Chaos Engineering: New Approaches To Security

Jamie Lewis, Chenxi Wang

August 2019

RAIN CAPITAL

Chaos Engineering: New Approaches To Security

A Rain Capital Research Note

Jamie Lewis, Dr. Chenxi Wang

Introduction: Cloud-Native Architectures Require Testing *and* Experimentation

As we discussed in the Rain Capital Research Note "DevSecOps and Detection Engineering: New Approaches To Security," the move to cloud-native architectures is having a profound impact on both security posture and operations. In cloud-native environments, organizations must apply the principles that drive DevOps and site reliability engineering (SRE) to security, bringing security programs into architectural and organizational alignment with the systems they protect. New controls, real-time metrics, and rapid feedback loops are just a few of the requirements security teams face in cloud-native environments.

Testing, of course, has always been a critical component of both the application development process and any effective security program. And like application development itself, testing techniques and timeframes have changed dramatically with the DevOps model. Within Continuous Integration/Continuous Deployment (CI/CD) pipelines, constant testing and improvement are an operational given. More significantly, cloud-native companies such as Netflix have discovered that testing alone cannot ensure the resilience of cloud-native systems. They have moved beyond testing to experimentation, using chaos engineering.

But in more traditional security operations, security testing is often a discrete phase in a linear and relatively static process. Code reviews often occur late in a development process, for example. In the larger enterprise context, penetration testing, red team exercises, and other approaches, while valuable, occur long after deployment and often aren't well-integrated with the development process. Like other elements of traditional enterprise security programs, these approaches must evolve to support DevOps and protect cloud-native systems.

The Rain Forecast: Chaos Engineering Increases the Resilience of Security Systems

As they apply DevOps and SRE principles to security controls, security teams must adopt new testing techniques and time frames, enabling continuous security pipelines.

To effectively support cloud-native systems, security testing must become a more continuous practice, reflecting the nature of CI/CD operations. Enterprises must also consider the long-term implications of distributed systems and the need to move beyond testing systems to experimentation. To that end, security professionals are advocating the application of chaos engineering to security operations.

Chaos engineering is the technique of using controlled experiments--often in production systems--to discover flaws in complex, distributed systems before problems happen. Chaos experiments have played a significant role in increasing reliability, uptime, and overall confidence in complex cloud-native systems, and have thus gained a great deal of attention. It seems only natural to apply the technique to the security mechanisms designed to protect these systems.

Like detection engineering, security chaos engineering drives stronger architectural and organizational alignment between security programs and the systems they protect. By uncovering unpredictable failure modes, security chaos engineering can:

- Increase resilience
- Improve risk management decisions
- Help security managers allocate budgets more effectively

Chaos engineering is a compliment to, not a replacement for, many existing security testing methods. While they must evolve to support DevOps in general and CI/CD in particular, existing security testing models play important roles in security operations. Red and purple team exercises focus on an adversarial approach, for example, allowing organizations to test how security systems and teams respond to active threats. In contrast, chaos engineering allows organizations to find security failures that are difficult, if not impossible, to find by traditional testing methods due to the complex nature of distributed systems. These include failures caused by human factors, poor design, or lack of resiliency.

As they transition to cloud-native systems, organizations should consider how and when to incorporate chaos engineering into their security programs. While these potential benefits are significant, however, organizations should not attempt chaos engineering without careful planning. Perhaps more than any technique in the DevOps arsenal, chaos engineering requires strong organizational alignment, a good bit of experience, and a methodical approach. Consequently, organizations without that experience should carefully consider how and when to start applying the technique to their systems, starting small and learning as they go.

Chaos Engineeering

One of the cardinal rules of distributed systems states that the most significant failures usually occur not within individual system components, but in their interactions. Given the significant number of microservices in a typical cloud-native system, and the exponential number of possible interactions between those services, the outcomes of all those interactions are nearly impossible to predict. Accepting that reality, Netflix pioneered the concept of chaos engineering as a means of rooting out failure modes before they cause problems in production systems.

Netflix defines chaos engineering as:

> **"the discipline of experimenting on a distributed system in order to build confidence in the system's capability to withstand turbulent conditions in production."**

In other words, instead of waiting for systems to fail, developers should introduce failures through experiments—in controlled conditions—so they can see what happens.

Testing vs. Experimentation

In general, testing allows a team to assess known capabilities and system attributes. In contrast, experimentation is about discovering the unknown. All of which leads us to phrases such as "known knowns" and "unknown unknowns."

