FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN PROFESSIONAL SOFTWARE DEVELOPMENT

CHAOS ENGINEERING

eMag Issue 67 - Nov 2018

InfoQ

IN THIS ISSUE

A LETTER FROM THE EDITOR

Nora Jones

As our systems scale, we need more complexity, which inherently increases exponentially over time. The need for understanding and navigating this complexity also increases. Chaos engineering is a discipline that allows us to refine, recalibrate, and navigate the understanding of our systems through intentional and careful experimentation in the form of failure injection. This greater understanding ultimately leads to a better experience for our customers and better outcomes for our businesses.

At Netflix, we've been embracing chaos engineering since Chaos Monkey was born in 2011. It has gone through several iterations and tools that eventually evolved into the Failure Injection Testing (FIT) platform and, ultimately, ChAP (a platform for safely automating and running chaos experiments in production) through the efforts of many amazing engineers. We've taken the opportunity to outline why this has been so beneficial for the business in a separate IEEE article titled "The Business Case for Chaos Engineering" and a free e-book from O'Reilly here.

One thing I've noticed in my experiences with chaos engineering at various companies is that each approaches it differently based upon the key business objectives, the architectural decisions, and behaviors and motivations of the people that make up the organization.

The first piece in this eMag is written by Michael Kehoe from LinkedIn, and it elaborates on just that point. Here we get a lens on how they incorporate and regularly embrace failure. He dives into the practice of chaos engineering at LinkedIn and their resilience-focused Project Waterbear. Waterbear is a fascinating effort that has allowed multiple chaos tools to exist within LinkedIn.

The next article is by Patrick Higgins from Gremlin. Patrick and I have spoken multiple times about why UI is such an important focus with infrastructure tooling — and specifically why chaos engineering needs dedication and care put into the UIs of the tools. A big thing we are tasked with as chaos engineers is not simply writing tools that inject failure, but writing tools that inspire users to act and develop a deeper understanding of the vulnerabilities they find. Higgins digs into how well-made UIs can facilitate this deeper understanding.

The article by Aaron Rinehart is a fascinating view on why chaos engineering doesn't just focus on reliability — security needs it as well. Through his lens, he provides concrete examples on how chaos engineering has worked for him in security engineering and how others can implement similar measures to protect their systems.

The final article of this series, by John Allspaw, ties this all together in a way that eloquently explains why we do chaos engineering in the first place: to recalibrate the mental models we have of our systems.

It was a great pleasure to work on this eMag, and I can't thank Charles Humble enough for the opportunity and patience had while working on this series. I also want to extend a thanks to all the contributors for their hard work and dedication to both their articles and their craft. I hope that you enjoy the eMag we have created together and that it inspires you to dig deeper into your systems, question your mental models, and use chaos engineering to build confidence in your system's behaviors under turbulent conditions. Happy reading!

CONTRIBUTORS

Nora Jones

is a Senior Chaos Engineer at Netflix. She is passionate about delivering high-quality software, improving processes, and promoting efficiency within architecture. Occasionally, she pokes holes in distributed systems to make them more resilient.

Michael Kehoe
is a staff SRE at LinkedIn, working on incident response, disaster recovery, visibility engineering, and reliability principles. He specializes in maintaining large-system infrastructure as demonstrated by his work at LinkedIn (applications, automation, and infrastructure) and at the University of Queensland (networks). Kehoe has also spent time building small satellites at NASA and writing thermal-environments software at Rio Tinto.

Patrick Higgins
is a software engineer at Gremlin. He focuses on UI development and is interested in the client-side implications of adopting principles from chaos engineering. He has instructed workshops focusing on React development and has spoken at numerous conferences. He has a background in civic engagement and political campaigns and has a bachelor's degree in government and international relations from the University of Sydney.

Aaron Rinehart

has been expanding the possibilities of chaos engineering in its application to other safety-critical portions of the domain, notably cybersecurity. He pioneered the application of chaos engineering to security during his tenure as the chief security architect at the largest private healthcare company in the world, UnitedHealth Group (UHG). While at UHG, Rinehart released ChaoSlingr, one of the first open-source software products to focus on using chaos engineering in cybersecurity to build more resilient systems.

John Allspaw

has worked in software systems engineering and operations for over 20 years in many different environments: biotech, government, online media, social networking, and e-commerce. Allspaw's publications include the books The Art of Capacity Planning (2009) and Web Operations (2010) as well as the forward to The DevOps Handbook. His 2009 Velocity talk with Paul Hammond, "10+ Deploys Per Day: Dev and Ops Cooperation at Flickr" helped start the DevOps movement. Allspaw served as SVP of infrastructure and operations and then CTO at Etsy, and holds a M.Sc. in human factors and systems safety from Lund University.

KEY TAKEAWAYS

LinkedIn was originally built as a monolithic application that served members during the hyper growth period of the company.

The firm moved towards SOA with RPC, and then to Restful APIs using the in-house-developed Rest.li framework. As the firm scaled their microservice architecture, they found resilience engineering became almost impossible.

LinkedIn started a per-application resilience engineering effort called Project Waterbear that built a suite of tools for running chaos-engineering experiments.

