API Sprawl and Emergence

For those not chartered with digital trust in their organizations, emergence might be thought of as just a cool trick. They might point to a beehive, anthill or rush hour traffic with wonder at how simple, straightforward behaviors, in aggregate and at scale, have ramifications that are challenging to predict in advance from observation of a single bee, ant or commuter. It would be hard, for example, to predict the formation of a hive just by observing the behavior of a few individual bees. But when you put thousands of bees together, their (seemingly) simplistic behaviors result in something elegant, unique and completely unexpected.

For those of us in the professional disciplines ISACA® serves, however, emergence is a potential source of risk. Aristotle (one of the first known philosophers to discuss emergence) said in his *Metaphysics*, "The totality is not, as it were, a mere heap, but the whole is something besides the parts."[1]

ED MOYLE | CISSP

Is currently director of Software and Systems Security for Drake Software. In his 20 years in information security, Moyle has held numerous positions including director of thought leadership and research for ISACA®, application security principal for Adaptive Biotechnologies, senior security strategist with Savvis, senior manager with CTG, and vice president and information security officer for Merrill Lynch Investment Managers. Moyle is co-author of *Cryptographic Libraries for Developers* and *Practical Cybersecurity Architecture*, and a frequent contributor to the information security industry as an author, public speaker and analyst.

By this, he meant that the parts of a complex system go beyond what is predictable and observable and result in something unforeseen. These unforeseen outcomes represent a blind spot in our ability to plan and can undermine security or privacy, make it harder to provide validation, or decrease the effectiveness of governance.

Example: Legacy VM Sprawl

To illustrate this point, consider the example of virtual machine (VM) sprawl. I have picked this example carefully for three reasons. First, it is familiar. Anyone who remembers the rise of the virtual data center or anyone making extensive use of Infrastructure as a Service (IaaS) (or hybrid cloud) has likely seen it in action. Second, it is not a new problem, so many professionals will have seen or built countermeasures designed specifically to address the challenges that result from it. Third, it is directly analogous to a new issue that I think we must start looking at more carefully.

For those unfamiliar with VM sprawl, it refers to the uncontrolled or semicontrolled proliferation of VM images within a virtualization context such as a virtual data center or an IaaS cloud environment. There are a number of reasons why this happens. Individuals and teams create new VM images and use them for their intended purposes, but they perhaps do not provide timely or direct feedback on when those VMs can be decommissioned. Administrators are often uncomfortable removing workloads, so they allow them to persist. Also, environments subjected to extensive physical-to-virtual (p2v) migration likewise may have sprawl resulting from the legacy environments.

This is a problem for several reasons. Images serialized for long periods of time (i.e., that are spun down and not currently running) can become stale as critical security patches and updates are missed, making them a prime target for security issues when they are brought back up. VM images are also mobile (e.g., vMotion), meaning they can cross segmentation boundaries between environments or bring about intermingling in unexpected ways. The rate at which images tend to proliferate creates administrative overhead associated with keeping the environment managed and organized as inventory accuracy decays and the purpose of specific VM images becomes lost over time.

While these issues are perhaps not directly what Aristotle meant in his description, they do reinforce the idea that predicting this behavior in advance is challenging. So, if a practitioner had never worked with virtualization at scale, it would be difficult to know that these challenges would arise.

In the enterprise, people deployed strategies to combat sprawl, for example, authoring scripts to automatically delete VM images if inventory records and usage information were not kept current. Vendors developed solutions designed to address exactly these problems: everything from controls built into hypervisor systems to help with management and record keeping of the virtual environment to controls that assist with mitigation of specific technical problems (e.g., patching stale images, enforcing segmentation). The practitioner community developed strategies to address sprawl and organizations such as ISACA and others worked to disseminate those strategies to others to the point that now, while the problem can still arise, practitioners have become largely hardened against it.

API Sprawl

Over time, we have seen similar situations arise in other areas. For example, there has also been cloud sprawl (used variously to describe not only proliferation of IaaS workloads but also Software as a Service [SaaS] and Platform as a Service [PaaS] relationships), container sprawl (as in application containers from tools such as Docker and others), storage sprawl (think cloud storage), and so forth. As one might have guessed, we are now seeing a new situation develop with similar dynamics in play for many and on the near-term horizon for others: specifically, application programming interface (API) sprawl or the uncontrolled proliferation of REST APIs—APIs that conform to the design principles of the representational state transfer architectural style (REST)—web services and other similar technologies.

Who cares about APIs you ask? You should, for several reasons. First, there is a tendency for APIs to get created without a clear plan on the part of developers for how they will get decommissioned in the future. Think about it from a developer's point of view. Imagine a web service has been created to accomplish a particular piece of business logic. The API that was built works so well it starts being used by other internal web applications and other APIs. Perhaps another team develops a mobile application (app) that employs it after more time elapses.

There is a tendency for APIs to get created without a clear plan on the part of developers for how they will get decommissioned in the future.

