increasing Duplication

While security boundaries serve to hinder a malicious tenant's attempts at both breaching vendor control plane components and directly accessing other tenants' data, we can further limit the impact of interface vulnerabilities by shifting shared functionality outside the vendor's trust boundary and duplicating it (or localizing it) according to a certain organizing principle (per-tenant, per-cluster, per-region, per-zone, etc.)[42]. For example, we could shift a hitherto shared server to the tenant's side of the trust boundary by duplicating it on a virtual machine per-tenant.



As explained previously, duplication serves to limit the blast radius of potential interface vulnerabilities and confine it to a specific environment (ideally, that of a single tenant). If taken to the extreme by adhering to the principle of least common mechanism[43], we could minimize shared functionality while maximizing duplicated functionality per tenant[44] (at the expense of efficiency).

[41] One might call this a "belt and suspenders" approach to tenant isolation.
[42] Duplication can also be used within a single tenant's environment to limit the impact of potential cross-user access vulnerabilities. For example, prior to the discovery of ZapeScape (CVE-2022-28802), an intra-tenant privilege escalation vulnerability in Zapier, Lambda function instances were only duplicated per tenant, but following disclosure, duplication was increased by deploying an instance per user.
[43] See "Least Common Mechanism", *Build Security In*, CISA
[44] We can find precedence for this design principle in the partial migration of operating system functionality in Windows from kernel mode to user-mode, such as font parsing. Also see sections on eliminating trusted code from "Some thoughts on security after ten years of qmail 1.0", Daniel J. Bernstein.

By virtue of downsizing vendor-side functionality, duplication also has the added benefit of **reducing vendor control plane complexity**, thus further lowering the overall likelihood of isolation escape vulnerabilities in the service environment as a whole[45]. However, note that interface vulnerabilities located "deeper" within the control plane are harder to mitigate by way of duplication alone, and duplicating the entire control plane per tenant would be equivalent to on-premises and unfeasible at scale.

Note that additional duplication should always be accompanied by some degree of separation (by repurposing existing boundaries to encompass newly duplicated components, or by placing additional boundaries); as a rule, every duplicated component should have a baseline of at least one boundary, regardless of complexity. By duplicating and separating more components, we place the tenant "further away" from the vendor's trust boundary, which in turn shrinks attack surface, due to isolation compensating for complexity and lowering the impact of potential vulnerabilities.

7. Additional Considerations

Over-isolation

The addition of too many duplicated components and security boundaries requiring orchestration could ironically lead to an undesired outcome in the form of an increase in the overall complexity of the environment, which in turn could introduce new vulnerabilities. For this reason, it is important to balance tenant-isolation against other factors. Such **"over-isolation"** can be avoided by prioritizing the duplication of complex components over simpler ones and limiting degree of separation to the minimum necessary. It is ultimately up to the vendor to decide how best to finetune their architecture, depending on important additional context such as **external constraints** (e.g., budget and compliance requirements) and expected **use-case characteristics** (e.g., data sensitivity).

Secure Control Plane Development

As mentioned, duplication can serve to lower vendor control plane complexity. However, vendors could also apply the following principles to further reduce the likelihood of control plane vulnerabilities:

1. Preference for small[46], well-managed and well-tested codebases, as software with a large legacy codebase is assumably more likely to be complex and prone to vulnerabilities[47].
2. Use of memory-safe programming languages (such as Java, Rust, or Go) to further reduce the likelihood of certain classes of vulnerabilities.
3. Development of control plane libraries and tools as open-source projects, perhaps even utilized by multiple vendors[48], which would serve to increase public scrutiny of the control plane's design and implementation.

Detective Controls

Besides the use of preventive security boundaries, vendors can also implement detective controls in the form of **monitoring mechanisms at the host and network level**, which could take the form of host-based agents, intrusion protection systems (IPS)[49] and/or tripwires[50].