In the fall of 2017, one of the LinkedIn products went through a small period of ungraceful degradation (on front ends) because downstream services were misbehaving.

LINKEDIN'S WATERBEAR: INFLUENCING RESILIENT APPLICATIONS

by **Michael Kehoe**

Since the beginning of SRE (and DevOps), professionals in this space have focused on building reliable and highly available applications. We've entrusted ourselves to successfully execute this difficult task with almost no way of testing our successes. In limited numbers, large-scale tests were previously done to test full disaster-recovery strategies (including LinkedIn's here).

Over the past few years, our industry has come to realize that this isn't enough. With the rise of microservice architecture, we can no longer ignore the cost of blind trust; therefore, we must constantly test applications to ensure they are reliable and available.

LinkedIn was originally built as a monolithic application that served members during the hyper growth period of the company. As our membership continued to grow, we realized a monolith was no longer suitable to scale our infrastructure. We started to build a service-oriented architecture (SOA) using remote procedure call (RPC) APIs that were available behind hardware load balancers. Infrastructure footprint continued to scale, as did developer velocity, and we quickly found ourselves needing a more agile approach to publishing and discovering APIs.

An internal "InDay" hack day resulted in the birth of Rest.li. Rest.li allowed engineers to write standardized RESTful API models that were easily discoverable. Moving away from RPC also freed us from high coupling with presentation tiers and many backwards-compatibility problems. Plus, by using Dynamic Discovery (D2) with Rest.li, we got automated client-based load balancing, discovery, and scalability of each service API. To date, we have over 3,500 Rest.li resources.

The move to a microservice architecture greatly enhanced our developers' velocity as well as their ability to scale our infrastructure. The downside of this approach is having to manage multiple dependencies to ensure that an application will be resilient during a downstream interruption. As the number of microservices grows, the call graph also grows, making manual application resilience engineering nearly impossible.

At LinkedIn, we started this focused per-application resilience engineering effort with a project aptly named Project Waterbear. Waterbear aims to:

- ensure we are running on a resilient cluster of resources,
- create or maintain robust infrastructure for our services,
- handle failures intelligently,
- gracefully degrade when required, and
- increase SRE happiness by designing self-healing systems.

This is done via three software platforms we've built as part of Waterbear:

- FireDrill provides an automated, systematic way to trigger/simulate infrastructure failure in production, with the goal of helping build applications resistant to these failures.
- LinkedOut is a framework and tooling to test how user experience will degrade in different failure scenarios associated with downstream calls.
- D2 Tuner recommends Rest.li D2 settings (like degradation and timeout thresholds) based on each service's historical production performance.

In the 18 months that we've been using the Waterbear suite of software, LinkedIn has been able to improve our application resiliency as well as adopt a mindset of building failure-tolerant infrastructure.

FireDrill

We started this project by creating host-level failures. The type of failures we chose to simulate initially were:

- consume CPU on a server,
- fill a disk-space partition, and
- inject latency or packet loss into TCP connections for an application.

As shared in our LinkedIn Engineering blog post on this topic, we use modules written in SaltStack to simulate these failures. Our goal with FireDrill is to move to an automated setup where we can constantly simulate these failures in production. A "detected failure" means a piece of infrastructure fails completely and can be identified as sufficiently faulty (network down, high CPU utilization, shutdown, etc.). A "gray failure" means that a system is reported as healthy but is not functioning properly (network latency/packet loss).

FireDrill allows us to inject both detected and gray failures to observe how the application responds to failure and how our monitoring and alerting systems react to the change in behavior. Through these observations, we are able to understand the changes required to build more resilient applications and to correctly observe these systems.

As FireDrill continues to develop, we have the ability to add more detected and gray failure modes to our toolset and ensure that LinkedIn applications are resilient to less common failure modes. As we get more comfortable with simulating host-level failures, we intend to use FireDrill to create power, switch, and rack failures in our data centers.

LinkedOut

As detailed in a LinkedIn Engineering blog post specifically about LinkedOut, we inject latency, errors, and exceptions at the API-framework layer. Targeting specific engineers allows us to execute fully realistic resiliency tests in a very simple, repeatable fashion, allowing engineers to completely understand how their applications operate in the face of instability. This particularly helps us evolve the resiliency-engineering culture as anyone can run experiments against their service seamlessly without impacting the member or their peers, lowering the bar for resilience testing.

LinkedOut is different from most chaos-engineering tools or products currently available; with this tool, LinkedIn engineers can perform resiliency testing at the application level (by manipulating framework responses). This means that instead of testing infrastructure resiliency individually, we test the actual application's response to both infrastructure and application latency or errors.

Using this application failure injection, we have the ability to not only test for resiliency in the face of failure, but also to inject arbitrary capacity constraints into the applications and concretely understand how the application behaves instead of making calculated guesses.

LinkedOut also goes one step further than traditional chaos-engineering tools as it allows engineers to perform regression tests to validate whether a page or API call fails to load when each downstream dependency is slow or fails. This ensures engineers and SREs are aware when a new critical downstream is introduced, or a code change modifies the graceful degradation of the page.

