Lamboozling Attackers

A New Generation of Deception

**SOFTWARE
ENGINEERING
TEAMS CAN
EXPLOIT
ATTACKERS'
HUMAN NATURE
BY BUILDING
DECEPTION
ENVIRONMENTS.**

KELLY SHORTRIDGE AND RYAN PETRICH

Deception is a powerful resilience tactic that provides observability into attack operations, deflects impact from production systems, and advises resilient system design. A lucid understanding of the goals, constraints, and design tradeoffs of deception systems could give leaders and engineers in software development, architecture, and operations a new tactic for building more resilient systems—and for bamboozling attackers.

Unfortunately, innovation in deception has languished for nearly a decade because of its exclusive ownership by information security specialists. Mimicry of individual system components remains the status-quo deception mechanism despite growing stale and unconvincing to attackers, who thrive on interconnections between components and expect to encounter *systems*. Consequently, attackers remain unchallenged and undeterred.

This wasted potential motivated our design of a new generation of deception systems, called deception

environments. These are isolated replica environments containing complete, active systems that exist to attract, mislead, and observe attackers. By harnessing modern infrastructure and systems design expertise, software engineering teams can use deception tactics that are largely inaccessible to security specialists. To help software engineers and architects evaluate deception systems through the lens of systems design, we developed a set of design principles summarized as a pragmatic framework. This framework, called the FIC trilemma, captures the most important dimensions of designing deception systems: fidelity, isolation, and cost.

The goal of this article is to educate software leaders, engineers, and architects on the potential of deception for systems resilience and the practical considerations for building deception environments. By examining the inadequacy and stagnancy of historical deception efforts by the information security community, the article also demonstrates why engineering teams are now poised—with support from advancements in computing—to become significantly more successful owners of deception systems.

DECEPTION: EXPLOITING ATTACKER BRAINS

In the presence of humans (attackers) whose objectives are met by accessing, destabilizing, stealing, or otherwise leveraging other humans' computers without consent, software engineers must understand and anticipate this type of negative shock to the systems they develop and operate. Doing so involves building the capability to collect relevant information about attackers and to implement anticipatory mechanisms that impede the success of their

Software engineers may have beliefs about attacker behavior, but observing and tracing actual attacker behavior will generate the insight necessary to improve system design against unwanted activity.

operations. Deception offers software engineering teams a strategic path to achieve both outcomes on a sustained basis.

Sustaining resilience in any complex system requires the capacity to implement feedback loops and continually learn from them. Deception can support this continuing learning capacity. The value of collecting data about the interaction between attackers and systems, which we refer to as *attack observability*, is generally presumed to be the concern of information security specialists alone. This is a mistake. Attacker effectiveness and systems resilience are antithetical; one inherently erodes the other. Understanding how attackers make decisions allows software engineers to exploit the attackers' brains for improved resilience.

Attack observability

The importance of collecting information on how attackers make decisions in real operations is conceptually similar to the importance of observability and tracing in understanding how a system or application *actually* behaves rather than how it is *believed* to behave. Software engineers can attempt to predict how a system will behave in production, but its actual behavior is quite likely to deviate from expectations. Similarly, software engineers may have beliefs about attacker behavior, but observing and tracing *actual* attacker behavior will generate the insight necessary to improve system design against unwanted activity.

Understanding attacker behavior starts with understanding how humans generally learn and make decisions. Humans learn from both immediate and

repeated interactions with their reality (i.e., experiences). When making decisions, humans supplement preexisting knowledge and beliefs with relevant experience accumulated from prior decisions and their consequences. Taken together, human learning and decision-making are tightly coupled systems. Given that attackers are human beings—and even automated attack programs and platforms are designed by humans—this tight coupling can be leveraged to destabilize attacker cognition.

In any interaction rife with conflict, such as attackers vs. systems operators, information asymmetry leads to core advantages that can tip success toward a particular side. Imperfect information means players may not observe or know all moves made during the game. Incomplete information means players may be unaware of their opponents' characteristics such as priorities, goals, risk tolerance, and resource constraints. If one player has more or better information related to the game than their opponent, this reflects an information asymmetry.

