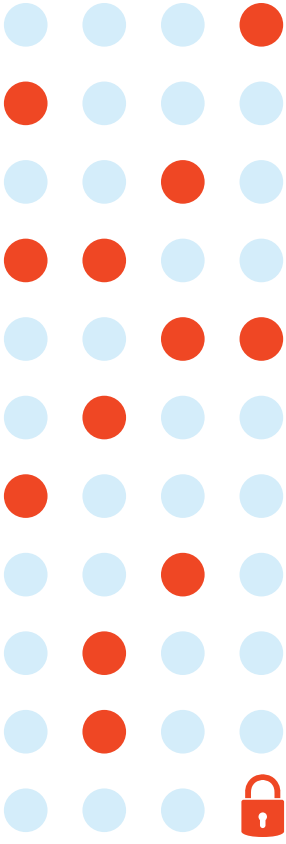
O'REILLY®

Security Chaos Engineering

Gaining Confidence in Resilience and Safety at Speed and Scale

Aaron Rinehart & Kelly Shortridge

REPORT



Proactively
discover
chaos

VERICA

before
it impacts
your *security*

www.verica.io

Security Chaos Engineering

Gaining Confidence in Resilience and Safety at Speed and Scale

Aaron Rinehart and Kelly Shortridge

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Security Chaos Engineering
by Aaron Rinehart and Kelly Shortridge

Copyright © 2021 O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Gary O'Brien
Production Editor: Christopher Faucher
Copyeditor: nSight, Inc.
Proofreader: Piper Editorial, LLC
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

November 2020: First Edition

Revision History for the First Edition
2020-11-09: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Security Chaos Engineering*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O'Reilly and Verica. See our statement of editorial independence.

978-1-098-10395-8

[LSI]

Table of Contents

The Case for Security Chaos Engineering

> *Definition of security chaos engineering: The identification of security control failures through proactive experimentation to build confidence in the system's ability to defend against malicious conditions in production.*[1]

Information security is broken. Our users and our customers—who make up our world—are entrusting us with more and more of their lives, and we are failing to keep that trust. Year after year, the same sort of attacks are successful, and the impact of those attacks becomes greater. Meanwhile, the security industry keeps chasing after the shiny new tech and maybe incremental improvement in the process.

A fundamental shift in both philosophy and practice is necessary. Information security must embrace the reality that failure *will* happen. People *will* click on the wrong thing. Security implications of simple code changes won't be clear. Mitigations *will* accidentally be disabled. Things *will* break.

By accepting this reality, information security can move from trying to build the perfect secure system to continually asking questions like "How will I know this control continues to be effective?", "What will happen if this mitigation is disabled, and will I be able to see it?", or "Is my team—including executives making critical decisions—ready to handle this sort of incident tomorrow?"

1 Aaron Rinehart, "Security Chaos Engineering: A New Paradigm for Cybersecurity," Opensource.com, January 24, 2018, *https://oreil.ly/gVMJk*.

Hope isn't a strategy. Likewise, perfection isn't a plan. The systems we are responsible for are failing as a normal function of how they operate, whether we like it or not, whether we see it or not. Security chaos engineering is about increasing confidence that our security mechanisms are effective at performing under the conditions for which we designed them. Through continuous security experimentation, we become better prepared as an organization and reduce the likelihood of being caught off guard by unforeseen disruptions. These practices better prepare us (as professionals), our teams, and the organizations we represent to be effective and resilient when faced with security unknowns.

The advanced state of practice for how we build software has reached a state where the systems we build have become impossible for our minds to mentally model in totality. Our systems are now vastly distributed and operationally ephemeral. Transformational technology shifts such as cloud computing, microservices, and continuous delivery (CD) have each brought forth new advances in customer value but have in turn resulted in a new series of challenges. Primary among those challenges is our inability to understand everything in our own systems.

This report does not consist of incremental solutions for how to fix information security. We are reassessing the first principles underlying organizational defense and pulling out the failed assumptions by their roots. In their place, we are planting the seeds of the new resistance, and this resistance favors alignment with organizational needs and seeks proactive, adaptive learning over reactive patching.

The Greatest Teacher

> *Pass on what you have learned. Strength, mastery, hmm…but weakness, folly, failure also. Yes: failure, most of all. The greatest teacher, failure is. Luke, we are what they grow beyond. That is the true burden of all masters.*
>
> —Jedi Master Yoda, *The Last Jedi*

Traditional defensive security philosophy is anchored to the avoidance of failure—preventing the inevitable data breach. Failure is seen as the axis of evil. We propose that failure is the greatest teacher we have in defensive security; it teaches us valuable lessons that inform us about how we can become better prepared for incidents.

If we have a poor understanding of how our systems are behaving, how can we drive good security in those systems? The answer is through planned, empirical experimentation. This report applies chaos engineering to the field of information security. We call this security chaos engineering (SCE). SCE is the way forward for information security and will facilitate the adaptation of defensive security to meet the requirements of modern operations.

SCE serves as a foundation for developing a learning culture around how organizations build, operate, instrument, and secure their systems. The goal of these experiments is to move security in practice from subjective assessment into objective measurement. Chaos experiments allow security teams to reduce the "unknown unknowns" and replace "known unknowns" with information that can drive improvements to security posture.

By intentionally introducing a failure mode or other event, security teams can discover how well instrumented, observable, and measurable their systems truly are. Teams can validate critical security assumptions, assess abilities and weaknesses, then move to stabilize the former and mitigate the latter.

SCE proposes that the only way to understand this uncertainty is to confront it objectively by introducing controlled signals. By injecting a controlled signal such as a security failure into the system, it becomes possible to measure your team's capability of responding to incidents. Additionally, we can proactively gain insights into how effective the technology is, how aligned runbooks or security incident processes are, and much more. As a practice, this helps teams better understand attack preparedness by tracking and measuring experiment outcomes across varying periods of time.

This report shares the guiding principles of SCE so you can begin harnessing experimentation and failure as a tool for empowerment —and so you can transform security from a gatekeeper getting in the way of business to a valued advisor that enables the rest of the organization.

CHAPTER 1

Experimenting with Failure

Chaos engineering is the practice of continual experimentation to validate that our systems operate the way we believe they do. These experiments help uncover systemic weaknesses or gaps in our understanding, informing improved design and processes that can help the organization gain more confidence in their behavior. The occurrence of failure is part of the normal condition of how our systems operate. Chaos engineering offers engineers a practical technique for proactively uncovering unknown failure within the system before it manifests into customer-facing problems.

The Foundation of Resilience

So what do we mean by resilience? According to Kazuo Furuta, "Resilience is the intrinsic ability of a system to adjust its functioning prior to, during, or following changes and disturbances so that it can sustain required operations under both expected and unexpected conditions."[1]

Resilience represents the ability not only to recover from threats and stresses but to perform as needed under a variety of conditions and respond appropriately to both disturbances as well as opportunities.

1 Kazuo Furuta, "Resilience Engineering," in Joonhong Ahn, Cathryn Carson, Mikael Jensen, Kohta Juraku, Shinya Nagasaki, and Satoru Tanaka (eds), *Reflections on the Fukushima Daiichi Nuclear Accident* (New York: Springer, 2015), sec. 24.4.2.

However, we commonly see resilience reduced to robustness in the information security dialogue (though it is far from the only domain felled by this mistake). A focus on robustness leads to a "defensive" posture rather than an adaptive system or one that, like a reed bending in the wind, is designed to incorporate the fact that events or conditions will occur that negatively impact the system. As a result, the status quo in information security is to aim for perfect prevention, defying reality by attempting to keep incidents from happening in the first place.

Robustness also leads us to prioritize restoring a compromised system back to its prior version, despite it being vulnerable to the conditions that fostered compromise.[2] This delusion drives us toward technical controls rather than systemic mitigations, which creates a false sense of security that facilitates risk accumulation in a system that is still inherently vulnerable.[3] For instance, if a physical barrier to flooding is added to a residential area, more housing development is likely to occur there—resulting in a higher probability of catastrophic outcomes if the barrier fails.[4] In information security, an example of this false sense of security is found in brittle internal applications left to languish with insecure design due to the belief that a firewall or intrusion detection system (IDS) will block attackers from accessing and exploiting it.

Things Will Fail

Detecting failures in security controls early can mean the difference between an unexploited vulnerability and having to announce a data breach to your customers. Resilience and chaos engineering embrace the reality that models will be incomplete, controls will fail, mitigations will be disabled—in other words, things will fail. If we

2 A. X. Sanchez, P. Osmond, and J. van der Heijden, "Are Some Forms of Resilience More Sustainable Than Others?" *Procedia Engineering* 180 (2017): 881–889.

3 This is known as the safe development paradox: the anticipated safety gained by introducing a technical solution to a problem instead facilitates risk accumulation over time, leading to larger potential damage in the event of an incident. See R. J. Burby, "Hurricane Katrina and the Paradoxes of Government Disaster Policy: Bringing about Wise Governmental Decisions for Hazardous Areas," *The Annals of the American Academy of Political and Social Science* 604, no. 1 (2006): 171–191.

4 C. Wenger, "The Oak or the Reed: How Resilience Theories Are Translated into Disaster Management Policies," *Ecology and Society* 22, no. 3 (2017).

architect our systems to expect failure, proactively challenge our assumptions through experimentation, and incorporate what we learn as feedback into our strategy, we can learn more about how our systems actually work and how to improve them.

Failure refers to when systems—including any people and business processes involved—do not operate as intended.[5] For instance, a microservice failing to communicate with a service it depends on would count as a failure. Similarly, failure within SCE is when security controls do not achieve security objectives. Revoked API keys being accepted, firewalls failing to enforce denylists,[6] vulnerability scanners missing SQLi, or intrusion detection not alerting on exploitation are examples of security failure.

Instead of seeking to stop failure from ever occurring, the goal in chaos engineering is handling failure gracefully.[7] Early detection of failure minimizes the blast radius of incidents and also reduces post-incident cleanup costs. Engineers have learned that detecting service failures early—like excessive latency on a payment API—reduces the cost of a fix, and security failure is no different.

Thus we arrive at two core guiding principles of SCE. First, expect security controls to fail and prepare accordingly. Second, do not attempt to completely avoid incidents but instead embrace the ability to quickly and effectively respond to them.

Under the first principle, system architecture must be designed under this assumption that security controls will fail, that users will not immediately understand (or care about) the security implications of their actions.[8] Under the second principle, as described by ecological economics scholar Peter Timmerman, resilience can be thought of as the building of "buffering capacity" into a system to

5 Pertinent domains include disaster management (e.g., flood resilience), climate change (e.g., agriculture, coral reef management), and safety-critical industries like aviation and medicine.

6 We will use the terms "allowlist" and "denylist" throughout the report. See M. Knodel and N. Oever, "Terminology, Power, and Inclusive Language," Internet Engineering Task Force, June 16, 2020, *https://oreil.ly/XsObA*.

7 See Bill Hoffman's tenants of operations-friendly services in J. R. Hamilton, "On Designing and Deploying Internet-Scale Services," *LISA* 18 (November 2007): 1–18.

8 "End users" and "system admins" are continually featured as "top actors" involved in data breaches in the annual editions of the *Verizon Data Breach Investigations Report*.

continually strengthen its ability to cope in the future.[9] Acceptance that compromise and "user error" will happen and a focus on ensuring systems gracefully handle incidents are essential. Security must move away from defensive postures to resilient postures and let go of the impossible standard of perfect prevention.

Benefits of SCE

The practice of SCE and leveraging failure result in numerous benefits. Among these are a reduction in remediation costs, disruption to end users, and stress level during incidents, as well as improvement of confidence in production systems, understanding of systemic risk, and feedback loops.

SCE reduces remediation costs by better preparing teams to handle incidents through the repeated practice of recovering from unexpected events. Security teams may have incident response plans available somewhere in their knowledge base, but practicing the process is a more reliable way to gain comfort in the ability to efficiently recover from incidents. Think of it as your team developing muscle memory.

SCE also reduces disruption to end users.[10] Each scenario tested generates feedback that can inform design improvements, including changes that can help minimize impacts to users. For example, simulating a distributed denial-of-service (DDoS) attack, which causes an outage in an ecommerce application, could lead to the adoption of a content delivery network (CDN), which would substantially reduce end-user disruption in the event of a subsequent DDoS attempt.

Reducing the stress of being on call and responding to incidents[11] is another benefit of SCE. The repeated practice of recovering from

9 P. Timmermann, "Vulnerability, Resilience, and the Collapse of Society," *Environmental Monograph* 1 (1981): 1–42.

10 Vilas Veeraraghavan, "Charting a Path to Software Resiliency," Medium, October 2018, *https://oreil.ly/fimSI*.

11 As an example, see key findings in Jonathan Rende, "Unplanned Work Contributing to Increased Anxiety," PagerDuty, March 10, 2020, *https://oreil.ly/jKewe*; see also S. C. Sundaramurthy, et al., "A Human Capital Model for Mitigating Security Analyst Burnout," *Eleventh Symposium on Usable Privacy and Security ({SOUPS} 2015)* (Montreal: USENIX Association, 2016), 347–359.

failure minimizes fear and uncertainty in the event of an incident, transforming it into a problem with known processes for solving it. The muscle memory gives teams more confidence that they can work through the problem and ensure system recovery based on their developed expertise.

Few organizations are highly confident in the security of their systems, largely due to an inability to track and objectively measure success metrics on a continual basis.[12] SCE can improve your ability to track and measure security through controlled and repeatable experimentation. SCE can also boost confidence by informing your organization of its preparedness to unexpected failures. Feedback from these scenarios can be used to improve the system's ability to be resilient over time. After repeated experimentation, your team can gain a clearer picture of the system, efficacy of security processes, and the ability to recover from unforeseen surprises.

Key Takeaways

Resilience is the foundation of chaos engineering
: A resilient system is a flexible system that can absorb an attack and change along with the evolving threat context of its environment.

Only focusing on robustness leads to a false sense of security
: Trying to prevent failure or only implementing technical solutions to block threats leads to risk accumulation that ultimately results in worse consequences when incidents happen.

SCE accepts that failure is inevitable and uses it as a learning opportunity
: By prioritizing handling failure gracefully, you can reduce remediation costs, minimize disruption to end users, and decrease stress levels during incidents—not to mention improve your organization's confidence in the stability of its production systems, understanding of systemic risk, and operational feedback loops.

12 Discussing the myriad issues with risk measurement in information security is beyond the scope of this report, but for a starting perspective on it, we recommend reading Ryan McGeehan's "Lessons Learned in Risk Measurement," Medium, August 7, 2019, *https://oreil.ly/toYaM*.

CHAPTER 2

Decision Trees—Making Attacker Math Work for You

Thinking through how attackers make choices during their operations is essential for informing security strategy at all stages of the software delivery life cycle. Understanding the attacker decision-making process can save you from excess engineering efforts to stop a niche threat and instead focus your efforts on plucking off the low-hanging fruit in your systems that attackers will try to compromise first.

As Phil Venables said, "Attackers have bosses and budgets too."[1] Just like any other project, an attack campaign is expected to generate a positive return on investment (ROI). This attacker ROI is colloquially referred to as "attacker math." Understanding the attacker math related to your individual systems and services is invaluable in helping you prioritize what security controls to implement.

In the context of SCE, attacker math additionally provides a blueprint for the type of game-day scenarios you should conduct. By thinking through the highest ROI options of an attacker, you're discovering which actions they are most likely to take—and thus illuminating what types of failure you should inject into your systems to test their resilience.

1 Phil Venables (@philvenables), "Attackers have bosses and budgets too," Twitter, September 13, 2014, 6:01 p.m., *https://oreil.ly/S8EoJ*.

Decision Trees for Threat Modeling

A threat model enumerates the characteristics of a system, product, or feature that could be abused by an adversary, and sets up systems for security success when implemented during the design phase. The model covers all issues relevant in your system, even ones that already have mitigations, because contexts change over time. More tactically, a threat model describes the system architecture in detail, the current processes used to secure that architecture, and any security gaps that exist. A threat model should include asset inventory and inventory of sensitive data, as well as information about connections to upstream and downstream services. A threat model ideally links to code bases, tools, and tests performed so anyone can better understand it and track progress.