D2 Tuner

As mentioned earlier, LinkedIn's internal applications use the Rest.li framework to communicate with each other.

The Rest.li framework is super powerful, but requires users to tune a number of configurable parameters to make it work optimally. As a result, some service owners just use the default setting. However, this can be suboptimal, because different services have dramatically different profiles (QPS, latency, etc.).

Rest.li is comprised of two parts: R2, the REST and JSON framework, and D2, the service-discovery layer. D2 has a number of parameters that allows the weighting of a server to be decreased if it is exhibiting high latency or errors. By default, D2 comes with a 10-second timeout for any API call. This isn't suitable for many client-server relationships, and can actually cause stability issues for the client if it has to wait 10 seconds for the response.

D2 Tuner analyses the client-server latency and error rates over a period of time and recommends ideal latency and error thresholds for D2. When engineers implement these recommendations, it helps increase resiliency of applications by ensuring the service only talks to healthy servers and fails the response without compromising the health/capacity of the client service.

The tool in practice

In the fall of 2017, one of the LinkedIn products went through a small period of ungraceful degradation (on front ends) because of misbehaving downstream services. After the second event, the Waterbear team got involved in the war room to provide insight into what was going wrong.

After running the affected service through a series of resilience-engineering regression tests and expecting that one or more downstreams would either be slow or respond with an error, the development team had a comprehensive picture of how each downstream affected the performance and stability of their service. Using this data, the team made a series of improvements to their codebase with resiliency directly in mind. Again, using LinkedOut, these engineers were able to validate the improvements to the application without harming the end-user experience. The result of this work was significant performance and capacity gains for the product. A similar effort was subsequently put into our other front-end services to improve their resiliency.

Conclusion

As Waterbear continues to gain adoption within engineering teams at LinkedIn, we are all reminded that resilience engineering is a part of our craft as engineers. As LinkedIn moves towards white-box servers and switches, we expect failure in those devices, which affects the software on that infrastructure. Waterbear gives us an opportunity to validate our software's behavior against many types of failure and ensure the best possible user experience in all scenarios.

KEY TAKEAWAYS

A UI can function as an important teaching mechanism regarding the product. It helps to set expectations of the capabilities of the tool and good implementation practices.

Investing in UI tooling provides the ability to implicitly encourage engineers to take particular actions.

A key consideration in performing any chaos-engineering experimentation is the necessity for strong safety mechanisms to allow engineers to, for example, halt running experiments if they get out of hand.

Chaos engineering is regularly a collaborative practice. As chaos engineering grows in prominence, gamedays have become a process through which teams come together to test their systems collaboratively.

UIS: VALUE OF THE VISUAL IN CHAOS ENGINEERING

by **Patrick Higgins**

When building user interfaces (UIs) for chaos engineers, creators face a particular conundrum: if the user can code, what value does a graphical UI add? It's a fair assertion that a user interface is essentially an additional layer of abstraction placed between the user and their objective, so if a product already has an API or CLI with great documentation, why provide a UI at all? The answers lie in the additional insight that a strong UI can convey and the incentive that it can offer users.

Guidance for new users should be digestible and highlight clear user workflows. One first step in obtaining this outcome would be to establish a strong contextual consistency for the user.

UI as guide

The UI can be an introductory medium through which a user becomes acquainted with a tool. First impressions are usually hard to shake and the initial interaction between the user and the UI generally forms the basis for how a user feels about the product or tool in its entirety. For this reason, it is particularly important to the adoption and sustained use of a tool that those early interactions are helpful, constructive, and clear.

It's helpful to think of the relationship between a UI and users to be much like that between a hotel's concierge and the guests. A guest, much like a user, arrives at reception with a particular objective in mind — for example, directions to a local restaurant or a tourist attraction. It is the job of the hotel concierge to give directions based on the guest's knowledge of the location and capacity to reach the destination. A guest with no familiarity of the surroundings perhaps requires directions out the door and all the way to their intended destination. Removing presumptions of a base level of knowledge makes a tool accessible to those that may be coming from an entirely different technical context and creates opportunities to appeal to a broader audience. At its most effective, perhaps this opens the door for technologists coming from an entirely different field of specialization.

Guidance for new users should be digestible and highlight clear user workflows. One first step in obtaining this outcome would be to establish a strong contextual consistency for the user. Contextual consistency essentially refers to intuitive design. It can usually be achieved through making elements that function similarly within the UI resemble one another. Establishing contextual consistency when building a UI is of paramount importance for chaos-engineering tooling because configurations are often complex, and accurately understanding the target, impact, and state of an experiment is vital to safe and successful experimentation. Additionally, the UI may have to accommodate the administrative or organizational complexity of the team or company that the tool is servicing. The need for strong standardization around administration and permissions means that a lot of upfront consideration may be required for organizing how data is represented within a UI application.