When Donald Rumsfeld famously used the term "known knowns", he brought a new level of notoriety to a concept familiar to many national security and intelligence officials. The idea of known and unknown risks, which has its roots in cognitive psychology and the Johari Window, has come into more frequent use in software development circles. It certainly applies to the security of complex distributed systems. Chaos experiments are a means of better understanding "known unknowns", becoming aware of "unknown knowns", and discovering "unknown unknowns". Figure 1 illustrates these various states of system- (or self-) knowledge.

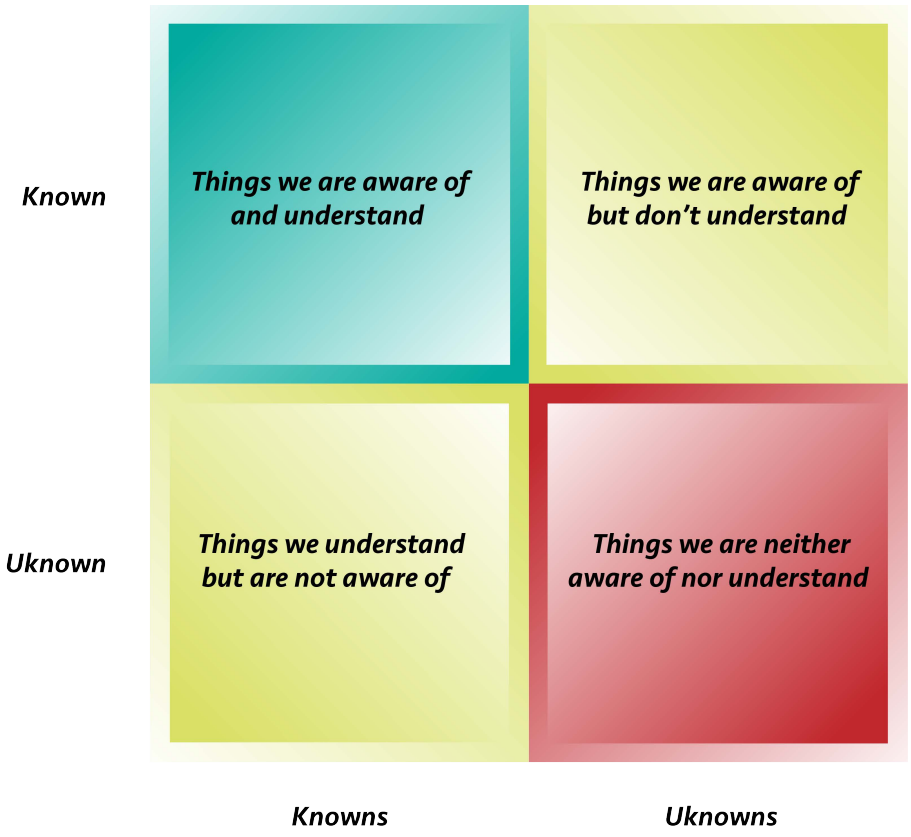

Figure 1: The Known Unknowns Matrix

The term "experiment" also brings with it the essential notion of relying on the scientific method. Instead of making multiple changes to the system, or introducing an external event, the experiment involves making one change. The team introduces that change under controlled conditions, limiting the "blast radius," or potential impact. As Figure 2 illustrates, if the team finds no problems within a small blast radius, it can gradually increase the radius, until experiments run at scale. In most cases, teams run both an experimental group and a control group to evaluate what impact changes made on system behavior. The goal of these experiments is to assess and increase both the observability and understanding of how a complex system actually works.

Early on, for example, Netflix created Chaos Monkey, a tool that randomly shuts down instances in its production systems to test how the remaining systems respond to outages. Knowing that Chaos Monkey could hit their systems at any time gave developers a strong incentive to focus on resilience. And as it operated, Chaos Monkey revealed "unknowns" -- information about how the system actually works under a variety of conditions that was difficult to know beforehand. Netflix released Chaos Monkey under an open source license in 2011, and its chaos engineering toolbox has evolved significantly since then. (For more information on using chaos engineering in software development, see this and this.)