Decision trees are a form of threat model that incorporates attacker ROI. Decision trees[2] are a visual representation of the different actions an attacker can take at each phase of an attack campaign. When wanting to compromise a specific asset, attackers have multiple avenues and methods at their disposal. Creating decision trees can help you map out these attacker choices and visualize which paths attackers are most likely to take or methods they are most likely to employ. Keep in mind that our aim here is to use adversarial thinking to model potential attacker choices for the purposes of proactive experimentation, and that there may be gaps in our model and the real-world decisions that attackers make.

Before exploring the process of building decision trees and other ways they can add value, let's walk through a tangible example of a decision tree to help frame the rest of the chapter. In this example, we will dig into how we'd craft a decision tree for an S3 bucket representing an attacker's ultimate goal of obtaining "sensitive customer data."

2 Decision trees are sometimes referred to as "attack trees" in this context.

Decision Tree Walkthrough: S3 Bucket with Customer Data

We've identified a business priority: the S3 bucket containing customer video recordings, which fits under the general attacker goal of "sensitive customer data." Our next task is brainstorming how the attacker would most easily get to their goal of accessing and exfiltrating this data. While it can be tempting to dream up scenarios where Mossad fluctuates your data center's power supply to exfiltrate data bit by bit, determining the least-cost path attackers can take to their goal is the most sensible place to start your testing.

An attacker can easily compromise an S3 bucket containing sensitive data if the S3 bucket is public rather than private, allowing the attacker to access it directly without any authorization required. While it may seem contrived, this case is similar to the most common "cloud breaches" over the past decade, affecting organizations such as Booz Allen Hamilton,[3] Accenture,[4] and Twilio.[5] It represents a lack of any security control in place—the failure to apply security controls in the first place—which has lovingly been referred to as "yolosec" by one of the authors.[6] Starting with the assumption of "yolosec" and how the attacker would take advantage of it is a sensible way to craft the least-cost path in your decision tree. In our case, the truly easiest path for the attacker would be to access the cache of the public bucket, as Figure 2-1 shows.

3 Sean Gallagher, "Defense Contractor Stored Intelligence Data in Amazon Cloud Unprotected," Ars Technica, May 31, 2017, *https://oreil.ly/ugICX*.

4 Phil Muncaster, "Accenture Leaked Data via Another AWS Misconfig," infosecurity, accessed October 4, 2020, *https://oreil.ly/pAIOA*.

5 Shaun Nichols, "Twilio: Someone Waltzed into Our Unsecured AWS S3 Silo, Added Dodgy Code to Our JavaScript SDK for Customers," The Register, July 21, 2020, *https://oreil.ly/ddaEA*.

6 Kelly Shortridge, "On YOLOsec and FOMOsec," 2020, *https://oreil.ly/wVMzh*.

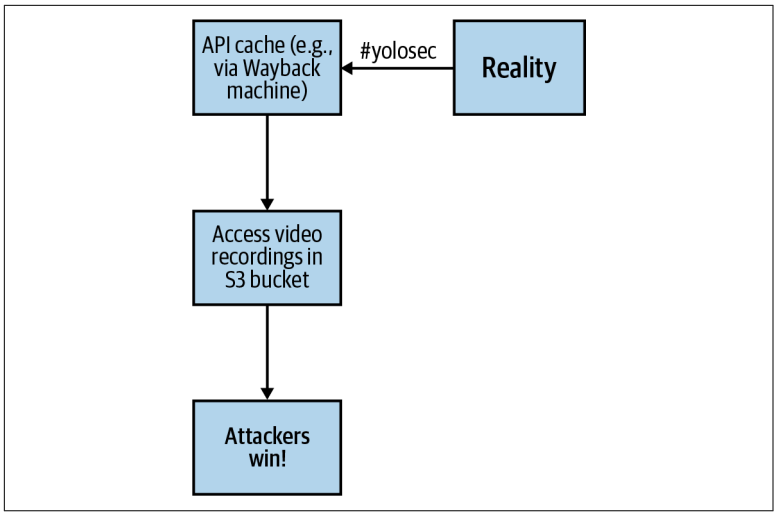


Figure 2-1. The first "branch" of the decision tree demonstrates the easiest path for the attacker to get to their goal.

Once you have the first branch of the decision tree—the easiest path for the attacker—add a new tree with the most basic defensive assumption in place. In our case, ensuring the S3 bucket uses the default of private access with access control lists (ACLs) may seem like the easiest mitigation. However, the first necessary mitigation is actually disallowing crawling on site maps (shown in dark blue in Figure 2-2) to avoid any of the data in a public S3 bucket being cached. Now consider what the attacker will do to work around that requirement. The attacker may have multiple options available to them, with varying degrees of difficulty. For instance, crafting a phishing email to steal the necessary credentials is usually an easier path than exploiting a vulnerability in the server itself.

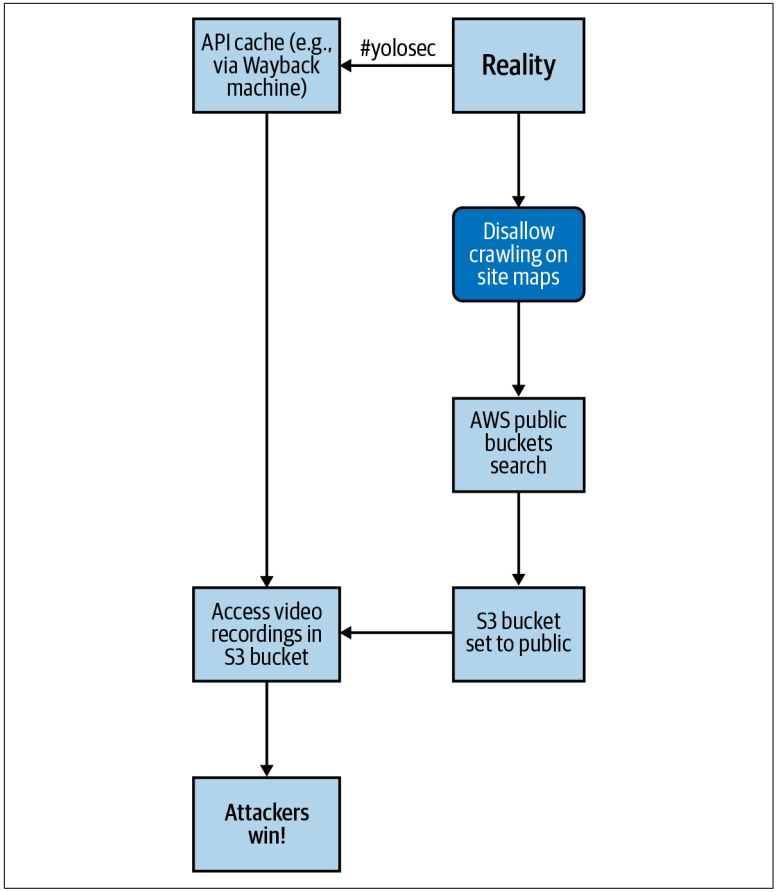


Figure 2-2. The second branch of the decision tree implements the first necessary mitigation and considers how the attacker will subsequently respond to the mitigation.

Next, you would start down the path of considering what the attacker would do if the S3 bucket was instead kept private (as is now the default in Amazon Web Services) with basic ACLs in place. The easiest branch under phishing would be to compromise user credentials that are authorized to access the web client and then see whether there are any misconfigurations that allow for client-side manipulation leading to access to the S3 bucket itself. With the right mitigations in place (shown in dark blue in Figure 2-3), the attacker will be sent back to the drawing board to pursue other phishing methods.

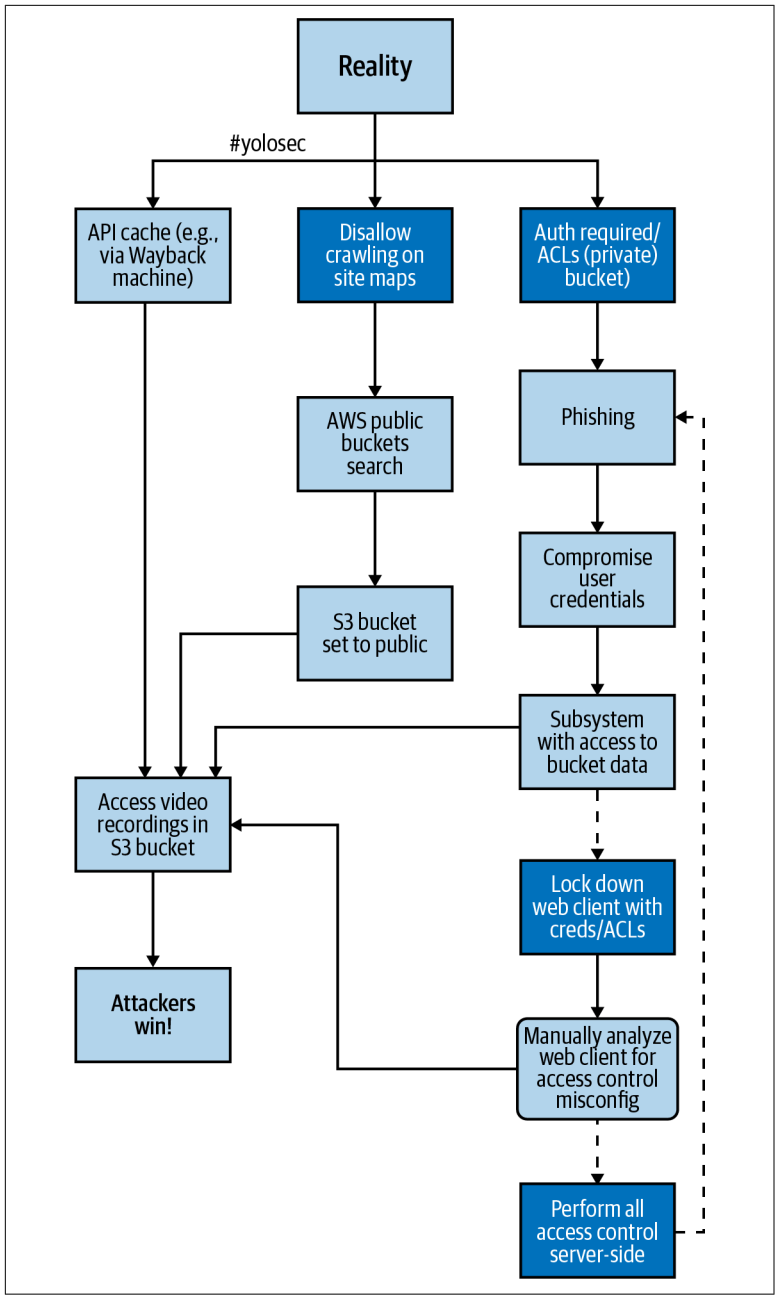


Figure 2-3. The third branch of the decision tree maps out the next initial avenue attackers will explore, what mitigations could be effective, and how the attacker will respond to each mitigation down the branch.

You and your team should continue this process of considering the attacker's easiest tactic to get to their goal, what you could implement to mitigate it, and how attackers would adjust their tactics to get around the mitigation. By the end of the exercise, the hardest path should be a string of zero-day vulnerabilities being exploited, as that almost universally reflects the hardest and most expensive set of challenges for an attacker. There are minimal defenses against zero-day exploits—after all, no one knows about them. But for most organizations, building your defenses to a level where attackers are forced to use zero days will restrict the spectrum of potential attackers to only those who are the best resourced and experienced, such as nation state actors. You are harnessing attacker math to ensure that an attack against you presents poor ROI.

In our example of customer data stored in an S3 bucket, Figure 2-4 shows how eliminating the low-hanging fruit—the easy paths for attackers—by implementing the mitigations in dark blue, eventually forces attackers onto the "0day path." Even nation state attackers will find the prospect of backdooring supply chains to be frustrating and daunting—so they won't do it unless the target goal is extremely valuable (which is a win for your organization!). The full decision tree includes potential mitigations you can deploy to thwart these actions and how attackers will respond to these mitigations, which can inform what strategies and tools you need to implement to raise the cost of attack.

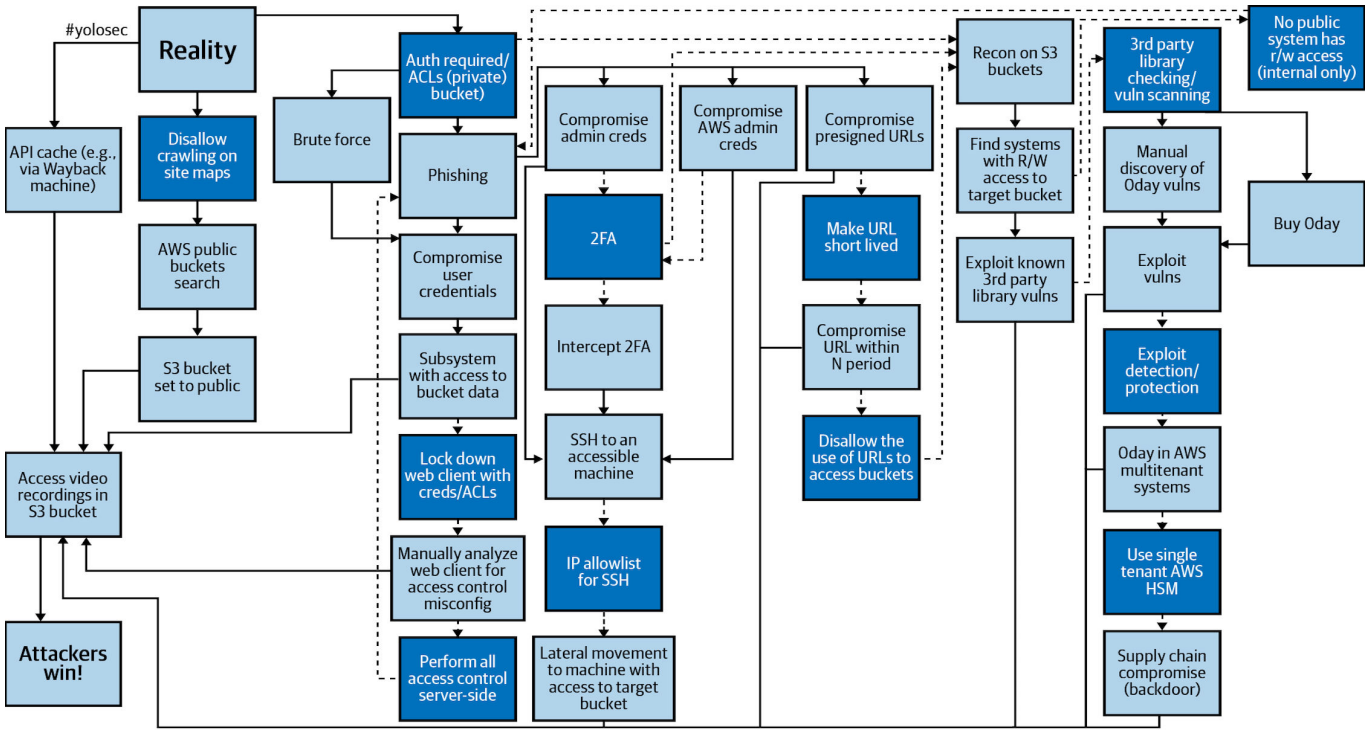


Figure 2-4. The full decision tree maps out the range of actions attackers are likely to take, from easiest path to hardest.

Outthinking Your Adversaries

As you can see in this example, an important benefit of decision trees is that it helps you think ahead of your adversary's decision-making. Decision trees encourage you to perform second-order thinking—or, even more powerfully, n-order thinking. First-order thinking, which is usually the default mode of operation for human brains, considers the obvious implications of an action. N-order thinking, in contrast, considers cascading implications. If you think in terms of a game like chess, n-order thinking considers how decisions made in the current round will impact future rounds.