While these principles are essential in developing good tooling for chaos engineering, they have always been fundamental in the creation of good UIs. In 1986, Ben Shneiderman published the first edition of Designing the User Interface: Strategies for Effective Human-Computer Interaction. The book featured eight golden rules for interface design, the first of which is "strive for consistency." Shneiderman suggested familiarity in the use of icons, colors, menu, hierarchy, calls to action, and user flows to facilitate similar tasks that a user may be attempting to achieve, thus limiting the cognitive overhead of learning new representations for common processes. While the capabilities of what can be done within a browser have leapt ahead, certain staples of communication have clearly remained fundamentally relevant.

UI as trainer

Investing in UI tooling also provides the ability to implicitly encourage engineers to take particular actions. This can occur through the prioritization of certain information or particular user flows. From the perspective of chaos engineering, this could be done by linking related experiments to focus on the iterative nature of the discipline. Most chaos engineering begins with small, localized experiments. As these experiments are validated, experiments may increase in either scope or intensity. This relative increase can be easily reflected in the UI and could focus on the process of developing resilience rather than emphasizing one specific kind of experiment in the abstract. Not only would this display promote the running of experiments as part of a continual practice, it would organize the attacks in a general sense, giving the user a sense of the continual progression of their practice.

Another of Shneiderman's eight golden rules is "enable frequent users to use shortcuts." Shortcuts shouldn't be limited to only templates and hotkeys within the UI but could extend to shortcuts for tooling in a more integrated sense, with the UI forming one part of a coordinated experience alongside an API and CLI. As an experiment is configured in the UI, the functionality to export the parameters for use in a CLI or API would provide users the flexibility to adapt experiments to their particular use case with ease. If users move away from UI usage and begin to configure experiments through an API, this wouldn't constitute a failure for the UI as a tool. To the contrary, it represents a level of maturity that was reached through the regular use of the graphic interface.

The UI can function as an important teacher about the product. It helps to set expectations of the capabilities of the tool and good implementation practices. This is particularly important in the field of chaos engineering where injecting failure comes with the territory. The UI has the opportunity in these instances to form the link between good documentation and the application of the tooling that it graphically represents. In this way, the UI becomes a pedagogical mechanism for the user as they delve into the concepts and principles that underpin the tooling.

UI for safety

A key consideration in performing any chaos-engineering experimentation is the necessity for strong safety mechanisms. The Gremlin UI, for example, features a "Halt All" button, which is available on every page after the user has logged in. Having clear indicators to show the state of any experiments helps to create a controlled experience throughout the process of experimentation. Not only does it provide a level of safety, the system status information actually builds implicit trust. This increases the user's attachment to the tool and to chaos engineering more broadly.

Trust also develops through feedback mechanisms after actions. Feedback gives the user confidence that the actions taken in the UI are immediately registered throughout the process of a chaos-engineering experiment. The ability to provide contextual feedback is another strength of the UI, particularly in the case of error messages that can guide the user through unhappy paths. Feedback in these circumstances is absolutely vital to the user's ability to make sound judgements as to their next course of action.

UI for collaboration and inspiration

Chaos engineering is regularly a collaborative practice. As chaos engineering grows in prominence, gamedays have become a process through which teams come together to collaboratively test their systems. As teams run gamedays, a common UI provides the advantage of a shared context, language, and metrics among the different teams. Chaos engineering is situated within a context of constant innovation. With the rise of new technologies and their associated paradigms (serverless being a good example), establishing shared terminology can help foster effective collaboration and improve the efficacy of a team.

As the application of chaos engineering extends across specializations, a well-designed UI helps bring new teams on board and cultivate best practices. Those already convinced of the benefits of chaos engineering do not need to be sold the conceptual merits of the practice. To those who have not yet seen the light, graphic interfaces can tell a story that they might be ready to hear. The ability to display data that shows how a team has increased resilience over time can only encourage new teams and grow adoption of chaos-engineering practices: metrics put weight behind the principles of the practice. A team that has been working hard to remove the fragility from a system can and should celebrate successes. It is my hope that engineers will keep these successes in mind as they create UIs that can display the hard work and accomplishments.

Chaos engineering has entered a particularly exciting period and the recent growth of the chaos-engineering community has been nothing short of meteoric. As engineers discover weaknesses in their system, so too will they discover new and exciting ways to experiment with failure. I hope that user interfaces can help along that innovation and drive new and exciting ways to break things on purpose.

USING CHAOS ENGINEERING TO SECURE DISTRIBUTED SYSTEMS

by **Aaron Rinehart**

KEY TAKEAWAYS

We must be more proactive in identifying failure in our security and the systems they protect.

Chaos engineering tests a system's ability to cope with real-world events such as server failures, firewall rule misconfigurations, malformed messages, degradation of key services, etc. using a series of controlled experiments.

The gameday exercise is the core channel used to plan, build, collaborate, execute, and conduct post-mortem analyses for experiments.

Consider the evolving nature of contemporary engineering practices. Our systems are becoming more and more distributed, ephemeral, and immutable in how they operate. Not only are we becoming more complex but the rate of velocity at which our systems are interacting and changing is making the work more challenging.

In this shifted paradigm, it is becoming problematic to comprehend the operational state and health of security in our systems. The engineering community in this new frontier is ill equipped and outgunned when it comes to observing, understanding, or predicting our systems' behavior.