Attackers choose an attack plan based on preexisting beliefs and knowledge learned through experience about operators' current infrastructure and protection of it.¹ Operators choose a defense plan based on preexisting and learned knowledge about attackers' beliefs and methods.

This dynamic presents an opportunity for software engineers to use deception to amplify information asymmetries in their favor.⁷ By manipulating the experiences attackers receive, any knowledge gained from those experiences is unreliable and will poison the attackers' learning process, thereby disrupting their decision-making.

Deception systems allow software engineers to exacerbate information asymmetries in two dimensions: (1) exposing real-world data on attackers' thought processes (increasing the value of information for operators); and (2) manipulating information to disrupt attackers' abilities to learn and make decisions (reducing the value of information for attackers).

The rest of the article will discuss the challenges and potential of deception systems to achieve these goals in real-world contexts.

THE HISTORY OF HONEYPOTS

The art of deception has been constrained by information security's exclusive ownership of it. The prevailing mechanism used to implement deception is through a host set up for the sole purpose of detecting, observing, or misdirecting attack behavior, so that any access or usage indicates suspicious activity. These systems are referred to as *honeypots* in the information security community. It is worth enumerating existing types of honeypots to understand their deficiencies.

Levels of interactivity

Honeypots are typically characterized by whether they involve a low, medium, or high level of interactivity.

LI (low interaction) honeypots are the equivalent of cardboard-cutout decoys; attackers cannot interact with them in any meaningful way. LI honeypots represent simple mimicry of a system's availability and are generally used to detect the prevalence of port scanning and other basic methods attackers use to gather knowledge relevant

Conventional deception approaches are unconvincing to attackers with a modicum of experience.

for gaining access (somewhat like lead generation). They may imitate a specific port or vulnerability and record successful or attempted connections.

MI (medium interaction) honeypots imitate a specific kind of system, such as a mail server, in enough depth to encourage attackers to exploit well-known vulnerabilities, but they lack sufficient depth to imitate full system operation. Upon an exploitation attempt, MI honeypots send an alert or record the attempt and reject it. They are best for studying large-scale exploitation trends of public vulnerabilities or for operating inside of a production network where any access attempt indicates an attack in progress.

HI (high interaction) honeypots are vulnerable copies of services meant to tempt attackers, who can exploit the service, gain access, and interact with the base operating-system components as they normally would. It is uncommon for HI honeypots to include other components that imitate a real system. For the few that do, it is usually a side effect of being built by transplantation. HI honeypots usually send an alert upon detection of an attacker's presence, such as after successful exploitation of the vulnerable software.

Limitations of honeypots

While LI and MI honeypots are generally understood to be ineffectual at deceiving attackers⁹ (and thus can be dismissed as applicable options for real-world deception), the existing corpus of HI honeypots is primitive as well. Conventional deception approaches are unconvincing to attackers with a modicum of experience. Attackers need

only ask simple questions—Does the system feel real? Does it lack activity? Is it old and forgotten?—to dissipate the mirage of HI honeypots.

The limitations of HI honeypots mean that attackers often uncover their deceptive nature by accident. HI honeypots also lack the regular flow of user traffic and associated wear of production systems—a dead giveaway for cautious attackers.

Finally, a fundamental flaw of all honeypots is that they are built and operated by information security specialists, who are typically not involved in software architecture and are largely divorced from software delivery. They may know at a high level how systems are supposed to behave but are often unaware of the complex interactions between components that are pivotal to systems function. As such, this exclusive ownership by security specialists represents a significant downside to current deception efficacy.

MODERN COMPUTING ENABLES NEW DECEPTION

A new generation of deception is not only possible, but also desirable given its strategic potential for systems resilience. The design and ownership of this new category, *deception environments*, reflects a significant departure from the prior generation. Deception environments are sufficiently evolved from honeypots that they represent a new, distinct category.

It is not surprising that attackers find individual honeypot instances unconvincing, given their expertise in attacking *systems* and understanding the interrelation between components to inform their operations. The combination

of new types of computing and ownership by software engineers means that *environments* dedicated to distributed deception can be created that more closely resemble the types of systems attackers expect to encounter.