---

[45] As mentioned earlier, we assume that complexity correlates positively with likelihood of vulnerability.
[46] See "Analytic Framework for Cyber Security", Peiter "Mudge" Zatko
[47] We can find precedence for this rationale in the following well-established cases: (1) Chrome's V8 JavaScript engine has a large codebase that can be assumed vulnerable while the smaller sandbox can be considered more secure; (2) the Windows kernel is a large legacy component as opposed to the smaller and more modern hypervisor (Hyper-V); (3) ARM TrustZone virtually splits the processor between trusted and non-trusted "worlds".
[48] Similarly to the Linux kernel or Chromium.
[49] Equivalent to MITRE ATT&CK Enterprise M1040 (Behavior Prevention on Endpoint) and M1031 (Network Intrusion Prevention), respectively.
[50] e.g., Thinkst canaries

Monitoring mechanisms should be set up in such a way as to detect host-level activity that might indicate sandbox escape, code execution, or local privilege escalation, as well as network-level activity indicative of reconnaissance or lateral movement. Each of these incursion scenarios should be regularly tested against to ensure sufficient coverage.

Monitoring can create more opportunities for would-be attackers to fail, and for defenders to detect and respond to malicious activity in time. Moreover, monitoring can help compensate for undiagnosed weak links; unrecognized implementation failures or isolation drift; as well as undiscovered (0day) vulnerabilities. Detection efficacy is highly dependent on observability into the service environment, and on uniformity of duplicated interfaces (the greater the similarity between instances, the easier it is to detect variance indicative of malicious activity).

Customer Adaptation

Cloud customers have levers of their own when it comes to deciding if and how to make use of a given cloud application (as informed by the conclusions of an isolation review): first, when comparing between different services, they could **assign more importance to isolation as a deciding factor** (if taken to the extreme, this would entail adopting the service with the highest isolation levels, regardless of other factors). They could also **use their buying power** to influence their chosen vendor to improve the service's isolation level, but this is a more strategic solution that could take time to implement.

Second, if possible, they could **adapt to low isolation levels** (i.e., accepting the risk and acting accordingly) by modifying their usage patterns of the service (such as avoiding storage of overly sensitive data like PII or payment information) or compensating by way of 3rd-party encryption (essentially bringing along their own trust boundary to place between themselves and other tenants).

8. Isolation Maintenance

Drift Prevention

The process of evaluating the isolation levels of a given service should be performed continuously or iteratively, starting at the initial design phase, and repeated prior to every major version release. This is meant to prevent **isolation drift**, caused by unintended differences between as-designed as opposed to as-built dataflows, as well as the addition of new functionality. During these "checkpoints", the implementation should be reviewed in comparison to the original design so as to ensure that no changes have been made that might break any assumptions related to either complexity or isolation (such as the addition of features granting the user greater freedom of action).

Isolation Verification

Automatic testing should be in place to challenge design assumptions and **empirically verify correct implementation**. This can take the form of provable security (via automated reasoning)[51], or of implementation failure "canaries", such as an intentionally broken peer-to-peer network set up within the service environment. If the network shows any signs of life (e.g., two hosts that shouldn't be able to communicate with each other suddenly establish a session), this could indicate isolation drift in the form of implementation changes contradicting assumptions of the original design.

Patch Management

Vendors should enforce timely **patch management** and **vulnerability scanning**[52] of all hosts in the service environment, so as to significantly reduce the probability of an attacker exploiting any known vulnerabilities (Ndays) which may have been discovered in relevant security boundary technologies since the launch of the service.

[51] e.g., AWS's IAM Access Analyzer and Network Analyzer

[52] Equivalent to MITRE ATT&CK Enterprise M1051 (Update Software) and M1016 (Vulnerability Scanning), respectively.