The n-order thinking process is the essence of belief prompting.[7] Belief prompting is a tactic to foster strategic thinking by encouraging people in a competitive setting to articulate their beliefs about their opponent's probable actions. Experimental evidence suggests that, by thinking about how your opponent will respond to whatever move you are about to make, you will make a substantially smarter choice.[8] Even seeing a visual representation of the potential sequence of moves (referred to as an "extensive-form game tree" in game theory lingo) within a competitive scenario is shown to encourage more rational choices—that is, choices that maximize your chances of winning.[9]

Not only does the exercise of creating the decision tree force you to articulate your beliefs about your opponents' probable moves in response to your own, but the resulting graph serves as an ongoing visual reference to inform more strategic decision making. When applied to information security, belief prompting involves the defender repeatedly asking themselves what the attacker would do in response to a defensive move. For instance, if the defender blocks outbound direct connections, an attacker might use SSL to hide their traffic. Whether you perform belief prompting or craft decision trees, these are some of the key questions to ask:

---

7 Colin F. Camerer, Teck-Hua Ho, and Juin Kuan Chong, "Behavioural Game Theory: Thinking, Learning and Teaching," in Steffen Huck (ed.), *Advances in Understanding Strategic Behaviour: Game Theory, Experiments and Bounded Rationality* (New York: Palgrave Macmillan, 2004), 132, *https://oreil.ly/yZd6r*.

8 Camerer, Ho, and Chong, "Behavioural Game Theory."

9 A. Schotter, K. W. Weigelt, and C. Wilson, "A Laboratory Investigation of Multiperson Rationality and Presentation Effects," *Games and Economic Behavior* 6, no. 3 (1994): 445.

- Which of our organizational assets will attackers want? Enticing assets to attackers could include user data, compute power, intellectual property, money, etc.
- How does the attacker decide upon their delivery method, and how do they formulate their campaign?
- What countermeasures does an attacker anticipate encountering in our environment?
- How would an attacker bypass each of our security controls they encounter on their path?
- How would an attacker respond to our security controls? Would they change course and take a different action?
- What resources would be required for an attacker to conduct a particular action in their attack campaign?
- What is the likelihood that an attacker will conduct a particular action in their attack campaign? How would that change based on public knowledge versus internal knowledge?

Getting Started on Your Own Decision Tree

To craft your own decision tree, start with the business priority that is at risk. This business priority (most often a resource, service, asset, etc.) will represent the attacker's ultimate goal. There are, of course, a whole range of goals attackers might have across the entirety of your organization's technology footprint, but you should focus on the largest material risk to your organization—the type that would need to be disclosed to customers, the board, or shareholders.

A simple prioritization matrix, as shown in Figure 2-5, can be a valuable precursor to building your decision tree by simplifying the prioritization problem. By considering the value of an asset to both the organization and the attacker, you can sort out which attacker goals most merit a decision tree.

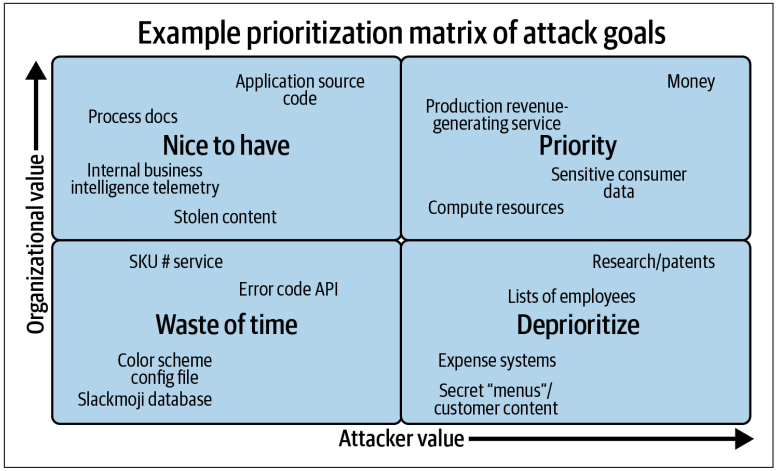


Figure 2-5. Example prioritization matrix of assets relative to attacker value and organizational value.

Importantly, the matrix in Figure 2-5 considers the *ultimate* goals in an attack, not assets that can be used as footholds. For instance, a list of employees isn't directly tied to revenue for the organization. The list of employees becomes valuable when used to inform who to target in a phishing campaign but isn't so valuable in and of itself. In contrast, money as an asset always matters to both sides of the attack equation—and, ultimately, the impact of most organizational assets being compromised is due to how it affects money. Reputational damage translates into lost revenue. A cryptominer translates to higher compute costs for an organization. Thus, a simple way to think of this matrix is on the spectrum of "how directly does this make money for the organization?" and "how monetizable is this for attackers?"

When figuring out where to start with your decision trees, focus on the "Priority" box in the matrix. Each of those assets represents a different decision tree that should be crafted. A decision tree that tries to bundle multiple ultimate goals at once will suffer from excess complexity and visual messiness. Aim to choose a specific example of one ultimate outcome. For instance, "sensitive customer data" is a priority goal, but enumerating all the attack paths to every place that data resides in your organization would suffer from complexity overload. Instead, choose something like our prior example of "an S3 bucket containing customer video recordings," which has a specific location and characteristics for more easily mapped attack

paths. Once the specific decision tree is complete, you can extrapolate the mitigations to apply to other resources matching that goal type.

After you've covered the key resources within the "Priority" box with decision trees, you can move on to the "Nice to haves." These resources and assets more directly contribute to revenue but are not as easily monetizable by attackers, like business intelligence metrics or documentation around company processes. After crafting decision trees for those, you can optionally move to the "Deprioritize" box, which is full of items that are valuable to attackers but are only abstractly or indirectly contributing to organizational revenue. These deprioritized resources and assets can include expensing systems, secret menus (think of Netflix's list of secret codes), or even speculative research.

The example resources and assets we included in each box are for illustrative purposes only. While attacker value stays reasonably constant across industries, it's up to you to decide which resources are most valuable to your organization! Work with your colleagues in business units or in the finance department to understand what drives your organization's revenue and operations. In our view, each matrix will be unique for each company, and thus the decision trees prioritized by each company will likely be different (although some resources containing money or sensitive customer data will likely persist across many industries).

Incident Retrospectives

While a decision tree is a useful threat-modeling tool for informing security improvements during the initial design phase of a feature or system, it also shines when used as a tool for continuously informing and refining strategy. When thinking through changes to the system or feature, you can consult your decision tree to see how the proposed changes would alter the relevant attacker math.

During incident retrospectives, pull up your relevant decision trees and see where your assumptions held true or false. Some of the questions you should be asking as a team include the following:

- Did the attackers take any of the branches modeled in the tree? In our example, perhaps log data from the API wayback showed repeated access attempts from the same IP address over a short period of time—suggesting the attacker began by trying the easiest path.
- Did the mitigations alter the attacker's path as anticipated? In our example, maybe implementing 2FA forced attackers to attempt vulnerability exploitation instead.
- What attacker options were missing from your assumptions? Perhaps your application monitoring data shows the attacker tried to exploit a vulnerability in a separate application in an attempt to gain a foothold on your Amazon Web Services (AWS) infrastructure.
- Did the attacker encounter any stumbling blocks that you didn't realize served as mitigations? In our example, maybe your use of immutable containers in your AWS infrastructure made it difficult for attackers to find a foothold via Secure Shell (SSH) and move laterally.

In addition to incident retrospectives, decision trees are useful for testing and experimentation. In the old school model, you can provide your red team (or penetrating testers) with a decision tree that highlights which functionality should be targeted. In SCE, your decision trees inform the failure scenarios you want to test. Starting your testing at each functionality point along the easiest branches lets you continuously verify that any low-hanging fruit for attackers is sufficiently eliminated—that the "basics" are properly covered. Once you feel confident in the resilience of your systems to the more obvious security failures, you can move to the branches that require higher attacker effort. Only after an extensive period of security chaos experimentation and strategy refinement based on the resulting feedback loop should you move on to tackling the "0day branch." While chaos testing the toughest threats can feel exciting, it won't do much to help your organization if easier attack paths are left untested.

Key Takeaways

Attackers will choose the easiest, low-cost path when possible
: Attackers care about the ROI of their attack and prefer using "dumb" tactics (like credential phishing) to reach their goal whenever possible. Relatively expensive attacks, like exploiting a zero-day kernel vulnerability, will only be used when necessary, as the resource expenditure (and risk) is often not worth the reward.

Decision trees allow us to conceptualize and visualize attack paths
: Mapping out the different paths attackers can take in a specific service or system to reach their goal—from what's easiest for the attacker (#yolosec) to what's hardest (0day)—provides visual guidance on where mitigations are needed and what scenarios to use in your chaos tests.

Raising the cost of attack is the key to security success
: Eliminating low-hanging fruit, by concentrating your mitigation and testing efforts on the easiest attack paths, forces attackers to invest more resources to reach their goal—which increases the likelihood that either they won't bother or they'll conduct harder operations that are "noisier" and thus easier to catch.

Leverage feedback loops for continuous refinement
: Decision trees can and should be used on an ongoing basis to inform your security strategy. They can represent a starting point for conducting experiments in SCE to learn from failure.

CHAPTER 3

SCE versus Security Theater—Getting Drama out of Security

No one disputes that the security of our data and systems is important. So why are security goals often rendered secondary to other goals? Traditional security programs commonly add friction to work being conducted (beyond just software development!), requiring organizational users to jump through extra hoops to achieve their goals. Security thereby serves as a gatekeeper—whether requiring user interaction with security tools for account access or making their rubber stamp essential in approving software releases. Of course, businesses need to make money, so anything standing in the way of making widgets or delivering services to customers is, quite understandably, placed at a lower priority.

The solution seems simple—just ensure security programs are enabling the business rather than slowing it down! But, as with many things in life and in technology operations, it's far easier said than done. Some compliance requirements, such as access controls stipulated by the Payment Card Industry Data Security Standard (PCI DSS) and the Health Insurance Portability and Accountability Act (HIPAA), are nonnegotiable if an organization wants to avoid fines and reputational damage. There can be ethical judgments—like preserving privacy over monetizing as much user data as can be collected—that are also irreconcilable. However, the constraints espoused as valuable by traditional security programs are often artificial in nature—a product of status quo bias ("this is how it's always been done, so it must be right") and an all-or-nothing mentality.

The principles of SCE lay the foundation for a security program that inherently enables the business and aligns with the understanding that security must be as convenient as possible. By embracing failure, you not only consider what types of failure will impact the business the most but also gain a deeper understanding of the systemic factors that lead to failure. Because of this, it is difficult to stick to absolute notions of what "must" be part of your security program, instead focusing on what works in practice to reduce security failure.

In this chapter, we'll explore the differences between security theater and an SCE approach, including security approval patterns that enable software delivery while still keeping it safe.

Security Theater

Unfortunately, one pillar of the ineffective status quo of security as a gatekeeper is "security theater"—work performed that creates the *perception* of improved security rather than creating tangible and positive security outcomes. When you perform security theater, you're optimizing, unsurprisingly, for drama! You focus on the "bad apples" cases, crafting processes that affect everyone just to catch the few who may do something silly or malicious. Underpinning these theatrics is the philosophy that "there might be someone who cannot be trusted. The strategy seems to be preventative control on everybody instead of damage control on those few."[1] Naturally, this makes the situation less tolerable for everyone—and this is seen in other areas of tech already, not just information security. But security theater is what fuels the reputation of enterprise information security teams as the "department of no" and sets up an antagonistic relationship between the security team and the rest of the enterprise.

In our discussion of security theater, we'll be leveraging Jez Humble's work on "risk management theater." As Humble noted, this theater —whether information security, risk management, or physical security—is a "common-encountered control apparatus, imposed in a top-down way, which makes life painful for the innocent but can be

1 Bjarte Bogsnes, *Implementing Beyond Budgeting: Unlocking the Performance Potential* (New Jersey: John Wiley & Sons, 2016), 10.

circumvented by the guilty."[2] This notion of top-down control is fundamentally incoherent with the need to adopt adaptive security to coevolve with the changing environment around you. Top-down control—what we (not so) affectionately call "gatekeeping"—may create a *feeling* of security, but it is less effective at achieving it than a systems-based approach that embraces experimentation and feedback loops.

How does this security theater look in practice so you can spot the red flags in your own organization? How does the drama-free approach look in contrast? Table 3-1 outlines the core differences between the SCE approach discussed in this report and the traditional security theater approach that creates security-as-gatekeepers in the enterprise. There are overlaps between this comparison and Jez Humble's comparison of adaptive risk management versus risk management theater, which is to be expected in applications of resilience to IT domains (especially when it comes to cultural concerns).

Table 3-1. The principles of SCE compared with those of security theater

SCE	Security theater
Failure is a learning opportunity. Seeks to uncover failure via experimentation as feedback for continual refinement. When incidents occur, conducts a blameless retrospective to understand system-level dynamics to inform improvement.	**Increases stringent policies and technology to reduce bandwidth for human failure.** Seeks to prevent humans from causing failure through policy and process enforcement. When incidents occur, investigates who is responsible and punishes them.
Accepts that failure is inevitable and blame is counterproductive. Systems are continually changing, and it's impossible to always be aware of all known and unknown sources of failure. Supports human decision-making by evolving process, technology, and knowledge through continual experimentation and feedback loops.	**Blames humans as the source of the problem.** If people always adhere to security policies and procedures, nothing bad can happen. Controls are put in place to manage "stupid," "lazy," or "bad apple" humans. Ignores the fact that policies and procedures are always subject to interpretation and adaptation in reality.

2 Jez Humble, "Risk Management Theatre: On Show at an Organization Near You," Continuous Delivery, August 5, 2013, *https://oreil.ly/1oBQZ*.

SCE	Security theater
Uses experimentation and transparent feedback loops to minimize failure impact. Understands real-world constraints and thus the value of continual testing and refinement to promote adaptation versus rigid policy enforcement. Prioritizes the ability to recover from incidents and learn from them over finding a place to pass blame.	**Uses policies and controls to prevent failure from happening.** The assumption that security policies will be followed allows blame postincident on failure to adhere to policy. New controls serve as proof that "something was done" postincident. Adaptation is limited, and unknown sources of failure emerge due to the inability to always follow global-level policies when performing local-level work under competing pressures.
The security team operates collaboratively and openly. Risk acceptance decisions are made by the teams performing the work, not the security team. Security shares its expertise across teams as an advisor and facilitates knowledge sharing across the organization.	**The security team operates in a silo.** Knowledge is often tribal—undocumented and unshared. Duties are segregated between nonsecurity versus security. Security is removed from the work being conducted but approves or denies changes. The act of security approval itself is seen as creating security.
Incentivizes collaboration, experimentation, and system-level improvements. Both engineering and security teams collaborate to improve system-level metrics like time to restore. No rewards for blocking change on an individual or function level. Accepts that locally rational decisions can lead to system-level failures.	**Incentivizes saying "no" and narrow optimization.** The security team creates barriers that discourage technology adoption that could increase operational quality or performance. Or the product security team optimizes for (perceived) stability at the expense of velocity.
Creates a culture of continuous learning and experimentation. Encourages blameless retrospectives after incidents and open discussion of mistakes or oversights. Promotes experimentation to generate knowledge, with the goal of system-level improvements. Teams feel safe raising issues transparently and early to the security team.	**Creates a culture of fear and mistrust.** Encourages finger-pointing, passing the blame, and lack of ownership for mistakes or oversights. Hiding issues is preferred, as raising issues is punished. Antagonistic vibe between the security team and other teams.
Is principle-based and defaults to adaptation. Guiding principles are applied to new, unfamiliar situations. There is an openness to rethinking and refining security strategy for new or evolving situations.	**Is rule-based and defaults to the status quo.** When new technologies (e.g., microservices) or processes (e.g., CI/CD) are encountered, rules are rewritten and the same security controls from prior tech/processes implemented (e.g., "container firewalls").
Fast, transparent security testing. Integrates with development workflows across the software delivery life cycle. Provides fast, digestible feedback to engineers on security issues. Facilitates threat models during the design phase and continuous refinement of them as experimental data is generated.	**Manual security reviews and assessments.** Releases are blocked until security approvals are received. Reviews are performed by a separate team, often in a separate location with limited information sharing. Delayed and siloed feedback on system-level security impacts.