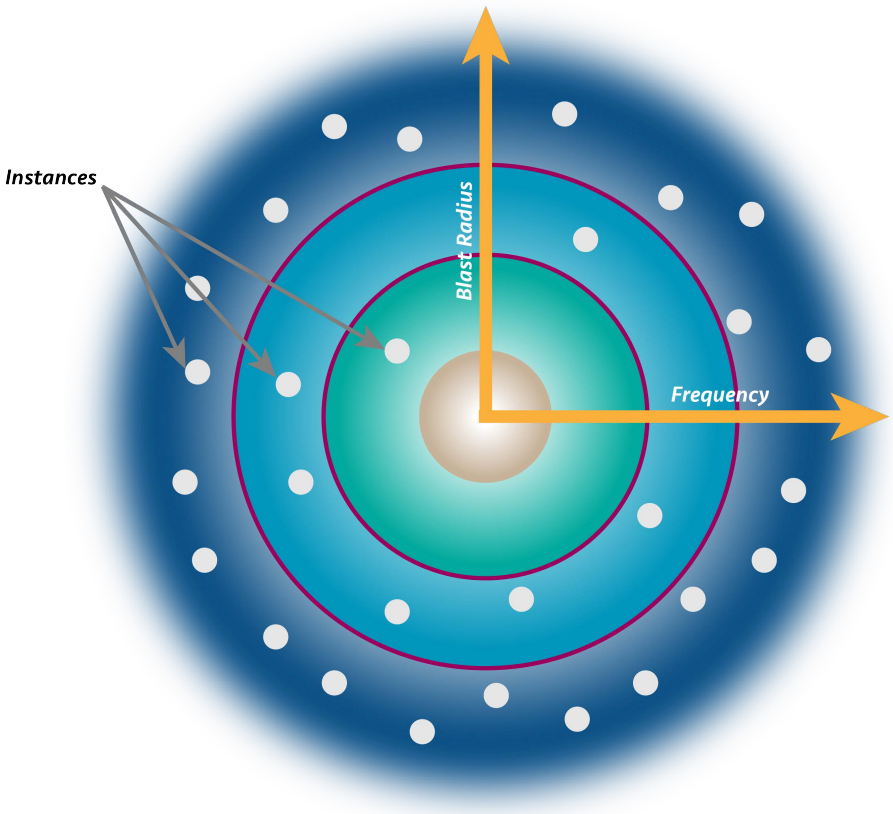

Figure 2: The Blast Radius

Security Chaos Engineeering

As they worked to secure cloud-native environments, both Charles Nwatu and Aaron Rinehart came to the conclusion that applying chaos engineering to security is imperative, co-authoring a post on the subject. (When they wrote the post, Nwatu was director of security at StitchFix. He's now with Netflix. Rinehart was chief security architect with UnitedHealth Group and led the development of ChaosInger. He's now with Verica.) In that post, Nwatu and Rinehart define "security chaos engineering" as:

> **"the discipline of instrumentation, identification, and remediation of failure within security controls through proactive experimentation to build confidence in the system's ability to defend against malicious conditions in production."**

The goal of these experiments is to assess and increase observability, moving security from subjective assessment into objective measurement. As they do in the DevOps world, chaos experiments allow security teams to reduce the "unknown unknowns" and, over time, replace "known unknowns" with information that can drive improvements to security posture. By intentionally introducing a failure mode, security teams can discover how well-instrumented--observable and measurable--security systems truly are. Teams can see if functions are working as well as everyone assumed they would, objectively assessing abilities and weaknesses, moving to stabilize the former and eliminate the latter.

These experiments are very different from traditional security testing. In a red team exercise, for example, a team may simulate a known attack in order to test specific controls designed to detect or prevent that attack. Or a security team can test anti-phishing measures to see how they work. Such tests can yield useful, but large-grained results, such as whether an attack failed or succeeded. But such tests also create a great deal of noise in complex, distributed systems, making it difficult to learn fine-grained detail about how security controls actually work under a variety of conditions. And it's in the interactions and failure modes of the components in a distributed system that the most significant security failures lurk.

Example Experiments

Instead of making multiple changes to the system, or introducing an external event (as in an attack simulation), the experiment involves making one change and observing what happens. Take, for example, a simple but seemingly perennial security problem in cloud-based systems: an open Amazon S3 bucket. In a controlled experiment, the security team could intentionally open an S3 bucket to unauthorized Internet access. Doing so allows the team to see if the controls designed to detect the problem actually work. More important, the team can also see what happens under those circumstances and compare actual results with expected results.