Chaos engineering is an empirical, systems-based approach to instrumentation that addresses the chaos in distributed systems at scale and helps builds confidence in the ability of those systems to withstand realistic conditions. The primary objective is to develop a learning culture surrounding the behavior of distributed systems by observing it during a controlled experiment. In layman terms, chaos engineering is the practice of breaking your own systems on purpose in order to observe and derive new insights from the results of a controlled experiment to include the cascading effects it has on systems.

Despite some similarities, chaos engineering differs from both red/purple-team security testing and penetration testing in its goals, purpose, and methodology. The ultimate goal of red/purple teaming is to gain access to sensitive resources through deceptive adversarial methods without causing disruption to live systems to ascertain the effectiveness of preventative security controls. However, the ultimate goal of security chaos engineering is to learn through careful and methodical systems experimentation. This approach focuses on injecting failure into specific components to uncover unknown and unforeseeable problems in systems security before they affect the operational integrity of core business products and services.

We must face the simple fact that the systems we are building are evolving so quickly that we can no longer cognitively interpret them. We have much bias and despite prior knowledge, we simply only see what we want to see. Our understanding and belief systems can act like a mirror, only showing us what we believe while hiding the unknown context of what really happened. The purpose of chaos engineering is not only to break things, it's about revealing how our complex adaptive systems really work versus what we thought we knew.

How and why to use security chaos engineering

The lack of feedback loops for security in iterative and adaptive distributed systems is causing a slow drift into state of unknown. The predominant approach to security design traditionally has focused on the prioritization of mechanisms that can provide prevention. Regrettably, you don't have to be a cybersecurity expert to recognize that this hasn't exactly been successful. One of the fundamental issues with modern security engineering is that we are still designing stateful security in a predominantly stateless world. The way we design, implement, and instrument security has not kept pace with modern product-engineering techniques such as continuous delivery, DevOps, and advanced microservice architectures.

We typically don't recognize failure until something isn't working properly, and this is often true with security incidents. Security incidents are not effective measures of detection because at that point it's already too late. We must find better ways of driving instrumentation and system observability to be more aware in detecting security failures. A common saying in chaos engineering is "hope is not a strategy." When we began our journey in applying chaos engineering to security with ChaoSlingr, we began to think differently about the role of failure, both in the way we build systems and the ways we attempt to secure them.

Security should be objective and rigorously tested, with data and measurement driving decisions about enterprise security. Complex distributed systems, continuous delivery, immutable machines, and advanced delivery models challenge traditional testing methods. While testing can be a useful feedback mechanism for system instrumentation, testing by itself is insufficient; we should also experiment.

Testing seeks to assess and validate the presence of previously known system attributes. Experimentation, in contrast, seeks to expose new information about a system through observation awhile adhering to the scientific method. Testing and experimentation are both excellent ways to measure the gap between how we assume the system is operating and how it actually is operating.

Like their engineering brethren, security technologies must be end-to-end instrumented.

Use practiced instrumentation that seeks to identify, detect, and remediate failures in security controls. Instrumenting the security-capability toolchain from end to end in order to derive insights into not only its effectiveness but also to discover where you can inject added value and improvement.

Continuously measure security to build confidence in the system's ability to withstand malicious conditions. This includes both testing and experimenting.

Proactively assessing the readiness of a system's security defenses ensures that they are battle-ready and operating as intended.

The insights can sharpen the operational effectiveness of incident management

Before we dive into chaos engineering and its evolution into security, consider where data breaches typically come from.

The Ponemon Institute's "2018 Cost of a Data Breach" study is the most recent of annual reports that analyze the prior year's notable public data breaches. It breaks down causes of breaches, targeted industries, and the costs victims incurred as a result of the breach. This year's edition reported that malicious or criminal attacks comprised 48% of data breaches in 2018. This is the most common area of focus within the cybersecurity industry, typically consisting of advanced persistent threats, zero-day attacks, phishing campaigns, nation-state attacks, malware variants, etc. Other causes of data breaches are human error (28%) and system glitches (25%), which together represent more than half of incidents. The majority of the cybersecurity industry largely focuses on only 48% of the overall problem: malicious or adversarial attacks. Most people find malicious and criminal activities more interesting to read about than the minutiae surrounding simple mistakes that humans and machines make.

In *Pre-Accident Investigations*, Todd Conklin wrote, "Accidents are unexpected combinations of normal variability."

Year over year, for almost a decade, error — the combination of human mistakes and system glitches — has been the leading cause of data breaches. Unless we start thinking differently, it unlikely to change. We seem to be unable to reduce the number of security incidents and breaches despite spending more on building and buying security solutions every year. Perhaps had the cybersecurity industry focused more on human factors and system failures, many of the malicious activities would have failed. However, it's difficult to prove this argument due to the lack of consistently reliable information regarding the relationship between malicious attacks and human factors or system glitches.

At the heart of confronting human error and system glitches is understanding the role of failure in how we build, operate, and improve our systems. The rate of failure, how we fail, and the ability to understand that we failed in the first place are important building blocks to success. In fact, failure itself is an important building block, as it is how we learn and grow as people. It would be naive to think that the way we build systems is perfect, given that "to err is human".