As you can see, SCE is all about outcomes rather than output, prefers psychological safety to ruling with an iron fist, and experiments with strategies optimized for the real world, not an ideal security-is-the-only-priority world. The simple premise of achieving tangible outcomes rather than dramatizing effort should be compelling to all stakeholders involved in enterprise security—from the security teams themselves to product and development teams, too (not to mention company leadership, which can start to see improved outcomes from security budget!).

The SCE approach is also more in tune with how product and development teams already think. Testing for failures crosses over considerably with teams wanting to test for bugs that could lead to performance issues. Many of your teams may already proactively discover bugs or other issues and think of how to improve their processes or tech stack if the current state isn't producing results. The shift in thinking required by the SCE approach, therefore, is pretty minimal: you're just assuming that failure exists as an emergent property of the system as a whole rather than assuming the security team checked everything perfectly.

It may seem like we're especially harsh on security policies and processes, and that they're all inherently bad. That's certainly not the case! As we'll discuss later in this chapter, intelligently crafted processes can facilitate change while still derisking it. The issue is that policies and procedures shouldn't be overly relied upon to prevent all possible badness from happening. Policies shouldn't represent punishment devices but instead encourage continuous learning, experimentation, and adaptation. Procedures shouldn't serve as "gotchas" that, if not followed, can be used as supporting evidence of someone being fired after a security incident. Anything that ignores human behavior or the workflows of the people to which the policy or procedure applies is doomed, ironically, to fail.

This is precisely how security theater can create a self-fulfilling prophecy. If everyone is treated as a potential source of error at which fingers can be pointed, it becomes more likely that mistakes will be covered up, concerns kept quiet, and thus feedback loops dampened. And without feedback loops, it's difficult to continuously improve your security strategy—holding everyone back.

Moving toward feedback loops also reflects a move toward measurement, which admittedly is daunting for many information security

teams, who are typically evaluated based on outputs rather than outcomes. If you want to ruin a CISO's morning, ask them how they quantify the monetary benefit generated by the dollars spent on their security program. Calculating an ROI in security is a notoriously fraught exercise, which is largely seen as a pipe dream. In fact, a hallmark of security theater, or any domain-specific theater, is that benefits cannot be directly tied to costs—that is, the ROI is nebulous.

If you establish success criteria for each product you purchase and continue to measure its efficacy over time across a few different dimensions, any inutility should become apparent. It can admittedly be nerve-wracking to migrate from a state in which you can spend money largely at your discretion to one in which you're held accountable for each project or strategic pursuit. To a certain extent, there's been safety in the herd among enterprise security teams: because ROI in security is expressed as being nearly impossible to calculate, enterprise security teams aren't expected to calculate it. However, that kind of fatalism won't work in a world where security enables the business. The security team must either be provably avoiding a loss of money or facilitating the gain of additional money (for instance, if strong security becomes a competitive differentiator and contributes to sales success).

The ROI of a security program involves more than the direct connection between dollars spent and security benefit gained. Your security program's negative externalities—any activity you conduct that negatively affects someone other than you—are relevant, too. Just as airport security theater created the unintended externality of increasing road fatalities,[3] we must consider what negative externalities digital security theater engenders. What amount of business activity is lost due to the friction that security theater creates? The business costs of inconvenience are rarely quantified by security teams, and they aren't used as a metric to inform decision-making. Rather than exclusively looking at security metrics, look at the metrics of other teams in tandem to spot possible negative externalities. For instance, if your security review code coverage is increasing, check whether the engineering team's lead time or deploy-frequency

3 Garrick Blalock, Vrinda Kadiyali, and Daniel H. Simon, "The Impact of 9/11 on Road Fatalities: The Other Lives Lost to Terrorism," *Journal of Law and Economics* 41, no. 14 (Feburary 2005), 1–19.

metrics are being impacted. While anyone who took a Statistics 101 course will remember that "correlation does not equal causation," a worsening metric on the engineering side at the same time as an improving metric on the security side is a starting point for investigation. Maybe the engineering team decided that playing a newly released video game was way more interesting than pushing code! By speaking with those teams and investigating, you can validate the problem is driven primarily by the video game, not the security program. If the security program *is* causing a negative externality in software delivery performance, then your investigation can help identify improvements to reduce friction.

The good news is that the security team doesn't have to achieve this transition away from security theater alone. Unfortunately, traditional enterprise security teams largely operate in silos. As shown in Table 3-1, operational silos are a component of security theater and hold us back from gaining situational awareness across the business, including the teams actually building the software being delivered. Part of reducing gatekeeping is to ensure that expertise does not reside solely in one place. This expertise should also not be limited to being leveraged at only one point during the software delivery process (like just before a release). To achieve this, security should be embedded on product teams rather than siloed, becoming "enablers" rather than "doers." To explore how this sort of model evolves the security approval process for changes, let's go through SCE's approval patterns.

Security Approval Patterns

We can plainly see that security theater achieves a lot of drama and toil, without many results. SCE, in contrast, suggests a healthier mode of operation that eliminates gatekeeping. However, as organizations discard gatekeeping as a security strategy, they're often stuck on how security approvals should take form. If you don't want to be a blocker, do you discard approvals altogether? But doesn't that lead to chaos (of the bad kind)? Taking a closer look at these questions, we can see that there's a perceived tension between stability and the type of speedy, experimental, and collaborative culture SCE fosters.

While some organizations believe stricter change management processes—including security approval processes—lead to greater stability, the data from Dr. Forsgren's Accelerate surveys show the

opposite is true. Organizations with the fastest deployment frequency and shortest lead time for changes *also* exhibit the lowest change failure rates and shortest time to restore service. In fact, the organizations you might think would be the most stable—those with more restrictions and requirements in place before code is deployed or changes are made—experience the highest change failure rates. Between 46% and 60% of changes deployed by the most conservative and, in theory, "careful" organizations result in degraded service and ultimately require remediation.[4] Clearly, the more hoops through which you need to jump, the more opportunities there are to fall on your face.

Now think of this finding from the security perspective. Let's say you need to deploy a patch for a remote code execution vulnerability. A cumbersome change management process will hold up deployment of the patch, which will directly put security and stability at risk. A nimble process, in contrast, allows you to make necessary changes faster and thus more continuously improve system stability. It even makes rollbacks easier, so if a patch goes wrong, it can be reversed more quickly than in heavy change management paradigms. Ensuring your systems are easier to change, improving testing, and building strong monitoring capabilities all reduce the cost of change management and ultimately result in far stronger security than that created by layered walls of bureaucracy and approvals.

Another danger of heavy layers of security review—the equivalent of change approval boards (CABs)—is that they provide a nice little hiding spot for accountability. Rather than performing the "thinky thinky" around smoothly implementing high impact changes up front during the design phase, such considerations are left for the external approver (like a security team or CAB board). After all, why waste time brainstorming how to implement changes if external approvers will take days to weeks to approve those changes, protracting the release timeframe?

4 Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle, *Accelerate State of DevOps Report 2019*, DORA, 2019, *https://oreil.ly/UaVfE*.

This brings us back to theater. This time, it's "approval theater"[5]—acting out the part of thoughtful change inspection. Unfortunately, both security approvers and CABs often serve as actors in approval theater rather than achieving the outcome of higher quality changes. External security approvers and CABs also mean that you miss out on feedback loops. By handing over inspection of changes to an external authority, teams miss out on discovering problems in quality that could inform improvements to their design or processes. This doesn't mean approvers are always useless. As noted by Scott Prugh,[6] CABs can be useful as notifiers and mediators, but teams still must be in charge of owning their changes.

Transparency is another core element of productive change processes. Clear change processes are shown to positively impact software delivery performance, while heavyweight change processes negatively affect it. Treat security review processes similarly. Is your security review process clear, or is it heavy? Does the security team share what the requirements are to pass, or are requirements decided on a per project basis, based on subjective judgment? Is there documentation around the steps within the security review process? Is there transparency on how security is thinking through their review, or, from the perspective of the product team, does the security team seemingly conduct their review in a high-walled fortress that allows no visitors?

Ultimately, requiring any external approval of a change or release will create a bottleneck—and a bottleneck inherently will add friction, reduce efficiency, and erode accountability. Bottlenecks also incentivize batched work. Engineers, anticipating the bottleneck, are thereby nudged toward larger chunks of work—because smaller batches would lead to a longer queue in the bottleneck. Unfortunately, these large batches involve more variables, and increase the likelihood that there will be defects and other unwanted variability.[7] Bottlenecks also fuel a fear of change, which inhibits learning and

5 Ron Forrester, et al., *Breaking the Change Management Barrier: Patterns to Improve the Business Outcomes of Traditional Change Management Practices*, IT Revolution, September 10, 2019. *https://oreil.ly/fuLuU*.

6 Scott Prugh, "CAB is useless," Twitter, October 23, 2018, 8:26 p.m., *https://oreil.ly/KRlqZ*.

7 Donald G. Reinertsten, *The Principles of Product Development Flow: Second Generation Lean Product Development* (Redondo Beach, CA: Celeritas, 2009).

thus stymies improvement in change processes. This is precisely why SCE is incompatible with external approval processes—because creating hypotheses, conducting experiments, and learning from the results is essential to SCE.

Instead, there must be a partnership, where the ultimate goal (the problem being solved) is the anchor for the relevant stakeholders to judge quality. In this partnership, the security team should function as an advisory group, not a decision authority.[8] It's important to include the teams doing the work in any change process so they can be in the continuous improvement loop as well. This means that security teams should empower other teams to help build and participate in new change processes. The status quo in security is to dictate terms and give directions to the teams making the changes, which doesn't foster accountability. By working together toward a solution, you foster accountability, which ultimately incentivizes higher quality changes.

To ensure processes are transparent, security reviews should be recorded and traceable. Optimize security tests or reviews for "just enough" to ensure you're sufficiently confident in the security of production systems—where "just enough" is heavily dependent on your organization's security and compliance requirements. This will also encourage you to create a security review process that is based on metrics and identifiable standards rather than opinion or emotion. For instance, instead of heavyweight processes, your team can promote frequent, automated testing (on smaller release candidates). In fact, by adopting SCE, you can think of this evolution from heavy change management processes to smaller tests and reviews as migrating to a series of experiments. This helps reduce the queue size of security reviews, which, as per Little's law,[9] will help minimize lead time—a key metric by which operational performance is measured.

8 Forrester, et al., *Breaking the Change Management Barrier*.

9 John D. C. Little and Stephen C. Graves, "Little's Law," in *Building Intuition: Insights from Basic Operations Management Models and Principles* (Boston: Springer, 2008), 81–100, *https://oreil.ly/i9fB8*.

Key Takeaways

Security theater optimizes for gatekeeping, not continuous improvement

Procedures that are meant to punish individuals for perceived failures suffocate the ability to learn from failure and discourage experimentation in the first place.

Stability and speed aren't at odds, and heavy security approvals don't promote stability

Relying on security teams as an external approver creates bottlenecks, which incentivize large batches of work and increase the time it takes to deploy changes—both of which jeopardize stability.

SCE approval patterns favor localized decision-making and lightweight advisory

By ensuring relevant teams are involved in the development of approval patterns, designing processes to facilitate localized change-making, and using the security team as an advisor, you encourage accountability among the teams creating the changes.

CHAPTER 4
Democratizing Security

Imagine if every software engineer in your organization was a former attacker. They could look at their team's feature or product and quickly brainstorm how they could benefit from compromising it and what steps they would take to most easily do so. After daydreaming this attacker fantasy for a bit, they could snap back to reality and propose design improvements that would make it harder for attackers to take the steps they imagined. While having this kind of feedback loop on each one of your engineering teams may seem like its own fantasy, it's more easily realized than you imagine.

A distributed, democratized security program can accomplish these goals. What do we mean by making defense "democratized"? It represents a security program supported by broad, voluntary participation with benefits accessible to everyone. It means that security efforts are explicitly neither isolated nor exclusive. Like a democracy, it must serve all stakeholders and involve participation by those stakeholders. Specifically, an organization's team of defenders can't just consist of security people—it must also include members of product and engineering teams who are building the systems whose security the defenders must challenge.

In this chapter, we'll explore what the critical function of defenders —alternative analysis—entails and how a democratized security program, such as a Security Champions program, fits into SCE.

What Is Alternative Analysis?

Before we dive into how democratized security is enabled by SCE (and vice versa!), we should probably answer the question "What is alternative analysis?" In modern information security, the function of alternative analysis is typically performed by the red team, for which we can borrow a definition from the military domain: "Red teaming is a function that provides commanders an independent capability to fully explore alternatives in plans, operations, concepts, organizations, and capabilities in the context of the operational environment (OE) and from the perspective of partners, adversaries, and others."[1]

Of course, there aren't "commanders" in the enterprise (as much as some CISOs, CIOs, and CTOs perhaps wish they were!), but swap that term out for "team leaders" and the "operational environment" for your "organizational environment." The end goals—which will underpin the goals of a democratized security program—include improving your organization's decision-making by broadening the understanding of its environment; challenging assumptions during planning; offering alternative perspectives; identifying potential weaknesses in architecture, design, tooling, processes, and policies; and anticipating the implications—especially around how your adversaries will respond—of any potential next steps.[2]

Fundamentally, red teams should be used to challenge the assumptions held by organizational defenders (often called the "blue team")—the practice which is known as alternative analysis. This idea is centuries old[3] but essentially revolves around anticipating what could go wrong, analyzing situations from alternative perspectives to support decision-making, and psychologically hardening yourself to things going wrong.

---

1 *University of Foreign Military and Cultural Studies: The Applied Critical Thinking Handbook (formerly the Red Team Handbook)*, vol. 5 (April 15, 2011), p. 1, *https://oreil.ly/HLuEc*.

2 Toby Kohlenberg, "Red Teaming Probably Isn't for You," SlideShare, October 27, 2017, *https://oreil.ly/7cAbZ*.

3 "The Roots of Red Teaming (Praemeditatio Malorum)," *The Red Team Journal*, April 6, 2016, *https://oreil.ly/rg4k9*; Valeri R. Helterbran, *Exploring Idioms: A Critical-Thinking Resource for Grades 4–8* (Gainesville, FL: Maupin House Publishing, 2008).

Applying alternative analysis to our digital world today, thinking through how things can be broken can help you build things more securely. Challenging assumptions underpinning your defensive strategy uncovers flaws and weaknesses that could otherwise promote defensive failure. And, as in the ancient stoic tradition, ruminating on how failure can manifest from your current system state makes you more psychologically resilient to failure, too.

Critically, alternative analysis is *not* about defenders employing method acting to fully embody the role of a real attacker. It's not helpful for a red team to sit separately from everyone else and focus exclusively on pwning systems, taking delight in uncovering failures in the enterprise's security program. This may be a ton of fun for them! But such a narrow focus creates a shallow feedback loop. If red teams aren't helping to level up shared knowledge, they aren't performing true alternative analysis and are, in essence, just hunting for sport.

Alternative analysis is a powerful tool in the SCE arsenal, but how should it be implemented? A democratized security program (like the one detailed in "The Security Champions Program" on page 37) offers a pragmatic, collaborative solution toward attaining the benefits of improved organizational decision-making.

Distributed Alternative Analysis

As you might imagine, red teams are expensive. The types of professionals who possess the necessary attack expertise combined with critical thinking skills are in hot demand. Luckily, SCE can provide a form of automated red teaming, as it continually challenges the assumptions you hold about your systems. It helps you view your systems from an alternative perspective, by subjecting those systems to failure.