Did the system produce the expected alerts and telemetry? Did the proper people receive those alerts in an acceptable time frame? Was the information in those alerts complete enough to allow them to take appropriate action, or did people have to dig to find the information they needed, wasting precious time? Most importantly, were there unexpected consequences or outcomes and if so, how should the team address them?

Another slightly more complicated example is AWS's sometimes complex and confusing identity and access management system. Application developers may need to change a role-based policy to support the service component they're building. But they may not understand the potentially far-reaching effects of changing such a policy given that many other components in the same or other systems may rely on that same policy.

By intentionally making a change in a critical IdAM policy, security teams can watch what happens under such circumstances. Was the developer asked via a Slack message to confirm the change? Were the proper people on the security team notified, and given the information necessary to contact the developer and discuss the change in the context of the project? Was the change properly vetted with risk owners in the business? Did unexpected things happen? And what do those things tell you about the system?

These are just two relatively simple examples. Security teams can apply chaos engineering to a wide variety of scenarios and controls. And it's in the unexpected results that security managers will learn the most about how their systems actually work and how they can best improve them. As Rinehart points out, more often than not, the response from security teams is **"I didn't realize the system worked like that."**

The Benefits of Chaos

Applying chaos engineering to security has several clear benefits. In short, it can:

- **Expose flaws and problems that are difficult to discover a priori,** defining the delta between how the security team thinks the system operates and how it actually operates. Chances are high that the team will find out a lot it didn't know about how the system actually works, discovering work it needs to do to improve the observability and security of the system.
- **Enhance risk management decisions,** helping business and security people determine how best to spend their security budgets. Nwatu advocates using chaos experiments to make vulnerability assessments and identify capability gaps. Managers can more clearly identify levels of risk and work with business (risk) owners to make better budget investment decisions, driving down risk by increasing capabilities in specific areas.
- **Give teams experience with failures,** significantly increasing their ability to deal with them quickly and effectively. Most security teams have playbooks that define an organization's incident response plan. Gameday exercises using security chaos experiments allow organizations to validate those playbooks—and the teams tasked with running them—measuring what actually happens, before real problems occur.
- **Bring security into architectural alignment with the systems it protects,** enabling end-to-end instrumentation that makes system behavior observable and measurable. Red and purple team exercises are more difficult to perform on a continuous basis. The constrained nature of chaos engineering makes continuous (rapid and iterative) improvement more feasible.

Products and Tools

Chaos engineering is a relatively new idea in the security domain. As the first to apply chaos engineering to software development, Netflix built many of its own tools. But given the results organizations such as Netflix have had with the practice, general-purpose tools have started to emerge, allowing more organizations to add chaos engineering to their cloud development arsenal. Such tools typically provide a general-purpose framework for developing experiments and tools for deploying them and reporting the results. Libraries of specific, focused experiments work within these general-purpose frameworks.

Currently, there aren't many security-specific experiments in the libraries included in these products, but we expect that to change. There is a growing community of security professionals who are both advocating security chaos engineering and developing tests through open source and other community initiatives. And as general purpose chaos engineering tools mature, their experiment libraries will include more security-specific experiments. Today, however, security managers should be prepared to design and build their own experiments using the tools these general-purpose platforms provide.

Here, we provide a basic overview of some of the tools available:

- **ChAP:** Netflix developed the Chaos Automation Platform, or ChAP, to automate experiments due to the rapid change inherent to its production systems, taking the notion of continuous experimentation quite literally. ChAP interrogates the deployment pipeline for a specific service. It then launches both experiment and control clusters of that service and routes a small amount of traffic to each. A specified scenario is applied to the experimental group, and the results of the experiment are reported to the service owner. ChAP will automatically end an automated experiment if it exceeds a predefined error budget. Netflix integrated ChAP with Spinnaker, its CI/CD system, allowing the engineers to run experiments often and continuously. Netflix says it has identified and prevented resiliency-threat regressions since deploying ChAP in this fashion.
- **The Chaos Toolkit:** The Chaos Toolkit is an open source project that provides an extensible toolkit for experiments that developers can adapt to specific use cases. The Chaos Toolkit allows developers to create their own "probes" (for observing system state as part of an experiment) and "actions" (for affecting the system while conducting an experiment). Developers can write and package a Python function in a module that can be called from the Chaos Toolkit, execute an arbitrary executable, or invoke an HTTP endpoint.
- **Gremlin**: Gremlin provides a general-purpose, commercial product that the company positions as "failure as a service." Gremlin supports chaos engineering experimentation and reporting on its library of failure testing modes, including resource exhaustion (CPU, Memory, IO, or Disk bottlenecks), bad behavior (dying processes, time drifts, instance reboots), and unreliable networks. The Chaos Toolkit team recently announced that the toolkit supports Gremlin.
- **Verica**: A recent startup, Verica was founded by Rinehart and Casey Rosenthal, who ran the chaos engineering team at Netflix. Rosenthal also co-authored the O'Reilly book on Chaos Engineering and the manifesto at PrinciplesofChaos.org. Verica provides an enterprise platform for what it calls "Continuous Verification," which is based on chaos engineering. Verica based its platform on Netflix's ChAP and, according to Rinehart, it includes security-specific chaos experiments. Verica's platform sits on-prem or within the customers' cloud.
- **Chaoslingr**: The first security-specific tool to appear was a relatively simple but interesting open source project, Chaoslingr. Created by a team led by Rinehart, Chaoslinger is a security experiment and reporting framework. Anyone can write their own experiments, but developers are encouraged to post their experiments for review and inclusion in the project. Written in Python, the framework consists of four AWS Lambda functions, as follows:
    - Generatr, which identifies the object to inject the failure on and calls Slingr
    - Slingr, which injects the failure
    - Trackr, which logs details about the experiment as it occurs
    - Experiment description, which provides documentation on the experiment along with applicable input and output parameters for Lambda functions

Beyond these general purpose chaos engineering tools, system-specific utilities for performing fault experiments are starting to emerge. For example, Istio, the popular open source service mesh technology, includes the ability to perform chaos experiments in a microservices environment, without the need to change the applications. More specifically, Istio allows testers to:

- Throw a 503 Service Unavailable error: System managers can easily configure Istio to return 503 errors to service requests, testing how robust distributed applications are in the face of unavailable services.
- Inject service response delays: Istio allows managers to inject variable-length network delays at different points in the system without changing any code. Random response delays can be a difficult problem to deal with in a complex microservices environment. Using this feature with Jaeger tracing, managers can spot problems proactively and increase system resiliency.
- Retry services a random number of times: When applications retry service requests after an unsuccessful attempt, they typically follow a predetermined pattern. With Istio, managers can dynamically change the number of retry attempts, another useful tool in distributed debugging and tracing.

Proceed With Caution

Chaos engineering is a clear example of how company culture and experience are tightly coupled with the cloud-native stack and DevOps model. At Netflix, the requirement for chaos engineering emerged as the company made the transition from using its own data centers to AWS. The company's culture--and practice of hiring talented engineers--allowed chaos engineering to evolve as an emergent property, based on how they were already working.

Companies new to DevOps and chaos engineering, then, should proceed carefully when and if they adopt the technique in any context. Unleashing full-blown security chaos engineering on your production system at one time would be what Rinehart calls "mayhem engineering," with all of the bad outcomes that moniker implies.

The first step is ensuring the right skill set is on staff. That means engineering talent, not just testers. Today, many organizations lack staff with skill sets necessary for chaos engineering. Even with the right talent, starting small is essential. Organizations should create a small team that can learn the tools and techniques, working with small experiments in the staging or development environment first. The team should start with a single host, container, or microservice in the test environment. As the team gains experience, it can expand the experiments to multiple hosts or other system components, methodically increasing its scope in lock step with rigorous analysis of the results.

Only when the team as gone as far as it possibly can in the test environment should it start experimenting in the production system. When doing so, the team should reset to the smallest experiment and scope possible, and grow from there. And some obvious rules should apply. Teams should never conduct a chaos experiment in production when they know it will cause severe damage, for example, possibly affecting customers. The team should also ensure that known problems have been fixed before starting experimentation.

Conclusion

As enterprises adopt cloud-native stacks and the DevOps model, their security programs must evolve to meet continuous deployment demands these systems create. Traditional security testing, while valuable, is insufficient to determine the resilience of security in cloud-native environments. When applied in a security context, chaos engineering has the potential to reveal valuable, objective information about how security controls operate, allowing organizations to invest security budgets more efficiently. Given that benefit, organizations should consider when and how to implement the technique in their cloud-native systems. Organizations without chaos engineering experience should start small, however, learning as they go.

Disclosure: Rain Capital has an investment in Tetrate, which provides products and services based on Istio and Envoy. As of this writing, Rain capital has no investments in the other companies mentioned in this report.