The goal of traditional honeypots is to determine how frequently attackers are using scanning tools or exploiting known vulnerabilities; tracing the finer nuances of attacker behavior or uncovering their latest methodology is absent from deception projects to date. Deception environments serve as a means to observe and understand attacker behavior throughout all operational stages and as platforms for conducting experiments on attackers capable of evading variegated defensive measures. This concentrates efforts on designing more resilient systems and makes fruitful use of finite engineering attention and resources.

A few dimensions of modern infrastructure are pivotal in nurturing a new deception paradigm with lower costs and more efficacious design.

- ➔ **Cloud computing.** The accessibility of cloud computing enables the ability to provision fully isolated infrastructure with little expense.
- ➔ **Deployment automation.** Full systems deployment automation and the practice of defining infrastructure declaratively, commonly referred to as IaC (infrastructure as code), decreases operational overhead in deploying and maintaining shadow copies or variants of infrastructure.
- ➔ **Virtualization advancements.** The widespread availability of nested virtualization and mature, hardened virtualization technologies inspires confidence that attackers are isolated from production, makes it possible

Starting with the design of a genuine production system provides an inherent level of realism to bamboozle attackers and glean insights pertinent to refining resilience in the real system.

to observe them in more detail, and extracts extra density out of computing resources.

➔ SDN (software-defined network) proliferation. With the ability to define networks programmatically, isolated network topology dedicated to attackers can be created without incurring additional cost.

NEW OWNERSHIP

This is another crucial catalyst for this latest generation of deception. Ownership based on systems design expertise, rather than security expertise, creates the dynamism necessary for deception systems to succeed against similarly dynamic opponents.

Software engineering teams are already executing the necessary practices. Software operators can repurpose their unique system deployment templates for building production environments and variants (such as staging environments) toward building powerful deception systems. They can then derive attack data that is distinctly applicable to their environments and cannot be garnered elsewhere. As a result, software engineers are more qualified for the endeavor than security teams and can gain a highly effective observability tool by deploying deception environments.

DESIGNING DECEPTION ENVIRONMENTS

The design philosophy underlying deception environments is grounded in repurposing the design, assets, and deployment templates of a real system instead of building a separate design for deception (as is the status quo). Deception becomes a new environment generated at the

end of software delivery pipelines after development, staging, pre-production, and production. From this foundation, attacker skepticism can be preempted by designing a deception environment that feels “lived in” through tactics such as replaying traffic and other methods of simulating system activity.

Starting with the design of a genuine production system

provides an inherent level of realism to bamboozle attackers and glean insights pertinent to refining resilience in the real system. Since every system has different resilience concerns, this also offers an opportune and safe test of how tactics perform against real attackers in a pseudo-real environment.

The FIC Trilemma

*Sweet Spots for
Deception
Environments*

FIC—The most important dimensions of designing deception systems: Fidelity, Isolation, and Cost

Replicomb—A full replica of a production host with imitated load and a purposefully vulnerable component.

Honeyhive—A full, scaled-down replica of an entire production environment with activity flowing through a web of replicomb hosts.

The FIC trilemma

Traditional honeypot design has focused on initial access, and success is determined by how well a honeypot can mimic the outer shape of a system. This framing limits the ability to evaluate approaches beyond the rudimentary ones seen to date.

The new model proposed here evaluates deception systems along three axes: fidelity, isolation, and cost (figure 1), representing a trilemma: The three properties are generally in conflict and therefore cannot

be fully achieved simultaneously. Understanding the FIC trilemma—and the tradeoffs between each of its axes—is vital for designing successful deception environments.

1

FIGURE 1: THE FIC TRILEMMA FOR DECEPTION SYSTEMS

Fidelity

Fidelity refers to the deception system's credibility to attackers and its ability to support attack observability. A credible deception system is effective at deceiving attackers into thinking the system is real; it avoids falling into the "uncanny valley." Attackers often interrogate compromised systems to unmask mirages and avoid revealing their methods. Attackers expect certain basic traits in systems, such as running a service, receiving production-like traffic, connecting to the wider Internet, coordinating with other services over a local network, being orchestrated and monitored by another system, and not having traces of debuggers or other instrumentation tools.