However, building a distributed team dedicated to viewing problems from an adversary's perspective (or, more generally, from alternative perspectives) is complementary to an SCE approach, and can further enhance your organization's defensive decision-making. The devil's advocacy that such a team provides can help you refine your SCE experiments and determine what automated testing is appropriate. But how can you accomplish this on a tight budget?

A rotational red team program can teach engineers how attackers think through the (usually) fun practice of compromising real systems, toward the end goal of distributing alternative analysis capabilities across your product teams. Think of this like spreading the seeds of "attacker math" knowledge across your engineering organization. You can populate your engineering teams with fresh security insight that can improve decision-making. And your security teams, by virtue of interacting with these engineers, will be exposed to new perspectives that can improve how the security program serves the organization.

How is the rotational red team composed? The red team should be anchored by leaders with attack experience (ideally with a genuine appreciation for organizational and development priorities, too). Crucially, they should be aligned with the foremost goal of contributing alternative analysis to improve organizational decision-making. Each leader should be capable of training the team members about alternative analysis and attack methodology. The team itself should consist primarily of engineers from each product team who can learn—and practice—how attackers think.

If your organization has a lot of different products or teams, we recommend starting with engineers from a subset of teams that exhibit the most interest in participating in the program. As Dr. Forsgren's State of DevOps research shows,[4] the two approaches that work best for fostering change are communities of practice and grassroots. Communities of practice involves groups that share common interests and foster shared knowledge with each other and across the organization—very much in line with the overall rotational red team program's raison d'être. A grassroots approach involves small teams working closely together to pool resources and share their success across the organization to gain buy-in—which, in the rotational red team case, could start with the security team and one or two product teams eager to build in security by design.

The goal of the training is for engineers to immediately provide value vis-à-vis alternative analysis once they rejoin their product team full time. Of course, you don't want your program to slow down development long term, so you should carefully consider what constitutes a sustainable portion of the engineer's time that is

4 Forsgren, Smith, Humble, and Frazelle, *2019 Accelerate State of DevOps Report*.

dedicated to learning attacker math. Optimize for what will help level up the engineer's ability to perform alternative analysis, while ensuring that they are not out of the loop on their product so long that they must relearn the system context.

As foreshadowed in Chapter 2, decision trees are invaluable tools to support democratized alternative analysis. Engineers participating in the rotational program will be leveling up their attacker math skills by having to apply that math to the real-world systems they're usually building or operating. The training, in essence, involves engineers creating their own branches in the tree as they conduct their attack to achieve a particular goal. Ideally, they will document the actions they took in a tree format to share with their product team postrotation. As a result, participants can go back to their product teams equipped with newfound expertise to inform what attack branches should be included on the trees. For existing decision trees, participants can poke holes in existing assumptions and propose amendments to the branches. With this experience, they can leverage the attacker mindset to consider how an attacker would respond to a proposed security control or mitigation—helping fuel second-order (and beyond!) thinking to improve the team's decision-making.

The most promising implementation of the democratized security model is via a Security Champions program. We'll now turn to a high-level overview of how this program looks in practice and what benefits it brings to organizations.

The Security Champions Program

Authored by Vladimir Wolstencroft, Staff security engineer at Twilio

A Security Champions program is a set of processes involving one or more existing members of an engineering team taking on additional responsibility in the domain of security. The goal of a security champion is to create a more accurate and consistent communication pipeline between their engineering team and the security teams. As a result, organizations gain a significantly improved understanding of the risk they are taking on through new product development and feature design.

If executed correctly, many tasks of security engineers can be handed off and handled within the product teams themselves, and

the security engineering team simply guides and leads the champions through complex decision-making. The Security Champions program promotes the continuous exchange of information about each product, direction, and systems interconnectivity. It also spreads knowledge around security practices to product teams, which the security champion ensures are understood and followed.

At Twilio, the Security Champions program allows the integration of security features into products at a much faster cadence. Security champions are now the point of contact when it comes to reporting and resolving bugs and other security issues. Mitigation efficacy is improved, too, along with the speed at which mitigations are introduced. Fewer bugs reach SLA limits and instead get resolved within defined time frames, allowing for risk to be better controlled.

After running the program for a few years at Twilio, engineers on different product teams actively volunteer for and ask to engage in the Security Champions program. They now see security challenges in the same way as load-balancing challenges or programmatic bugs: as part of their everyday responsibilities. Being a member of this program and continuously meeting with a security partner allows them to solve these challenges quickly and efficiently.

Key Takeaways

Considering alternative perspectives supports stronger security outcomes
: Provide constructive criticism during planning and operation, and emulate adversaries in both thinking and action.

A rotational red team program can democratize attacker math
: Within your organization, challenge existing assumptions and support second-order thinking to improve security decision-making.

A Security Champions program empowers engineering teams
: Engineering teams make security decisions for the systems they build and operate, under guidance of the security team. Organizations gain greater efficiency and improved security outcomes, and security challenges are considered part of the everyday responsibilities of engineers.

CHAPTER 5

Build Security in SCE

SCE helps you prepare for security failures in your systems, but different phases of the software delivery life cycle involve different types of failures. We'll be discussing "build security" in this chapter, referring to the security of software delivery beginning at the design phase and including the development work leading up to actual deployment of code on running systems. Postdeployment and ongoing operation of software and services will be covered in Chapter 6, *Production Security in SCE*. In this chapter, we'll discuss how to think about build-phase security as well as explore examples of failure in build pipelines and in the design and configuration of microservices.

When we think about security failure, we tend to think about situations, like data breaches, that occur after the build phase. But security failure starts within the design and development of software and services. Failure is a result of interrelated components behaving in unexpected ways, which can—and almost always do!—start much further back in system design, development processes, and other policies that inform how our systems ultimately look and operate.

Failure in the Build Phase

In the true spirit of SCE, we must conduct experiments to uncover potential failures. This leads to the question: What experiments should you conduct to uncover how your system handles build-time security failures? What happens if there is an outage in your vulnerability scanner or other build-time security tools? Does it halt PRs

being merged? Are releases frozen? Try forcing an outage in one of these tools, for instance, by using a faulty API token for the service.

Security testing tools and the people operating them will never execute perfectly. Therefore, your systems must be designed under the assumption of this failure. A single tool should not be allowed to become a single source of failure. For instance, relying on a vulnerability scanner but not having a compensating control for the potential inability to detect and fix a vulnerability is unwise. And focusing on checking off a list of vulnerabilities means less attention is directed toward design choices that can inherently improve security.

Importantly for cloud-native environments, misconfigurations are huge contributors to security failure but aren't vulnerabilities and don't receive a CVE identifier.[1] Containers allowing anonymous access, unnecessary ports open on servers, orchestration management consoles exposed publicly, revealing stack traces when handling errors, or remote administration functionality being enabled are all examples of misconfigurations that can lead to security failure. While it's easy to see how those can lead directly to security failure (and would constitute a big "oof" moment if discovered as the source of compromise!), other misconfigurations can be subtler. Outdated certificates are an obvious misconfiguration, but mishandled implementation of authentication can be harder to spot—while still capable of resulting in the same ultimate failure.

Embracing failure can help us discover the spectrum of potential sources of failure within applications, not just the OWASP Top 10 or goofs worthy of memes. The types of chaos tests outlined in the rest of the chapter fulfill the goal of validating your security controls, testing your ability to respond, and discovering any negative externalities within your systems. With this information, you can create a feedback loop that informs improved system design—such as creating an automated script that shuts down any opened ports that are not on an allowlist for your response layer of protection.

1 Known vulnerabilities are tagged with a unique identifier as part of the Common Vulnerabilities and Exposures (CVE) system operated by MITRE. For example, the notorious "Heartbleed" vulnerability is alternatively known as CVE-2014-0160.

We'll now explore security failures in applications across different environmental types so you can gain a practical sense of how to introduce SCE into the build phase of your software delivery life cycle.

Application Security Failures in Containers and Image Repositories

Let's start by exploring some common security failures in containers,[2] enumerated in Table 5-1, with an eye toward how this failure can be injected into your systems for SCE. Your organization may not have all these components or configurations in place in your systems, so treat these as an illustrative jumping-off point to brainstorm your own tests.

Table 5-1. Security chaos experiments for microservices environments

Chaos test	Question(s) answered
Add a (massive) throttling limit to APIs.	How does restricted communication affect your systems? Are there any security-relevant tasks that can no longer be performed? How quickly are you detecting availability problems?
Disable noncritical roles and functions in an API.	Is your API failing open or closed? Are any API endpoints hanging? Are they properly timing out? Are your triggers working and alerts being generated as intended? Does the API call still go through, even if there is no response or a malformed response?
Disable read-only access for filesystem and volume.	What is the impact of being able to write to disk in your containers? How does an erosion of immutability affect other security-relevant aspects of the system? How quickly are you detecting file or system modifications?
Disable resource limits (CPU, file descriptors, memory, processes, restarts).	What signals lead you to detect abuse of system resources? How quickly are you detecting spikes in resource utilization? How do you respond to a denial-of-service (DoS)?

2 We recommend looking through OWASP's Docker Security Cheat Sheet for other examples of failure within Docker containers specifically.

Chaos test	Question(s) answered
Enable anonymous access in your orchestrator (e.g., the `--anonymous-auth` argument in Kubernetes).	Is anonymous access enabled or disabled by default in your orchestrator? Are you granting RBAC privileges to anonymous users? Are you tracking activity performed by anonymous users? How quickly are you able to disable anonymous access?
Enable intercontainer communication.	What is the impact of all containers being able to talk to each other? What information can one container disclose to another? How quickly are you detecting and reversing these changes (like `--icc=true` in Docker)?
Enable swarm mode in Docker engine instances.	Are you detecting unwanted open ports? Can other systems on the network access sensitive data via network ports?
Expose Docker daemon sockets (e.g., the tcp Docker daemon socket or `/var/run/docker.sock`).	How does unrestricted root access to the underlying host propagate in your container infrastructure? How quickly are you detecting and reversing socket exposure?
Expose microservices in the LAN from the same application.	Are you able to detect lateral movement between containers?
Expose orchestration management interfaces/APIs.	Are you detecting unauthorized access to publicly exposed management interfaces? How are you tracking changes made via management interfaces or APIs?
Expose traditional orchestration services, like NFS and Samba.	Are you able to audit access to network shares? How quickly do you detect unauthorized access or modification of remotely hosted files?
Generate large loads on specific edges, like availability zones or data centers.	Are your APIs autoscaling? If they aren't, how quickly are you able to detect and restore functionality?
Inject orchestrator impersonation headers.	Do you have limits on user impersonation? Do you have restrictions on what actions impersonated users can conduct? Are you tracking activity by impersonated users?
Inject unapproved lineages.	How do unapproved base images in your image repositories propagate into your CI/CD pipeline? How quickly do you detect and remove unapproved images?
Insert known vulnerabilities.	Where in your build pipeline are you detecting and patching/removing known vulnerabilities in components?
Mount host system directories (like `/boot`, `/dev`, `/etc`, `/lib`, etc.) as container volumes.	How do changes to files within host directories impact your systems? How quickly are you detecting changes to sensitive directories?

Chaos test	Question(s) answered
Remove (or corrupt) random HTTP headers from APIs.	How do your APIs respond to removed headers? Are you detecting when REST bodies don't match the intended content type specified in the header?
Remove resource segmentation.	Which components can remove resource segmentation? How does lack of resource segmentation impact your systems? Are you able to detect when resource segmentation is removed?
Retrograde libraries and containers.	Are you detecting the use of outdated libraries or container images? Where in your build pipeline are outdated components detected? How quickly can you restore more recent versions?
Run services as root in the container.	Are you detecting containers that can facilitate full user privileges without a privilege escalation? Where in your build pipeline is this being addressed?
Run Docker containers with the `--privileged` flag.	Are there any containers with all Linux kernel capabilities enabled? How quickly are you detecting invocation of the privileged flag? What about for existing images?
Run unpatched container runtimes.	How do unpatched container runtimes propagate in your build pipeline, from image repositories, orchestration software, to hosts? What is your mean time to patch? Are you able to detect exploitation of vulnerable runtimes?
Simulate a lack of filtering sanitization on the edge.	How do you handle data downstream internally? What is the impact of not having sanitization?
Switch schema format for internal APIs (e.g., if JSON is expected, return a 404 in plain html).	How do your systems react to new API schemas? How quickly are you able to detect and reverse unwanted changes to schemas?
Turn off API authorization.	Are your internal APIs enforcing authorization? Are you able to detect race conditions? How quickly are you able to detect unwanted access to data? What signals are you using to do so?

Security Failures in Build Pipelines

Build pipelines—commonly manifesting as continuous integration / continuous delivery (CI/CD) pipelines—include a few different components: build and automation servers, code and image repositories, and third-party tools (which are often open source). We've already covered containers and image repositories, so let's now focus on build and automation servers as well as the tools that support CI/CD.

The typical CI/CD pipeline in a microservices environment operates in the following steps: a developer commits code to a code repository, which is integrated via CI into an image repository, which is delivered via CD to clusters. In some pipelines, the developer will have direct write access to image repositories and clusters—which could obviously lead to security failure. Furthermore, automation servers can often present a juicy attack surface due to possessing read and write access between image repositories and clusters.

In fact, there are quite a few ways that security failure can manifest in build pipelines specifically. In Table 5-2, let's explore some of the ways you can inject security failure into these pipelines.

Table 5-2. Security chaos experiments for build (CI/CD) pipelines

Chaos test	Question(s) answered
Add new users in authentication plugins, which access web consoles in automation/CI servers.	Is there authentication required to add new users? What happens when new users are added with web console access? Are you able to detect new users? Can you track their actions?
Approve commits with outdated/unapproved credentials.	How does the system handle outdated/unapproved credentials? Are alerts generated when outdated or unapproved creds are used? How do these commits propagate through the rest of the pipeline?
Create and schedule new jobs via the automation server's script console (bonus points if done with anonymous access!).	Is authentication required to create and schedule new jobs via the automation server's script console? If not, are those changes detected or otherwise audited?
Decrypt stored passwords from the automation server's script console.	Which users can decrypt stored passwords from the automation server's script console? Is any authentication performed?
Elevate the Jenkins service account to sudo or root access.	Are there any restrictions on elevating privileges? Do you detect when the Jenkins service account is granted sudo or root access?
Enable anonymous read access in your automation server, then read build history and credential plugins.	Is there anything blocking enabling anon access? Are you able to detect when anon read access is enabled? What is the access footprint of anon read-only users in the automation server?
Enable anonymous script console access and execute commands.	Is there anything blocking anon script console access? Are there any limitations on commands that can be executed? Are you detecting command execution from the script console? If so, how quickly?
Enable autodiscovery in publicly facing servers.	What impacts does autodiscovery have on your systems? What actions can be taken when autodiscovery is enabled?

Chaos test	Question(s) answered
Exfiltrate credentials via malicious AWS and GCP services (e.g., *storage.googleapis.com*).	Is any exfiltration allowed from your automation servers? Are you able to detect exfiltration via allowlisted services?
Grep and export *credentials.xml* and *jobs/.../build.xml* files.	What restrictions are in place for accessing and exporting sensitive files? Are you able to detect access and export of these sensitive files?
Grep and reads of *credentials.xml*, `master.key`, and `hudson.util.Secret` in Jenkins.	What restrictions are in place for accessing secrets in automation servers? Are you able to detect access and reads of secrets in automation servers?
Inject credentials into code.	How do credentials embedded in code propagate through your build pipeline? How long does it take to detect credentials introduced in code? How quickly are they removed?
Inject credentials into configuration files (e.g., Git config files).	How do credentials embedded in config files propagate through your build pipeline? How long does it take to detect credentials introduced in config files? How quickly are they removed?
Inject credentials or keys into automation scripts.	How do credentials or keys embedded in automation scripts propagate through your build pipeline? How long does it take to detect credentials or keys introduced in config files? How quickly are they removed?
Inject credentials or keys into build logs.	How do credentials or keys embedded in build logs propagate through your build pipeline? How long does it take to detect credentials or keys introduced in build logs? How quickly are they removed?
Inject expired OAuth tokens into builds.	Are expired OAuth tokens still accepted in your build pipeline? If expired OAuth tokens are used, are alerts generated?
Inject network egress outside of AWS S3 buckets, Google Cloud Storage buckets, etc.	Is network egress possible from your cloud storage buckets? Do you monitor network egress from those buckets? How quickly do you stop unwanted egress?
Inject old access tokens used for old apps.	Are old access tokens accepted by old applications? Are you monitoring access activity in old applications?
Inject outdated service account keys to Jenkins jobs.	Are outdated service account keys accepted by Jenkins jobs? Are alerts generated on attempts to use outdated service account keys?
Modify existing jobs and scheduling builds (bonus points if done with anonymous access!).	Can users store secrets or credentials within environment variables? Are you detecting when jobs and builds are modified?
Modify Git commit headers.	Can headers be modified to make it seem like repositories are gone?
Overwrite build logs.	Which users can overwrite build logs? Are build logs monitored? How quickly are unwanted changes to build logs detected and reversed?