Chaos engineering is entrenched in the concept that failure always exists, and what we do about it is our choice. Popularized by Netflix's Chaos Monkey, chaos engineering is an emerging discipline that seeks to intentionally trigger failures in a controlled fashion to gain confidence that the system will respond to failure in the manner we intend. Recalling that "hope is not a strategy," simply hoping your system is functioning perfectly and that it couldn't possibly fail is not an effective or responsible strategy.

What does it mean to have effective security controls? Even a single, consistent security capability could easily be implemented in a wide variety of diverse scenarios in which failure may arise from many possible sources. For example, a company might adopt a standard firewall technology that is implemented, placed, managed, and configured differently depending on the complexities in the business, web, and data-logic contexts.

In applying chaos engineering to cybersecurity, the focus has been on identifying the relationship between how human factors and unpredictable system glitches directly affect the security of the systems we build and secure. As previously stated, in the field of information security, we typically learn of security or system failures only after a security incident is triggered. We can almost all agree that by the time we hit the incident response phase, it's too late; we must be more proactive in identifying failure in our security and the systems they protect.

Chaos engineering goes beyond traditional failure or resilience testing in that it's not only about validating what we know already. Rather, it also helps us explore the many unpredictable consequences and discover new properties of our inherently chaotic systems. Chaos engineering tests a system's ability to cope with real-world events such as server failures, firewall rule misconfigurations, malformed messages, degradation of key services, etc. using a series of controlled experiments.

Briefly, a chaos experiment — or, for that matter, a security experiment — must follow four steps:

1. Identify and define the system's normal behavior based on measurable output.

2. Develop a hypothesis regarding the normal steady state.

3. Craft an experiment based on your hypothesis and expose it to real-world events.

4. Test the hypothesis thoroughly by comparing the steady state and the results from the experiments.

Just about everyone, whether engineer or not, can relate to the fact that the backup always works and it's the restore you have to worry about. Disaster recovery is a classic example of how the impact of an error can be devastating if backup/restore testing is not performed often enough. The same goes for the rest of the security controls. Don't wait to discover that something is not working; instead, proactively introduce failures into the system to ensure that your security is as effective as you think it is. This is also a great opportunity to discover how observable those modes of failure are in a system. Observability measures how well we can infer the internal state of a system from external observations.

A common way to get started with chaos engineering's security experiments is to start at the beginning of the chaos-engineering value chain: the gameday exercise. The gameday exercise is the core channel used to plan, build, collaborate, execute, and conduct post-mortem analyses for experiments. Gameday exercises typically run between two and four hours and involve a team of engineers who develop, operate, monitor, and/or secure an application. Ideally, they involve members responsible for these tasks working collaboratively. It's easy when getting started in chaos engineering to get caught up in the fun and excitement of designing experiments and tools to break your systems but it's important to understand the other components like measurements and observability in order to be truly successful at the endeavor.

Chaos engineering with security requires a firm understanding of the principles, best practices, and fundamentals documented in the "Principles of Chaos Engineering" produced by the Netflix team that pioneered the space. As a case in point, you would never develop an experiment and run it directly in a production environment without first testing any tools, experiments, etc. in a lower environment such as staging. Start by developing competency and confidence in the methods and tools needed to perform the experiments. Start small and manual. If your hypothesis proves true, automate the experiment.

Scenario: Sock-of-the-Month Club subscription service

Consider a simple web service or web application that takes orders for its Sock-of-the-Month Club service.

This is a critical service for this startup fashion bonanza whose orders come in from the mobile devices of their customers, the web, and via their API from retail stores who also sell their cool kid socks.

This critical service runs in the company's AWS EC2 environment.

The company passed its PCI compliance with flying colors last year and annually performs third-party penetration tests, so they assume that their systems are secure.

This company also prides itself in its DevOps and continuous-delivery practices by deploying sometimes twice in the same day.

After learning about chaos engineering with security-focused experiments, their development teams want to continuously determine how resilient and effective their security systems are to real-world events and to ensure that they are not introducing new problems in the system that their security controls cannot detect.

The team wants to start small by evaluating port security and firewall configurations for their ability to detect, block, and alert on misconfigured changes to the port configurations on their EC2 security groups.

The team's process is:

1. The team begins by briefly summarizing their assumptions about the normal state.

2. The team develops a hypothesis for port security in their EC2 instances.

3. The team selects and configures the YAML file for the "unauthorized port change" experiment.

4. This configuration designates the objects to randomly select for targeting as well as the port ranges and number of ports that should be changed.

5. The team also configures when to run the experiment and shrinks the scope of its

blast radius to ensure minimal business impact.

6. For this first test, the team has chosen to run the experiment in their stage environments and perform a single run of the test.

7. In true gameday style, the team has designated a Master of Disaster to run the experiment during a predefined two-hour window. During that time, the Master of Disaster will execute the experiment on one of the EC2 instance security groups.

8. Once the gameday has finished the team conducts a thorough, blameless post-mortem. The focus of the post-mortem is to compare the results of the experiment against the steady state and the original hypothesis.