A highly credible deception system will provide sufficient depth to stimulate extended attacker activity, luring even cautious attackers into moving between hosts and revealing their methods across the attack delivery life cycle. This

2

begets a detailed and high-quality record of behavior for engineers to gain an accurate understanding of attacker decision-making. Greater accuracy and depth in extracting and recording activity informs better system design that makes future iterations more resilient to attack.

Isolation

Isolation refers to the degree to which a deception system is *isolated* from the real environment or data, and is the second axis of the FIC trilemma. Operators are loath to jeopardize the availability of the real system or data privacy in order to learn about attacker behavior. A secondary element is the ability to keep attackers isolated from each other. This permits study of each attacker’s behavior independently.

FIGURE 2: **EXAMPLE DECEPTION SYSTEMS MAPPED TO THE FIC TRILEMMA**

Cost

Cost refers to the computing infrastructure and operational overheads required to deploy and maintain deception systems. As computing expenses continue to plummet, cost shifts to operational burden—which should not be underestimated. Expensive deception systems are unlikely to be fully deployed or maintained and will thereby fail to serve their purpose.

Mapping different types of deception systems to points around the trilemma elucidates the value of this model. As a starting point, let us consider which types of systems reflect extreme realizations of each of these axes: perfect fidelity, total isolation, and maximum cost (figure 2).

Real production systems reside at the intersection of *perfect fidelity, little cost, and no isolation*. These systems are likely to encounter attackers and be monitored by operators, because production is where organizations realize value from software-development activity (that is, it makes the money, which attackers and organizations similarly appreciate).

In contrast, LI honeypots reside at the intersection of *no fidelity, little cost, and perfect isolation*. They gather limited information about attackers and present them with a transparent trick; however, they are easy to deploy, can detect broad attack trends, and offer a risk-free incident impact.

At the intersection of *perfect fidelity, full isolation, and maximum cost* resides a hypothetical datacenter dedicated to deception. As an example, imagine a complete copy of a production data center with identical monitoring and

3

FIGURE 3: MI & HI HONEYPOTS ON THE TRILEMMA

maintenance, as well as perfect simulation of real traffic using an army of distributed clients. This obviously bears an exorbitant cost in terms of design and operation but provides high fidelity and full isolation.

To explore the FIC trilemma further, figure 3 evaluates the aforementioned approaches from the information security community.

MI honeypots offer minimal supplemental fidelity and cost about the same to deploy as LI honeypots; hence, they occupy a space close to LI honeypots. HI honeypots represent a minor increase in fidelity, at some cost, but are unable to fool most attackers. Even when simulated load is applied to boost authenticity, HI honeypots still suffer from the limitations of imitative design rather than sharing lineage with real existing systems.

4

FIGURE 4: THE FIC “SWEET SPOT”: HONEYHIVES AND REPLICOMBS

Sweet spots for deception environments

The model for deception environments supports solutions in previously unexplored spaces in the trilemma (figure 4). The following two trilemma “sweet spots” provide mechanisms for uncovering a richer and higher volume of attacker behavior for advanced observability.

Systems in the first sweet spot, called *replicombs*, are downgraded replicas of production hosts that run with the same set of monitoring, orchestration, and supporting services deployed in real production environments. The replica is fed with simulated or replayed load from actual systems. A full replica host with a production-like load creates a deception system that, to an attacker, appears indistinguishable from a real host (figure 5).

Modern deployment practices such as IaC make crafting

5

FIGURE 5: AN EXAMPLE REPLICOMB DEPLOYMENT

downgraded replicas easier, and the plummeting cost of cloud computing makes deployment of sizable systems inexpensive. While still higher cost than a honeypot, a replicomb offers palpable enhancements: It features impressive fidelity and supports inspecting an expansive range of attacker behavior beyond initial access. Because of this, the replicomb occupies a space on the trilemma closer to the full data-center replica. This is a sweet spot because it should, if implemented correctly, appear to be a real individual production host to even a cautious, skeptical attacker.