Chaos test	Question(s) answered
Read build history in the automation server's script console.	Can you detect users performing reconnaissance, including searching for credentials? Is there authorization to access build history?
Recursively clone a purposefully malicious parent repository.	How does your pipeline handle malicious hooks configured in submodule repositories? To what extent do they affect your pipeline and systems? Can you detect malicious repo configurations?
Reregister dead packages in use by your projects.	Can dead packages be reregistered? What access to your pipeline or secrets is granted by dead packages? Are you detecting the resurrection of dead packages?
Run a cryptominer on your automation / CI servers.	Are there resource limits on your automation / CI servers? Which users can run services on your automation servers? What signals are you using to detect cryptominers on your automation servers?
Run PowerShell commands in your automation server's script console.	Which users can run PowerShell commands in the script console? Are there any authentication controls? What commands are allowed? Are you detecting the execution of PowerShell commands in the script console?
Wipe all code from repositories.	Which users can wipe all code from repositories? Are there any controls to restrict code wiping? What signals are you collecting to detect ransomware? How quickly can you restore code to your repositories when they are wiped?

Key Takeaways

Preproduction security isn't just about vulnerabilities

Misconfigurations offer easier paths in for attackers than exploiting vulnerabilities but are often overlooked in a vulnerability-centric security program.

Conduct experiments early in microservices environments and in build pipelines

Testing helps you surface failure early in the software delivery life cycle and create a feedback loop that improves security-by-design. Consider build security from a systems perspective, including the tools that connect different components together, like automation servers.

CHAPTER 6
Production Security in SCE

As we mentioned before, when we say "production security," we're referring to the security of software delivery beginning at the deployment phase and including the ongoing operation of software and services. You may also hear "production security" called "runtime security" or "operations security." In this chapter, we'll cover some of the core characteristics in infrastructure that create security by design, examples of failure in production systems, and how failure can inform both better infrastructure design as well as a strong detection and response program.

The deployment and operation of software in production is rapidly becoming essential to supporting revenue generation at organizations across a variety of industries. Failure can feel especially scary when it means an outage in a customer-facing service that directly and immediately translates into missed revenue. But, as always, failure still presents a valuable tool in our arsenal to ensure live, running systems can handle incidents gracefully and recover smoothly.

One of the only ways to proactively understand how your production systems respond to certain failure conditions is by conducting chaos engineering tests in production. Don't fret: the authors are still anchored to reality! It's very likely that your team or organization won't feel comfortable starting its SCE program in production systems and will prefer development or staging environments be tested first instead. However, your program should view this as an initial transition period and have a plan to migrate its chaos tests into production. Otherwise, you'll never gain a true sense of how your

production systems will react to failure, which means when the inevitable happens, you won't be as prepared.

Let's start our exploration of production security with how we can make infrastructure more secure by design by embracing the DIE triad.

The DIE Triad

Authored by Sounil Yu, CISO-in-Residence at YL Ventures

The DIE triad is a different way to think about the future needs in security. It advocates for three new paradigms for how we secure systems. The old paradigm focuses on confidentiality, integrity, and availability, or what we know as the CIA triad. The new paradigm focuses on building systems to be distributed, immutable, and ephemeral, or the DIE triad.

The attributes of distributed, immutable, and ephemeral confer security benefits, addressing some of the main challenges that we've had in security, but more interestingly, these attributes can actually counter the need for the CIA triad. If something is distributed, then why do I need any one asset to be available? If something is immutable, then why do I need to worry about its integrity? If something is highly ephemeral, then why do I need to worry about its confidentiality?

This dichotomy can be understood more clearly when applying the analogy used in the cloud-native world around "pets" and "cattle."[1] Pets are things we name and treat with love and care, like that server under our desk or our Social Security number. On the other hand, cattle are branded with an obscure unpronounced string of letters and numbers, and when one gets sick (e.g., has a vulnerability and needs to get patched), it gets culled from the herd and we move on. Cattle are like containers and serverless functions and credit card numbers that change with every transaction.

This has a number of implications. Consider the workforce shortage in information security: is this shortage exacerbated more because we have too few cyber veterinarians or too many pets? One could

1 Randy Bias, "The History of Pets vs Cattle and How to Use the Analogy Properly," Cloudscaling, September 29, 2016, *https://oreil.ly/MXqgF*.

easily make an argument that the greater problem is that we just have too many pets. So over the long term, we need means to exercise better pet control by limiting new pet creation, making it easier to build cattle, and having controls to prevent cattle from becoming pets.

Failure in Production Systems

More attention tends to be paid to the preventative side of the spectrum—attempting to ensure vulnerabilities are removed before software is deployed into production. That is important! However, it is unrealistic to assume that all vulnerabilities will be fixed, especially in components of software over which you have no control, like the container runtime, orchestrator, or underlying kernel itself.

Decision trees, as outlined in Chapter 2, can help you determine the most likely paths attackers will take to compromise your production infrastructure. For instance, an S3 bucket with public permissions would be one of the easiest paths to accessing production infrastructure. Exploiting a zero-day Linux kernel vulnerability to escape a container can represent one of the harder paths to compromising production infrastructure.

Part of the difficulty of production security is the breadth of production infrastructure itself. Production infrastructure can be hosted on-premises or in private or public clouds. It can include physical servers, virtual private servers, virtual machines, and containers running consumer-facing applications or operational applications like automation, configuration management, orchestration, data visualization, and more.

Another layer of complexity is that there are countless types of activity that can jeopardize production security. Attackers can spawn an interactive shell, disable SELinux, and exploit a memory mismanagement vulnerability to gain root access—then load kernel modules to ensure their access is persistent. Or attackers can escalate privileges through an exploit using return-oriented programming (ROP), allowing them to escape a container and gain control over the host. And attackers and internal developers alike can attach a debugger to a production system, facilitating information exposure or privilege escalation.

Identifying the relationships between production components helps reduce the potential for contagion. Performing security chaos experimentation can expedite this process, as simulating the compromise of one resource can exhibit which of the resources connected to it are also impacted. For example, if a cryptominer is randomly injected onto a randomly selected Kibana instance, is the spike in resource utilization limited only to that instance?

This brings us to specific security chaos experiments that help inject failure into production systems.

System Failures in Production

Let's explore how failure can manifest in production systems, including servers, containers, orchestrators—really, any running service. Most production infrastructure runs on Linux, and on Linux systems, everything is a file. From that perspective, security failure can be seen as any file creation, deletion, or modification that results in unintended, negative impacts on the system. As you'll see in Table 6-1, a large number of experiments for production environments include injecting file-related behavior that can jeopardize security or performance.

Table 6-1. Security chaos experiments for production environments

Chaos test	Question(s) answered
Create or modify scheduled tasks via a file or program.	How does your infrastructure respond to unanticipated tasks? How quickly are you able to detect and revert persistence mechanisms?
Create a user account via CLI.	How quickly are you able to detect and revert new, unwanted users?
Create and execute a new file in a container.	How does your container respond to new file execution? Does it affect your cluster? How far does the event propagate? Are you containing impact? How quickly is the activity detected?
Delete `.bash_history`.	Is there any effect to your production operations when bash history is deleted? Can you detect when bash history is deleted?
Delete log files.	Can your team(s) still diagnose a problem or issue in production when log files are deleted? Do you have backups of log files? Are those backups restored automatically? Can you restore systems without certain log files being available? Are you detecting the deletion of log files?

Chaos test	Question(s) answered
Disable access to DNS.	How reliant are your systems on external DNS? Are you able to detect a potential DNS attack? Are you caching entries? Do you have a fallback in `/etc/hosts`? Are you generating alerts if DNS entries point to potentially malicious IP spaces?
Disable in-line API functionality.	Is the correct error-checking in place when inline functions are disabled or return malformed data? How does your API react when inline functions fail? Is the API still able to function? How does it affect your monitoring and logging ability?
Disable resource limits (CPU, file descriptors, memory, processes, restarts).	Can an infinite script take up resources? Is there an automatic process for restoring resource limits? Are you able to take the system offline and reintroduce a new version with resource limits enabled, without impacting operations? Are you detecting when resource limits are disabled?
Disable security mechanisms (e.g., SELinux, AppArmor, endpoint protection).	Are you detecting when security mechanisms are disabled? How quickly do you restore the security mechanisms?
Dump process memory.	Can you detect when attackers dump process memory as part of system exploitation?
Execute sudo commands.	Can you execute sudo in production? Is debugging disabled in production? Are you able to detect and audit when users execute sudo commands in production systems?
Exfiltrate large volumes of data (e.g., TBs).	Are you detecting spikes in data egress? What signals are you using? Do anomalous billing spikes generate security alerts? How quickly are you able to shut down unwanted data exfiltration?
Flood each resource type on a host (I/O, memory, CPU, disk).	Are resource limits in place? How does the system handle resource flooding with limits enabled versus disabled? Are alerts generated upon excess resource requests?
Inject expired API tokens.	Are any resources still accepting expired API tokens? Which resources are denying them? Is your monitoring triggered? Are you verifying signatures? Is your failover graceful? Are you producing standard API responses like 401 and 403?
Inject program crashes.	Is your node restarting by itself after a program crash? How quickly are you able to redeploy after a program crash? Are you able to detect a program crash in production?
Load a kernel module.	Can users load kernel modules at runtime? Are you detecting when kernel modules are loaded in production infrastructure? How quickly are you able to redeploy systems once an unwanted kernel module is loaded?

Chaos test	Question(s) answered
Modify `/etc/passwd`.	Is `/etc/passwd` modifiable? Is there any impact of the modification on your operations? Are you able to detect attacker persistence?
Modify `/etc/shadow`.	Is `/etc/shadow` modifiable? Is there any impact of the modification on your operations? Are you able to detect attacker persistence?
Modify AppArmor profiles.	Are your AppArmor profiles modifiable at runtime? Are you detecting modifications to AppArmor profiles? How quickly are you redeploying affected systems?
Modify boot files.	Are you detecting attacker persistence via modification of boot files? How quickly are you redeploying affected systems?
Modify internal firewall rules and access permissions.	How quickly are you detecting unwanted changes to firewall rules or access permissions? How quickly are you reverting unwanted changes?
Modify root certificate stores.	Are you detecting attacker persistence via modification of root certificate stores? How quickly are you redeploying affected systems?
Modify SSH authorized keys.	Are you detecting attacker persistence via modification of authorized SSH keys? How quickly are you redeploying affected systems?
Run a network service scanner on a host in your infrastructure.	Are you catching and/or restricting lateral movement? Do your internal systems crash due to the noise generated by the scanner? Does it generate any internal errors?
Run container runtimes at the "debug" log level.	How does running at the debug log level impact the information exposed in your container to a user? Is there any sensitive information logged? How do your logging systems handle additional data?
Run SSH within containers.	Is SSH allowed within containers? Are you able to detect what commands are run from SSH within a container? How quickly are you able to redeploy a container after SSH is detected?
Spawn unauthorized interactive shells.	Can unauthorized interactive shells be spawned? Can you detect if the interactive shell occurs outside of normal process guidelines? Are you detecting unauthorized shells? If so, how long does it take for you to redeploy affected systems?
Time travel on a host (change the host's time forward or backward).	How are your systems handling expired and dated certificates or licenses? Are you relying on external NTP? Are time-related issues (e.g., across logging, certificates, SLA tracking, etc.) generating alerts?
Write to files related to bash `profile/rc`.	Are you detecting persistence mechanisms like writing files related to `profile/rc`? How quickly can you redeploy affected systems?

Key Takeaways

Production systems are the new engine of business
: Thus, conducting chaos experiments to learn from failure early and often is essential for supporting ongoing business stability.

Uncovering unknown relationships
: Discovering relationships between components in production systems is a key benefit of security chaos experimentation. Simulating the compromise of one resource can exhibit which of the resources connected to it are also impacted. Identifying the relationships between production components helps reduce the potential for contagion.

CHAPTER 7
The Journey into SCE

By simply "responding" to "incidents," the security industry overlooks valuable chances to further understand and nurture those incidents as opportunities to proactively strengthen system resilience. What if instead we proactively and purposefully initiated incidents expressly to learn their impacts and design graceful automated responses?

Validate Known Assumptions

If the majority of malicious code is designed to prey on the unsuspecting, ill-prepared, or unknowing practitioner, aka "the low-hanging fruit," then it makes sense to ask the following questions: How many attacks would still be successful if there wasn't such a large surface area to begin with? Could it be that this "low-hanging fruit" is the key to proactively understanding how our systems—and the humans that build and operate them—behave?

If we always expected humans and systems to behave in unintended ways, perhaps we would act differently and have more useful views regarding system behavior: assume failure, and design the system to expect failure and handle them gracefully.

We should focus on learning from the failures that happen in our systems. Through this shift in thinking, we can begin to understand what it takes to build more resilient systems. By building resilient systems based on learning from experimental testing, we make

unsophisticated criminals and attackers work harder for their money.

Crafting Security Chaos Experiments

SCE introduces observability through rigorous experimentation to illustrate the security of the system. Observability is crucial to generating a feedback loop. Testing is the validation or binary assessment of a previously known outcome. We know what we are looking for before we go looking for it. Experimentation seeks to derive new insights and information that were previously unknown, and these new insights complete the feedback loop and continue the learning.

Injecting security events into our systems helps teams understand how their systems function and increases the opportunity to improve resilience. SREs, product teams, and security teams are all expected to implement security. They should all be coding their services to withstand potential failures and gracefully degrade when necessary without impacting the business. By running security experiments continuously, we can evaluate and improve our understanding of unknown vulnerabilities before they become crisis situations.

Experiment Design Process

Documentation is an important part of designing, building, and exercising chaos experiments. Careful documentation of experiments is one of the most overlooked and undervalued aspects of the process of creating valuable chaos experiments.

Document Steady State

The first step in developing SCE experiments starts with developing a model of the characteristics of the steady state—also referred to as the normal operational state—of the target system and its expected outcomes. Be sure to document the observable characteristics describing the normal operation of the system.

Design Hypothesis

After documenting the steady state behavior, we use that information to formulate a hypothesis of what could and should happen during your test as a starting point. Hypotheses are typically in the

form of "in the event of the following X condition, we are confident that our system will respond with Y." It's important to remember that we never run a chaos experiment that we already know is going to fail. If we know it's going to fail, we should just fix the known problem.

Hypothesis design considerations:

1. What do we expect to happen in the experiment?
2. What are the specific criteria and characteristics that support the hypothesis?
3. What is the scope of this hypothesis?
4. What are the variables?
5. What are our assumptions?

Contain the Blast Radius

Another crucial design element is to carefully consider the potential adverse impacts an experiment could have and make an attempt to minimize the blast area. Examples of minimizing blast radius could include not injecting failures on internet-facing instances or conducting experiments on instances with lower numbers of users. Many open-source chaos engineering toolsets often include an opt-in/opt-out model to indicate the boundaries for experimentation. For more, see "SCE Tools: ChaoSlingr" on page 65 later in this chapter.