The questions would be similar to the following:

- Did the firewall detect the unauthorized port change?
- If the change was detected, was it blocked?
- Did the firewall report log useful information to the log aggregation tool?
- Did security information and event management (SIEM) throw an alert on the unauthorized change?
- If the firewall did not detect the change, did the configuration management tool discover the change?
- Did the configuration management tool report good information to the log aggregation tool?
- Did SIEM finally correlate an alert?
- If SIEM threw an alert, did the security operations center (SOC) get the alert?
- Was the SOC analyst who received the alert able to act on it or was necessary information missing?
- If the SOC alert determined the alert to be credible, was security incident response able to easily conduct triage activities from the data?

The acknowledgement and anticipation of failure in our systems has already begun unravelling the mystery of our assumptions in how our really systems work. Our mission is to take what we have learned and apply it more broadly to begin to truly address security weaknesses, going beyond the reactive processes that currently dominate today's security models.

When searching for the unexpected, the only logical and sane action is to understand that in order to uncover the unexpected, we must objectively become the expected. Let's face it: our systems evolve so rapidly that we have to seek new methods such as chaos engineering in order to better understand how they work, to improve them, and to anticipate the nonlinear nature of their unpredictable outcomes.

KEY TAKEAWAYS

Modern software systems are complex, not merely complicated. This results in never being able to develop a comprehensive and accurate mental model of the system's behavior.

The design and operation of these systems ultimately depend on the ability of those responsible for them to continually update, compare, and contrast (or recalibrate) their mental models.

Incidents are effective directors of attention for organizations; they represent "in the wild" evidence that mental models need recalibration and send specific signals on where in the system to focus this attention. Truly effective post-incident reviews are ideal conditions for this.

RECALIBRATING MENTAL MODELS THROUGH DESIGN OF CHAOS EXPERIMENTS

by **John Allspaw**

In 2017, David Woods, a professor in integrated systems engineering wrote a theorem: As the complexity of a system increases, the accuracy of any single agent's own model of that system decreases rapidly.

As software systems become successful, they necessarily increase in their complexity; they do not tend to get less complex over time. A significant source of this complexity comes from not just needing to manage new forms of faults and disruptions, but also to take advantage of opportunities, whether they be technical (related to the software or hardware) or business-related (such as entering a new market, launching a new feature or product, or partnership, etc.).

Mental models and recalibration

An important finding in Woods' recent research of cognitive work in software engineering and operations environments is that a primary way engineers cope with this complexity is by developing mental models of how their software systems behave in different situations (normal, abnormal, and varying states in between). These mental models are never accurate, comprehensive, or complete, and different individuals have potentially overlapping but different models of the same areas of their systems.

While engineers' mental models of their systems and their behavior may overlap in some ways with their colleagues' models, they are never complete or comprehensively accurate.

These mental models contain:

- understanding of how different (and different types) of components are connected to others;
- expectations of how components behave as they process inputs, produce outputs, and depend on other components in varying ways to do this work;
- sources of concern, such as what components, relationships, or behaviors are more worrisome than others, under what conditions, and what signals to look for that indicate trouble; and
- tips, tricks, or rules of thumb to use when behaviors appear to need correction or modification.

The inaccuracy or incompleteness of these various mental models generally (and perhaps surprisingly) does not cause significant problems. Multiple people (even those on the same team!) can hold these differing and incorrect understandings of how their systems behave for some period of time without much consequence, and without any awareness that these understandings contrast.

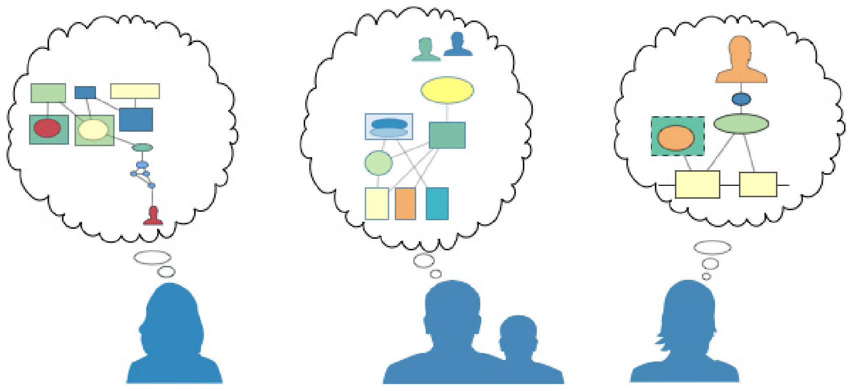
There are many potential reasons for people to have different mental models about how their systems are configured or expected to behave in certain conditions. One reason is simply the churn or turnover of personnel on teams in modern engineering organizations. Growing companies may have newly hired staff with but a surface understanding of the details of the technology and not enough experience to encounter the edges of its behavior. More tenured engineers will have a deeper and broader understanding of the systems simply because of their experience, but unless there is a reason to relay this understanding to new hires (such as an incident), there's little motivation to disseminate this knowledge. A phenomenon known as the "fundamental common-ground breakdown" describes a similar dynamic this way:

- Party A believes that party B possesses some knowledge.
- Party B doesn't have this knowledge, and doesn't know they are supposed to have it.
- Therefore, party B doesn't request the knowledge.
- This lack of a request confirms to party A that party B has the knowledge.