From the developer's point of view, this is great. But what happens next? Say the developer needs to change an API from using HTTP GET to using HTTP POST, or wants to change the URL on which it is hosted, or wants to change the signature (i.e., number and content of inputs or the type and format of outputs.) It can be difficult to do these things because of the other components that have the API as a dependency. Sure, developers can deprecate the old version of the API and create a new update. But since it is actively being used, they may feel pressure to not do so or to retain both the new and the old version of the interface. Hence the sprawl.

Emergent and risk-undermining behavior can arise in a few different ways given this backdrop. Under DevOps and/or DevSecOps, where changes to software can be particularly fast-paced, the order in which APIs are called can change literally from day to day. Likewise, with technologies such as service mesh (e.g., Istio) or API concentrators (e.g., KrakenD), the complexity compounds as indirection is introduced, either via the concentrator or via "sidecar" reverse proxy (e.g., under service mesh).

With the use of security or assurance controls that assume a static ordering of what is called and when, this will absolutely have a significant impact. As an example, application threat modeling is one such control that normatively assumes a constant and static path through the application logic. In fact, threat modeling normally begins with the creation of a data flow diagram. This means that the starting point for threat modeling is the execution path through components. But what happens when that pathway is ever-changing? To say that it undermines threat modeling is a significant understatement.

A second behavior to examine here is the intersection between APIs and the technology used to access them. Consider an API that uses the GET verb and requires callers to submit input via query parameters—for example, a request such as *https://api.exampleorg.dom/processUserData?username=testuser&supersecretvalue=somesecret*. From a risk perspective, that call is problematic. Many (including me) would argue that using GET in this way is almost always problematic for a variety of reasons, but it is particularly problematic when called from a browser. Why? Because the browser will, by default, cache the values (including the secret value) in the history, and normative browser behavior will be to include the referring URL for any downstream request. If there is a landing or callback page after that API that sources content hosted off site (e.g., images, fonts, JavaScript libraries), any such requests will include the full URL given in the "Referer"(sic)[2] header value. Given the presence of a secret value in the query parameters, obviously neither caching of the value nor relaying it to third parties is desirable.

ENJOYING THIS ARTICLE?

- Learn more about, discuss and collaborate on emerging technology in ISACA's Online Forums. *https://engage.isaca.org/onlineforums*

Depending on the particular environment, there can be dozens of behaviors introduced that one might not expect. They are generally less visible from a security, audit and governance perspective for several reasons. First, because many organizations spend comparatively less time on application security than other technology areas. Second, it can require some investigation to actually find, understand and look in depth into the APIs used within the organization for developed software, integrations with commercial off-the-shelf (COTS) software, software customizations, software used to support business partner relationships, etc.

On the plus side, knowing what to look for helps practitioners to arm themselves and minimize potential negative outcomes. And it turns out there are a few things they can do. First and foremost, practitioners can lean into making sure that they have documentation for the APIs in use. In addition to commercial tools, open source tools such as Redoc[3] and Swagger UI[4] help generate consistent API documentation. Assuming developers play ball by authoring documentation consistently, these tools can help practitioners understand what APIs do, what they are for, and how they are used.

Additionally, while uncontrolled use of service mesh, API gateways and the like can compound confusion, they can help reduce or alleviate that confusion when used in a controlled way. Instead of having to track and maintain where APIs are hosted, for example, the practitioner can use service mesh or API gateways to keep track of where APIs are hosted at any given point in time. In this way, they can be used as an authoritative source of truth for where individual APIs live and where and how they are being used.

Lastly, it can often be helpful for practitioners to gain some familiarity with knowing how to test APIs. Open source tools such as OWASP's ZAP[5] or SoapUI[6] can provide information for the practitioner willing to dig deeper. Because APIs use HTTP/s as their transport mechanism, practitioners with a solid understanding of HTTP can fairly easily understand the mechanics of how most APIs are called. For those with less experience in this arena, HTTP is a fairly simple protocol—one that is advantageous for practitioners to know. The skill base to understand API functionality is fairly readily built for those desirous of learning.

Regardless of how practitioners choose to engage with the many APIs in their scope, it is important that they begin to factor APIs into their planning. APIs are not only intrinsic to how modern applications are built, but also potential sources for emergent, and thereby unexpected, behavior.

Endnotes

1 Aristotle; *Metaphysics, Book VIII*, Greece, 350 BC, translated by W. D. Ross, *http://classics.mit.edu/Aristotle/metaphysics.8.viii.html*

2 Note that the HTTP "Referer" header is spelled incorrectly in the governing specification (RFC 1945). As the misspelling is part of the specification and is reflected in actual browser behavior (in conformance with the specification, browsers also preserve the misspelling), it is reproduced verbatim here for accuracy.

3 GitHub, Redocly/redoc, *https://github.com/Redocly/redoc*

4 GitHub, swagger-api/swagger-ui, *https://github.com/swagger-api/swagger-ui*

5 ZAP, OWASP Zed Attack Proxy (ZAP), *https://www.zaproxy.org/*

6 SoapUI, Accelerating API Quality Through Testing, *https://www.soapui.org/*