Start small and verify expectations as you build maturity. For example, if the intention is to experiment on a web application firewall's specific types of failure modes, you might choose to initially do this on an application that is not business critical or is in a nonproduction environment to ensure the tool does not create customer-impacting problems.

Fallback Plans

Taking the time to review and document fallback scenarios and procedures as part of preparing to execute the experiment can prove valuable. In doing so, the stakeholders involved will feel more confident in their ability to respond in the event of an experiment hypothesis not being correct. Furthermore, it can be a value-added

exercise to further characterize the expected outcomes of the experiment.

Notify the Organization

When getting started in running chaos experiments in your organization, it's important to inform other members of the organization about what you're intending to do, why you are doing it, and when you plan to do it.

Plan Your Game Days and Execute the Experiment

Tie experiments to game day exercises. Meticulous planning for game days will help you. You should enumerate what could happen in the game day scenarios. This will also help limit the chance of a nasty, business-impacting surprise. Game days should incorporate a variety of stakeholders including security, software engineers, business stakeholder representatives, and other members of IT before launching them. In the next section of this chapter we will discuss SCE game days in greater detail.

Measure the Impact of Each Failure!

You cannot manage what you cannot measure. Measurement provides you the confidence that you can scale the experiment to add scope, width, or breadth to the experiment.

When exploring the defined fault spaces (attack surfaces), security hypotheses should be framed within the context of the security attributes as it is imperative to properly categorizing outcomes (findings). Using outcome types to categorize experiment results can be a good way to measure experiment findings over time.

Security experiments most often result in the following types of outcomes:

Prevented
: The action is prevented.

Remediated
: If not prevented, it's remediated (triaged).

Detected
: If not remediated, then it's detected (logs, alerts).

Validate the Feedback Expected from Your Security and Visibility Tools

Check that your assumptions held true. Confirm that systems and processes (automated or manual) functioned as expected. Was the hypothesis correct? Did anything happen that you didn't expect?

Automate the Experiment for Continual Use

To the greatest extent possible, automate your experiments. Automation allows us to conduct the experiment repeatedly in a precise, controlled fashion. Additionally, it ensures the same environment variables are considered and can thereby be manipulated. Automation also allows you to grow the rigor and coverage of the experiment over time.

NOTE

Security chaos experiments do not:

- validate a config; they exercise it
- check authentication privileges; they attempt to thwart them
- trust network settings; they send real traffic
- check application policy; they interact with the application
- build a model from infrastructure templates; they build understanding from experimentation

Game Days

One great way to get started with SCE is with game days. Game days immediately generate valuable results by injecting failure during a live-fire exercise. The intention is to develop a series of hypotheses associated with security components, plan an experiment, and execute the experiment to either validate or refute those hypotheses.

Security chaos game days are a way of introducing controlled, safe, and observed experiments that allow engineers to observe how well their security responds to real-world conditions. During these exercises, teams often exercise their recovery procedures and security control effectiveness and sharpen their resolution protocols such as runbooks.

A game day is a practice designed to evaluate how to deal with real-world turbulent conditions. They build confidence in the resilience of a system. These exercises involve forcing certain failure scenarios in production to verify that our assumptions about fault tolerance match reality.

Considering that the objective is to answer a question that starts with a "how," the answer should be a strategy, a method, or a list of actions to mitigate failure in the system. Properly executed, a game day program can help improve resilience.

The game day practice consists of a session in which engineers work together and brainstorm various failure scenarios (such as those suggested in Chapters 5 and 6). They create a planning document that describes the system being tested, the proposed failure scenarios to be attempted (including steps that will be taken to simulate the failures), expectations surrounding how the system should respond to the failures, and the expected impact on users. The team works through each one of the scenarios, documenting observations and identifying differences between expectations and reality. With these results, the team has real-world information with which to build greater and greater resilience into the system.

Implementing security chaos game days involves planning before your game day, execution during it, and analysis afterward. There is a complete description of each step in *Learning Chaos Engineering* by Russ Miles,[1] so the following guidance mentions adaptations specifically for security game days.

During the planning:

- The exercise is meant to be cross-functional; take care in planning who should be part of it. For example: SREs, security incident responders, product owners, software engineers, etc.
- Develop proposed hypotheses related to potential real-world security concerns in the system. Some topics for the questions: "What would happen if…?" or "What if…failed? What would happen? Would we know? How?"

1 Russ Miles, *Learning from Chaos Engineering: Discovering and Overcoming System Weaknesses Through Experimentation* (Sebastopol, CA: O'Reilly Media, 2019).

Running the exercise:

- Schedule and execute the proposed hypothesis by injecting the failure condition as defined.

After the game day:

- Once the exercise is completed, the participants conduct a postmortem to understand what worked or didn't work as intended.
- Develop remediation actions.
- Once remediated, schedule time to rerun the experiment to validate that the remediation actions were successful and the hypothesis is current.

Use Case: Security Architecture

Theoretical security architecture will no longer suffice to protect a modern engineering organization transforming its business and products to compete in today's high-demand digital world. Historically, Security Architecture was tasked with incorporating security by design as early in the engineering life cycle as possible. The goal was to avoid the mayhem of the 11th-hour scramble to address security concerns before project go-live. Architects regularly take part in design discussions and assist engineers in translating security requirements into engineering deliverables as part of a project implementation.

The effectiveness of the Security Architecture function is measured against the outcomes of a process that they cannot influence beyond the design phase. No matter how innovative or technically sound their guidance is, their results rely on engineers to properly adhere to the strategies, standards, designs, and implementation requirements they provide. Most projects rarely execute exactly to specification. Unfortunately, unforeseen technical limitations, narrowing project timelines, functional changes, and mistakes can easily undermine the security baselines that Security Architecture works so hard to create. Even if the team met all requirements at project go-live, changes during a system's lifetime routine erode security effectiveness.

But what if we could move to a world where the product team and engineers were conducting their own security experiments, and

were incentivized and excited to learn the outcome and take steps that would improve security and resilience...all on their own initiative?

SCE can be an extremely valuable tool for introducing a feedback mechanism into the design process helping engineers and architects understand whether it was implemented correctly and was operationally effective. The primary need we are solving for is to create a way to directly ask systems basic questions about the state of their operational security after the systems have been deployed.

Use Case: Security Monitoring

Authored by Prima Virani, formerly OpenDoor

Security teams rely on high-quality logs and accurate alerts for a vast majority of security functions. In order to detect incidents and prevent negative consequences from occurring, teams need observable insights into where and when suspicious events are occurring. Suspicious event alerts, fueled by logs, are the most important sources of this insight. If the log pipeline fails, the alerting infrastructure automatically fails along with it.

Additionally, security teams are required to respond to and contain the incident as quickly as possible. Incident response teams rely on accurate, complete, high-quality logs, which not only help them find additional context around any event but also help identify appropriate countermeasures and provide evidence to present in court should the security event escalate and need to involve law enforcement.

There are two major points of failures that can impact an organization's security posture adversely:

- The organization doesn't have accurate log data to start with.
- The organization faces a glitch in the detection infrastructure, hindering them from gaining insights into an event. A "glitch" can be in any of these three parts of the detection infrastructure:
  - — logging
  - — alerting
  - — response

Failure in any of these can cause significant damage in the case of a successful attack.

Gaining New Insights with SCE

Just like any other challenge, there are many ways of approaching this potential failure in logging and alerting infrastructure. One of the fundamental approaches to start solving for it is to verify that all logs are collected centrally and are consistently monitored for unusual access and/or activities. This, in addition to monitoring and alerting on log trends, can take a team's detection and response infrastructure robustness very far. With that, the team can observe unusual spikes and dips in the log flow on every pipeline from which a security information and event management tool (SIEM) receives logs.

To proactively check and identify breakages in the log pipeline and alerting infrastructure, it's important to schedule regular security chaos experiments to check that all the logs and alerts are being received and collected in the security logging and alerting toolsets as expected in a timely manner. In our experience log pipelines are rarely tested after they are initially implemented. As a result, the teams often identify the breakages either completely by accident or during the unfortunate occurrence of a security event and/or incident. Under these conditions, the team is trying to look for the particular log or alert and is unable to find it. Periodic chaos experimentation can help verify the health of these critical functions.

Again, going back to the example of the broken log pipeline, the engineering team were operating under the assumption that the logging and alerting infrastructure was working correctly. When assumptions like that are wrong, they can cause a company to lose millions of dollars on an hourly basis. Consider the July 2018 Amazon Prime Day outage where Amazon incurred costs of up to $33 million per hour while engineers scrambled to diagnose and triage the problem. That three-hour outage could potentially have been identified proactively using resilience techniques such as chaos engineering.

In conclusion, with (well-warranted) increasing emphasis on automation in detection and response, the infrastructure and tools involved in automation are rapidly becoming more complex. In light

of this, we must seek new methods, such as chaos engineering, in order to better understand how these complex systems work, improve them, and anticipate the nonlinear nature of their unpredictable outcomes.

Use Case: Incident Response

By Kyle Erickson, Director of Product Security for Medtronic

Security incident response is a reactive and chaotic exercise. What if you could flip that scenario on its head? Chaos engineering takes the approach of advancing the security incident response framework by reversing the postincident and preparation phases. This is done by creating live-fire exercises that can be measured and managed. It develops teams by challenging responders to react outside their comfort zone.

The most common way we discover security failures is when a security incident is triggered. Security incidents are not effective signals of detection, because at that point it's already too late. The damage has already been done. We must find better ways of instrumentation and observability if we aspire to be proactive in detecting security failures. Insert SCE. What if we could harness the lessons learned from a security incident without the collateral damage?

An intangible aspect of being better prepared for failure is the burnout factor on security incident responders. In the wrong culture—usually a prebreach organization—incident response is seen as the last line of defense for keeping the company out of the news. This is an incorrect approach and is usually abused because of the often-positive perception outcome of a postincident review. So how can we benefit from the incident lessons learned in a safe and blameless environment? Insert SCE.

The reality of real-world failure is that even with all the people, planning, and resources available, you cannot plan and prepare for the unpredictability of a real-world incident. We have conditioned our teams to do nothing but respond. This robotic impulse is costing us time and talent. On top of that, we are missing things that we can learn from. The only way we can improve is to better understand our environments and how they react under threats and attacks.

Now what if we could harness the lessons learned from this event within a safe and controlled environment? SCE allows us to create the live fire without the collateral impact to improve our ability to respond effectively and ensure security tools, techniques, and processes are effective prior to responding to the chaos of real-world situations.

Incident response chaos examples:

- Simulate DoS on a performance or staging environment
- Credential stuffing of a controlled login page
- Disable syslog alerting on a single firewall
- Misconfigure port 445 server message block (SMB) to be left open

SCE Tools: ChaoSlingr

ChaoSlingr, as seen in Figure 7-1, is a security experiment and reporting framework originally created by a team at UnitedHealth Group led by Aaron Rinehart. It was the first open source software tool to demonstrate the value in applying chaos engineering to information security. It was designed, introduced, and released as open source with the intention of demonstrating a simplified framework for writing security chaos experiments.

One experiment involved misconfiguring a port. The hypothesis for this experiment was that a misconfigured port should be detected and blocked by the firewall, and the incident should be appropriately logged for the security team. Half the time, that's exactly what happened. The other half of the time, the firewall failed to detect and block it. But a commodity cloud configuration tool did always catch it and block it. Unfortunately, that tool did not log it in such a way that the security team could easily identify where the incident had occurred.

Imagine that you are on that team. Your fundamental understanding of your own security posture would be shaken by this discovery. The power of ChaoSlingr is that the experiments prove whether your assumptions are true or not. You aren't left guessing or making assumptions about your security instrumentation.

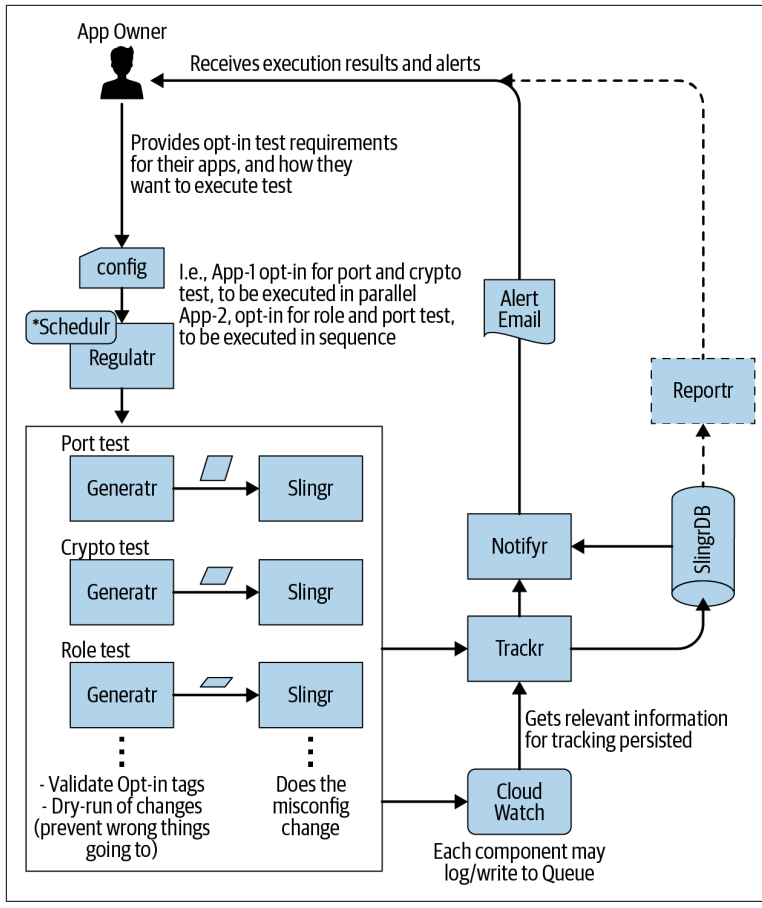


Figure 7-1. ChaoSlingr's high-level context and design

The framework consists of four primary functions:

Generatr

Generatr's job is to identify the target environment to perform the security chaos experiment by filtering on an identified "opt-in/opt-out" AWS reference tag. It selects a random security group that is tagged with the specified opt-in tag and picks a random port range to open for ingress with either TCP or UDP access.

Slingr

Once the target has been identified and opt-in tags have been validated, Slingr executes a misconfigured port change on the

target by opening or closing a port that wasn't already open or closed.

Trackr
: After the change has been executed, Slingr gathers relevant information to Trackr via Slack for reporting and alerting purposes.

Experiment description
: This provides documentation on the experiment along with applicable input and output parameters for Lambda functions.

ChaoSlingr was originally designed to operate in AWS. It proactively introduces known security failure conditions through a series of experiments to determine how effectively security is implemented. The high-level business driver behind the effort was to improve the company's ability to rapidly deliver high-quality products and services while maintaining the highest level of safety and security possible.

ChaoSlingr was developed to uncover, communicate, and proactively address significant weaknesses before they impacted customers in production. The majority of organizations utilizing ChaoSlingr have since forked the project and constructed their own series of security chaos experiments using the framework provided by the project as a guide.

SCE Tools: CloudStrike

Authored by Kennedy Torkura, Data4Life

CloudStrike[2] is the practical realization of the need to develop security fault injection algorithms to instrument the security properties (confidentiality, integrity, and availability) of cloud services.

The starting point for chaos engineering is the selection of a hypothesis around normalcy abnormality, with measurable attributes. Thus, we formulated the concept of *expected state*: the secure state of a resource at time *t*. Essentially, this state is known by the *cloud*

2 Kennedy Torkura, Muhammad Ihsan Haikal Sukmana, and Feng Cheng, "Security Chaos Engineering for Cloud Services" (working paper, Hasso-Plattner-Institute for Digital Engineering, University of Potsdam, Potsdam, Germany, September 2019), *https://oreil.ly/gzOsW*.

resource orchestration engine. For example, an access control policy might specify access for a user (e.g., Alice) for a specific cloud resource at the *provisioning time*. This access policy is known by the orchestration system and a measurable attribute is defined. For example, an HTTP 401 status message (*unauthorized*) is produced if Alice makes a request against the bucket after her privileges are removed. In this example, the policy is modified during a security fault injection action.