Appendix – Isolation case study: ChaosDB

"ChaosDB" was a chain of vulnerabilities discovered in Azure's Cosmos DB service and disclosed in August 2021. The attack sequence was as follows[53]:

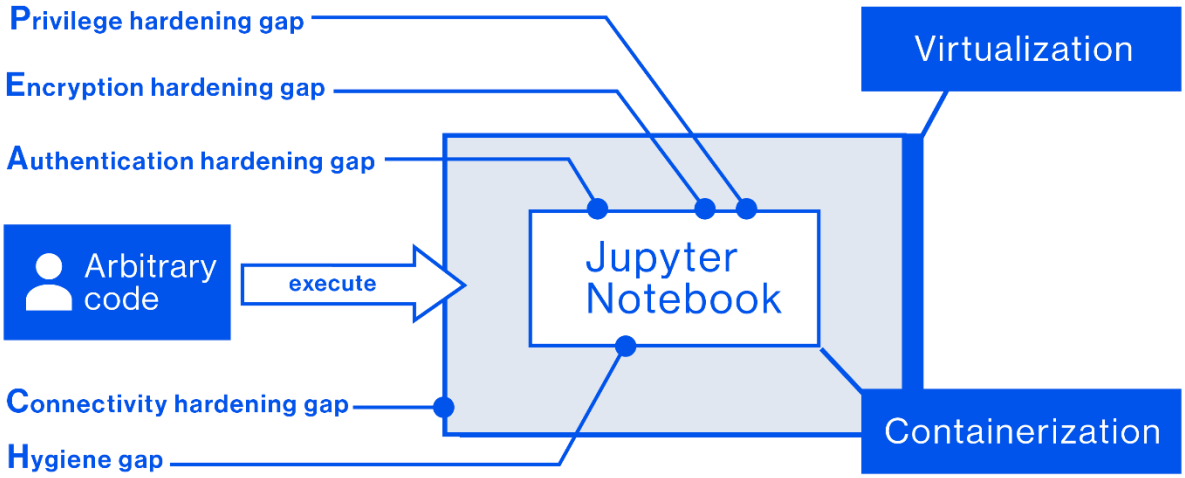
1. Deploy an embedded Jupyter Notebook container in an attacker-controlled Cosmos DB instance.
2. Run any C# code to elevate attacker privileges to host-level root.
3. Remove locally set firewall rules (iptables), thus gaining unrestricted network access in the environment.
4. Authenticate to an orchestrator management interface using a self-signed certificate, as the server did not properly validate these, and retrieve encrypted access keys belonging to other tenants.
5. Query the server to obtain certificates for other services, including an administrative certificate that allowed decryption of other tenants' access keys.

At the time of disclosure, each Cosmos DB tenant's embedded Jupyter notebook seemingly ran in a container nested within a virtual machine, thus fulfilling the requirement of a strong security boundary. However, this isolation scheme had gaps in all five PEACH hardening factors:

1. Privilege hardening gap – tenant-allocated VM with access to shared admin certificate.
2. Encryption hardening gap – tenant API keys encrypted with shared key.
3. Authentication hardening gap – self-signed certificate not validated.
4. Connectivity hardening gap – network controls only enforced within container (iptables) and orchestrator interface accessible from tenant container.
5. Hygiene gap – tenant access to unrelated certificates and keys.

For simpler interfaces this might have been sufficient, but Jupyter notebook should be considered a highly complex interface, as it allows arbitrary code execution. Therefore, these gaps in PEACH allowed the attack sequence to unfold.

Additionally, host protection was lacking – C# code execution was overprivileged – thereby creating an exploitable local privilege escalation (LPE) vulnerability which would have had minimal impact were it not for the insufficient isolation scheme. Moreover, the apparent lack of host-level or network-level monitoring reduced defense-in-depth, lowering the chances of discovering this attack.



Isolation design diagram and implementation table for Cosmos DB embedded Jupyter Notebook at the time of ChaosDB's discovery

[53] "ChaosDB explained: Azure's Cosmos DB vulnerability walkthrough", wiz.io