As a result of this miscommunication, parties A and B fail to catch the mismatch in their beliefs about what party B does know and is supposed to know.

Furthermore, party A interprets party B's subsequent statements and comments with the assumption that party B possesses the critical knowledge, thereby constructing a cascading set of incorrect inferences about party B's beliefs and actions.

When system behaviors tread beyond those that are anticipated and designed for, however, they can sometimes produce surprising results. These outcomes elicit



Maybe the process of designing a chaos experiment is just as valuable as the actual performance of the experiment.

reactions from the engineers responsible for them, which range from excited curiosity to outright astonishment and panic that the technology (and therefore, the business) is irreparably damaged.

These events are typically referred to as incidents, and while they can take on different shapes and flavors, they have one thing in common: they can direct attention to where these mental models need recalibration. In one way, incidents can be considered to be a form of untyped pointers in that they direct attention to parts of the technology that require a deeper understanding.

This phenomenon of mental-model recalibration typically takes place in well-guided and facilitated group meetings that have a semi-structured format. In many organizations, this is represented by post-incident review meetings, sometimes called "post-mortems". If conditions exist where multiple people can describe their understanding, expectations, and concerns about the behavior of the involved systems to each other, they can contrast and compare these to reveal the specific ways in which their models do not match. Together with observations made during the incident, this is a key opportunity for recalibration to take place.

If incidents are "in the wild" signals that can direct our attention to those areas of our systems where mental models need recalibration, then chaos experiments reflect an explicit acknowledgment that recalibration is continually needed and valuable.

The where, when, and how of confidence

A chaos experiment is a deliberate activity that involves introducing some form of intentional disruption to a software system (or set of subsystems) to reveal contrast (if any) between a hypothesized normal (or steady-state) behavior and behavior that such disruptions might produce.

The primary goal of these experiments is to build confidence in systems behavior.

Confidence is a complex cognitive construct; in this context, we will take the term to mean the "state of feeling certain about the truth of something".

The chaos experiments will focus on various components, processes, and behaviors that people are to some degree uncertain about. The challenge, of course, is that while the list of options of where, how, and what to experiment on is theoretically infinite, the time and resources to design and carry out a chaos experiment are finite. Therefore, teams have to prioritize what parts of the system to experiment on. Teams typically focus their experiments on areas of their system that a) they are most uncertain about and b) expect to yield the greatest value.

In this way, chaos experiments can be seen as opportunities for mental-model recalibration in the same way that incidents are. Unlike incidents, chaos experiments are planned activities and therefore have some qualities that make them different from incidents.

Before performing a chaos experiment, a hypothesis is made about what constitutes a normal or steady-state operating envelope. This establishes a baseline or control against which to contrast the results of any experiments.

This hypothesis is formed collectively by multiple individuals referencing observations they have made and expectations they have about what levels of variation the given system(s) are likely to experience.

A chaos experiment should have:

- a hypothesis,
- a specific scope, and
- a set of metrics or behaviors to monitor.

We might have similar questions about the experiment's design as we did for establishing the steady state:

- How is the hypothesis for the experiment formed? Is it formed collectively, by multiple individuals?
- How is the scope of the experiment established? Is this scope influenced by past experience with particular outages or is it generated by an absence of confidence about the effects of a future feature, function, integration, etc.?
- How are the metrics or behaviors to monitor during the experiment arrived at? Theoretically, modern applications at scale could have many millions of metrics to give attention to but significantly fewer of those are selected as those to focus on; what makes them more valuable than others?
- How does the overall focus of this experiment compare to others in terms of perceived value?

Perhaps the greatest value of chaos experiments is not the specific results of the experiments themselves. The process of designing these experiments has value in and of itself; it ostensibly sets up conditions for multiple mental models (potentially overlapping yet different, incomplete, and always only partially accurate) to be compared, contrasted, and recalibrated.

Maybe the process of designing a chaos experiment is just as valuable as the actual performance of the experiment.

Conclusions

The design and operation of modern software systems is not done with the expectation that engineers will comprehensively and accurately understand a system's behavior once it is in production. If this were the case, bugs and incidents would be extinct phenomena and there is little evidence that suggests this extinction will happen at any point.

Recent research in the cognitive work of software engineers reveals that instead of developing this elusive omnipotent understanding, real and successful work critically depends on the continual recalibration of mental models that people have of the systems they are responsible for. These mental models are always being updated with new understanding of the system's configuration, dependencies, and behaviors under a huge variety of conditions.

Incidents are valuable opportunities for people to engage in this mental-model recalibration; the systems are (metaphorically) sending signals to people about what areas are in need of recalibration. It's almost as if incidents were entities that say "hey everyone, you should probably come and take a look at this bit over here because it doesn't work the way you think it does."

Likewise, the design and construction of a chaos experiment offer the same opportunity. What behaviors of our system are we less than confident about, collectively? What behaviors are we better at anticipating than others? Seen in this way, chaos experiments are valuable in ways that go beyond the actual results of the experiment.