In order to simulate real-world events, a variation of possible attacks is implemented. CloudStrike orchestrates random actions against target cloud systems (e.g., deletion, creation, and modification using cloud APIs). Three chaos modes are supported: LOW, MEDIUM, and HIGH, corresponding to the magnitudes of 30%, 60%, and 90%, respectively. Table 7-1 is an example of AttackPoints used: each AttackPoint defines a specific action to be conducted, and a combination of two or more attack points constitutes an attack scenario. Figure 7-2 is a flowchart that illustrates the combination of AP1 and AP4 to create a scenario where an attacker creates a random user in a cloud account, creates a privileged policy for accessing a cloud bucket, and attaches the policy to the malicious account.

Table 7-1. Example of AttackPoints implemented in CloudStrike

Attack ID	Cloud Resource	Chaos Action	Description
AP1	User	create	create random user
AP2	User	delete	delete existing user
AP3	User	modify	change user configuration (e.g., privileges, role, or group)
AP4	Policy	create	create new policies with random ACLs and attach to cloud resource(s)
AP5	Policy	modify	modify existing policy (e.g., change ACL to deny original owner access to the resource)
AP6	Policy	delete	detach policy from a resource, delete the policy
AP7	Role	create	create a new role
AP8	Bucket	make public	alter private configuration to public
AP9	Bucket	disable logging	stop logging API calls against bucket
AP10	Bucket	make unavailable	simulate bucket unavailability (e.g., by changing bucket ACL from ALLOW to DENY)

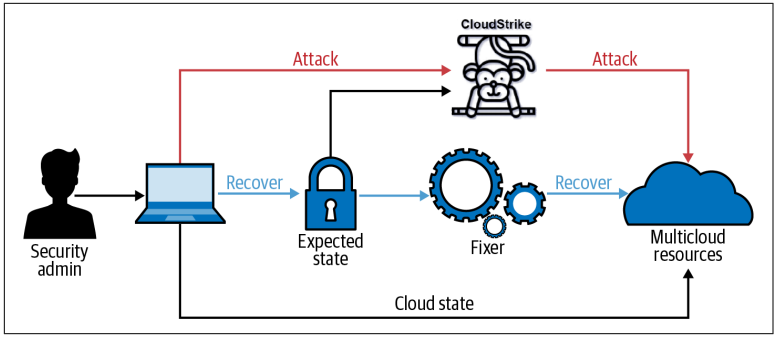


Figure 7-2. High-level architecture of CloudStrike

Key Takeaways

Focus on validating known assumptions
: The prevalence of known systemic weaknesses or "low-hanging fruit" continues to afford malicious adversaries the ability to successfully compromise our systems. Instead of reacting to commonly known accidents, mistakes, and errors in the form of security incidents, these situations become opportunities to proactively strengthen system resilience through proactive security experimentation.

Get started with SCE game days
: In getting started with SCE, game days can be a great way to start generating valuable results from the practice through injecting a failure manually during a live-fire exercise. The intention is to develop a series of hypotheses associated with security components, plan an experiment, and execute the experiment to either validate or refute those hypotheses.

A valuable tool for validating and bolstering security
: SCE can be an extremely valuable tool for introducing feedback mechanisms into the design process, helping engineers understand whether it was implemented correctly and operationally effective. It can also help proactively verify that logging and alerting infrastructure is working as expected, as log pipelines are essential for the accurate alerting that powers prevention and incident response.

CHAPTER 8
Case Studies

There is a growing community of security practitioners and organizations who are both advocating SCE and developing experiments through open source and other community initiatives. The following chapter shares a handful of case studies from organizations that have successfully implemented SCE as a practice within their security programs.

Case Study: Applied Security—Cardinal Health

Authored by Jamie Dicken, Cardinal Health and Rob Duhart Jr., Google

The need for SCE grew organically at Cardinal Health, a global Fortune 20 health-care products and services company based in Dublin, Ohio. With revenue of over $140 billion, Cardinal plays a critical role in the health-care ecosystem.

At the executive level, securing the company's distribution centers, operations, products, and information was critical—not just for the company's reputation but for the stability of the United States health-care system. In addition to investing in teams to advise on and implement security controls, they authorized purchases of security products and tools to protect the business, totaling millions of dollars. With such large investments, executives logically held high expectations for their effectiveness.

The SCE journey at Cardinal Health arose not only from the executive needs for security validation but from the Security Architecture team as well. That team, led by Robert Duhart, knew that theoretical security architecture would not protect an organization sprinting to the Cloud.

What Cardinal Health needed was "Applied Security"—a means to bring continuous verification and validation concepts of SCE to the organization.

Building the SCE Culture

In order to succeed in blazing a new trail, it was imperative that this new adventure begin with a culture of fearless exploration. SCE is an evolution of security and engineering thinking. Rather than building walls and hoping they won't be breached, SCE introduces disruption to maximize the security value within an environment. Given the state of the Cardinal Health cloud transformation, we decided that we were in need of a revolution in terms of how we approached our security practices.

The Mission of Applied Security

The concept of Applied Security was simple: test your security before someone else (an adversary) does.[1] The name itself paid homage to the Security Architecture team's needs to verify that the security measures or designs were appropriately applied, and carefully made no mention of things like "compliance" or "engineering" in order to minimize confusion with other established security teams.

In mid-2019 I (Jamie) was hired by the newly formed Applied Security team to help shape and navigate this new territory. I had over a decade of experience in building and leading software engineering practices, so the concept immediately resonated with me; to me, proactively instrumenting your security was exactly the same as proactively instrumenting your software application.

1 The phrase "test your security before someone else does" was taken directly from Aaron Rinehart's SCE presentations. Not only did this phrase influence and inspire the team, but Robert Duhart used these exact words to justify the creation of the Applied Security team at Cardinal Health.

My first objective in getting started was to reconcile the business case used to create the team with the current expectations of my CISO, other information security teams, and Enterprise IT. After the first month, our mission became clear: identify unknown technical security gaps and partner with the organization to remediate them before they are exploited.

The distinction of "unknown" was made first because anything that was already known—that is, able to be entered into the risk register—already had the attention of our leaders who could prioritize remediations. Applied Security wouldn't need to spend energy on those. Additionally, if the primary root cause or remediation was not a technical control, this remediation could be driven by our strong security awareness team whose responsibilities include security education and consultation. This means that instead of us spending time on addressing known issues or awareness of process gaps, we were able to remove those from our scope and count on other teams to drive those to remediation through their partnerships with department leaders.

The Method: Continuous Verification and Validation

As exciting as it would be to just go nuts, get access to every system and every project, and just look for technical security gaps, we knew we needed a disciplined and repeatable process. In deciding what that process was, we identified three key goals.

First, our process needed to identify indisputable and critically important deficiencies. It did us no good to identify problems that the organization didn't agree were worthy of solving when considering the risk. Therefore, we knew we had to establish benchmarks that were relevant to our company and our systems. Those benchmarks could not be arbitrary, and they couldn't be considered to be merely theoretical best practices that practicalists could dismiss.

Second, our process needed to be thorough enough to see the "big picture" and detail the technical security gap. Whereas previous technical gaps had partial visibility to front-line employees, they lacked the detail to be entered into the risk register, which impacted their prioritization and remediation. For the Applied Security team to be successful, we had to not only identify the technical gaps but integrate into the risk team's remediation process—and that required thoroughness.

Finally, we had to ensure that the technical gaps we found and fixed were not unknowingly reopened in the future.

With these three goals in mind, we created a process called continuous verification and validation (CVV). Simply put, on a regular basis, we wanted to continuously verify that a technical security control was in place and validate that it is implemented correctly.

The CVV process includes four steps.

1. Establish the benchmarks by which gaps would be identified, called Applied Security Benchmarks (ASBMs). To give authority to our findings, we largely use the security standards set by our Security Architecture team and approved by our CISO. If those standards don't yet exist, we partner with the Security Architecture and Security Operations teams to establish the recommendations, and we socialize the benchmarks with any relevant teams. This gives credibility to any gaps we identify and establishes the requirement that either fixes need to be prioritized or leaders must accept the risk.
2. Implement continuous checks that our systems are adhering to the ASBMs. In many cases, we write custom scripts to verify the presence and proper configuration of security tools or design patterns, especially because we check adherence to Cardinal Health security standards. Before we write any code, we also evaluate both open source and commercial products and make an informed decision to build versus buy.
3. Create a dashboard that illustrates the real-time compliance of all systems with the ASBMs. This dashboard serves two purposes: it allows us to get the big picture of a possible technical gap on demand, and it is a great communication tool when we speak to our leaders.
4. If at any point adherence to the ASBMs decreases, we create an issue in our risk register. Thanks to the work our risk team has done over the past five years, there is already a fine-tuned governance process that drives remediation and is incredibly effective. This means that the Applied Security team can then move on to implement CVV in other areas while remediations naturally occur in parallel.

While tremendous progress has continued and we've "taken the first hill," we recognize that victory in one battle does not win a war. We must continue "once more into the breach" and create a sweeping culture of SCE. As the organization shifts to an agile and DevSecOps mindset, the goal is to bring SCE into the fold at a larger scale and more broadly across teams. Our vision is that all teams are proactively testing their own security, discovering their weaknesses, and fixing issues before they ever get to production—effectively not even giving attackers a chance to exploit our systems. If a single five-person team has succeeded at proactively finding and fixing technical security gaps for a fraction of our tools, we know that an emboldened organization doing the same grants us far more reach and maximizes our chances for success.

Case Study: Cyber Chaos Engineering—Capital One

Authored by David Lavezzo, Capital One

Complexity. It's extremely difficult to predict all the outcomes of one seemingly small change. Having a combination of an on-premise and multicloud environment presents challenges that may not be apparent until they're in production. How do you know what is and is not working? How do you measure progress?

Measurement is hard. Production systems need to be available at all times. Traditional metrics look for install base and evidence that it's running. An audit looks for evidence it works, so you pull a snapshot showing that one time the tools detected something or stopped something. But what does that really tell us? We know it's installed, it's available, and at a specific point in time, has detected/prevented something. With constant development and platform management, measurement is a moving target.

"Go-live" is, for the most part, the only time you truly know your systems. You've documented the configurations and the policies, and validating things are working as expected. Soon, the realities of business operations set in. Platform updates, patching, tuning, competing priorities—these all find ways to introduce variables into how something operates. A new update meant to enhance capabilities could introduce downstream effects such as loss of effectiveness or bugs that cause issues at large scale.

Enter SCE

Measuring and testing the effectiveness of tooling is a relatively recent practice; previously, public and private experts forged their own paths. The challenge wasn't technology but building consensus. The assumption at the time was "Everyone knows we have challenges with endpoint visibility," and this led me to question if the consensus was true. Enter chaos engineering, but for security. Instead of a focus on resilience against service disruptions, the focus was to identify the true current state and determine what is "normal" operation.

At Capital One, we started by measuring the top 10 critical endpoint alerts to find out if our assumptions were supported. I had two goals in mind: alert validation and detection. Using real-world threats and events, I was able to confirm that the work we did on endpoint visibility was a huge success, which led directly to the increase in detection and alerting capabilities. It gave us a way to address the issue of measurement and development of a five-stage process we abide by:

1. **Know your steady state.** The baseline where all measurements start. There is no good, there is no bad, only the beginning of the journey.
2. **Experiment, observe, and analyze.** See what happens when you introduce new things to the baseline.
3. **Identify gaps and deviations from expected outcome.** Does everything work how you think it works?
4. **Improve the system based on analysis.** Leverage existing partnerships to help everyone get better, making the entire program better.
5. **Repeat continuously in production.** Real systems need to be tested for real results. Just be careful.

Leadership Buy-In

By baselining the defensive environment, we could show a measurable improvement, prove tooling is working as expected, and do more than measure security uptime.

It's a powerful change, moving from "We don't know…" to "Here's what we know and how we know it." By applying automation after the baseline, we can identify when something deviates from the

steady state and help drive improvements and bring visibility to unintended effects of positive changes. We identify what works and what needs improvement. We can show the great work our teams are doing that isn't traditionally recognized, and the positive impact being made to the enterprise.

Key Takeaways

SCE and Applied Security

Cardinal Health decided that they were no longer going to rely on theoretical security architecture to protect a Fortune 20 health-care company as it transitions to the public cloud. The need for a more objective measurement of security effectiveness gave rise to the business case for proactive security testing and experimentation, or what they refer to as "Applied Security." Cardinal Health's Applied Security became a means to launch the CVV concepts of SCE to the company.

Cardinal Health's three key goals to focus on for building a disciplined and repeatable process

Prioritize indisputable and critically important deficiencies, be thorough enough to see the big picture of the technical security gaps, and, lastly, ensure the previously identified gaps are not unknowingly reopened in the future.

Capital One's "Cyber Chaos Engineering"—challenging security assumptions

Instead of a focus on resilience against service disruptions, the focus was to identify the true current state and determine what is "normal" operation. Capital One kept their scope small initially, only focusing on validating alerts and detection capability for the top 10 subset of endpoint alerts. Keeping focused allowed the team to experiment on a prioritized subset of their security, quickly aiding in their ability to demonstrate success.

Conclusion

At a time when constantly evolving software and systems feel so vulnerable—fragile even—applying chaos engineering to information security feels not just apt but imperative. When applied in a security context, chaos engineering bears the potential to reveal valuable and objective information about systems security, allowing organizations to more verifiably ensure the stability and reliability of the systems on which they depend.

SCE—and even chaos engineering itself—is still an emerging and advancing set of practices, techniques, and toolsets. It's true that the shift in mindset is going to be tough for a few organizations initially, especially for those clinging to a traditional security ethos. But if we begin to challenge our assumptions, cultivate collaboration rather than silos, and learn through experimentation, we are certain to make a profound impact on an industry that so desperately needs it.

In conclusion, it is our belief that organizations charged with building safe and performant systems will find the same value we have in applying SCE. We hope that you, the reader, will join us in building a community of practice and help advance the state of SCE.

Acknowledgments

We are grateful to all the amazing folks who helped us produce this report. It would not have been possible without the tireless efforts of our authors and technical reviewers.

We'd like to take time to recognize the many contributing authors that contributed to each chapter by providing thoughtful input, content, and review:

- Vladimir Wolstencroft
- Sounil Yu
- Kyle Erickson
- James Wickett
- Tim Dentry
- Jerome Walter
- Prima Virani
- Yuri Nino
- Kennedy Torkura
- Jamie Dicken
- Rob Duhart Jr.
- David Lavezzo
- Sue Allspaw Pomeroy

We'd also like thank the following reviewers for providing especially valuable feedback:

- Bea Hughes
- John Allspaw
- James Burns
- Randall Hanson
- Quinn Shamblin
- David Reese
- Sean Poris
- Will Gallego
- Esteban Gutierrez

About the Authors

Aaron Rinehart has been expanding the possibilities of chaos engineering in its application to other safety-critical portions of the IT domain—notably cybersecurity. He began pioneering the application of security in chaos engineering during his tenure as the Chief Security Architect at the largest private healthcare company in the world, UnitedHealth Group (UHG). While at UHG Aaron released ChaoSlingr, one of the first open source software releases focused on using chaos engineering in cybersecurity to build more resilient systems. Together with Casey Rosenthal from Netflix, Aaron recently founded a chaos engineering startup called Verica and is an author, consultant, and speaker.

Kelly Shortridge is the Vice President of Product Management and Product Strategy at Capsule8. Kelly is known for research into the applications of behavioral economics and resilience to information security, on which they have spoken at conferences internationally, including Black Hat USA, Velocity, AusCERT, Hacktivity, TROOPERS, and ZeroNights. Previously, Kelly served in product roles at SecurityScorecard and BAE Systems after cofounding IperLane, a security startup that was acquired.