Systems in the second proposed sweet spot, called

6

FIGURE 6: **EXAMPLE HONEYHIVE BASED ON A PRODUCTION ENVIRONMENT**

Building replicombs and honeyhives is no more difficult for software organizations than setting up a new variant of an existing environment tier through IaC declarations.

honeyhives, extend the replicomb approach with a full network of like-production hosts to observe how attackers move from their initial point of access onto adjacent hosts and services. Complete but scaled-down copies of an entire environment are deployed as a honeyhive with simulated, replayed, or mirrored activity flowing through the entire system. Therefore, a honeyhive yields a thoroughly lifelike environment for observing and conducting experiments on attackers, even if their behavior spans multiple systems (figure 6).

The honeyhive environment may sound similar to a preproduction or staging environment—and it is. Modern IaC practices and inexpensive full isolation via cloud computing allow for a deception system such as a honeyhive to be deployed at a more reasonable cost than previously feasible. The honeyhive occupies a space on the trilemma nearest to the full data-center replica, offering profound fidelity to attackers and in the intelligence gathered from it. With a honeyhive, behavior can be observed through more stages of an attacker's operation.

A replicomb is the starting point for a honeyhive, but outsized fidelity is unlocked by deploying the rest of the environment. A replicomb is effectively a copy of a service, so it requires simulated load to appear real. A honeyhive, in contrast, needs only simulated load applied to any points that would naturally interact with users; only one “true” replicomb is required as the initial entry point. The other replicomb hosts receive traffic from their peers just as they would in a production environment, so the honeyhive simply needs some external traffic to engender realistic flows.

Real-world implementation

Building replicombs and honeyhives is no more difficult for software organizations than setting up a new variant of an existing environment tier through IaC declarations. Deploying a replicomb is similar to a canary release of the chosen service, and deploying a honeyhive is similar to a soak or load test environment.

With this said, safely building deception environments requires careful attention to details beyond the usual concerns when creating a new environment.

Isolation boundaries

Where does the isolation boundary exist between the deception environment and any other environments that process user data or must remain available? How permeable is this boundary? Purposefully deploying vulnerable instances of a service without properly isolating them from user traffic is dangerous.

Similarly, some organizations run multiple environments within the same network, allowing direct communication between them. Deception environments should not follow this pattern but instead should be deployed with little to no ability to communicate with environments dedicated to other purposes, especially those handling critical production traffic. Virtualization techniques, SDNs and cloud computing can be used to create fully isolated networks for deception environments.

Discoverability

Attackers must be able to discover deception environments to collect real data on their attack operations. Placing a deception environment on a public IP address without any association to your organization attracts only the attackers

searching across the Internet for open holes to poke. Placing a deception system inside a production environment as a discoverable host captures behavior only after attackers are already inside and seeking additional hosts.

One technique to trap attackers seeking access to a specific organization and then to observe their behavior across all stages of their operation is *honeypatching*. This technique directs traffic intended to exploit a known and already-patched vulnerability in production to the deception environment's unpatched service by leveraging the configurability of modern firewalls and load balancers (figure 7).

Tamper-free observation

A salient benefit of funneling attacks into a deception environment is the ability to trace the attackers' actions without any risk to actual service availability. This tracing should be invisible to attackers and resistant to tampering by them. Once attackers gain access to a deception environment, they can potentially manipulate any monitoring or observability tools running within it. Accordingly, the best way to ensure tamper-free observation of attacker behavior is to deploy these tools outside the environment but peering inward.

Network behavior can be observed by capturing all packets entering, leaving, and moving between hosts by using a CSP's (cloud service provider) native features for archiving traffic within a virtual network or by using the packet-capture facilities of virtualization systems. Host behavior can be observed by taking regular snapshots of memory and disk to view the deception system's exact state at a given time.

7

To improve resilience against attackers evading any monitoring of their actions, packet capture and periodic snapshots can be supplemented with standard observability tools. Essential events such as process launch and file activity collected inside the environment can also enrich the trace of attacker activity.

Accidental data exposure

Organizations may inadvertently accept liability by purposefully exposing user data to attackers in the

FIGURE 7: AN EXAMPLE REPLICOMB ENVIRONMENT WITH HONEYPATCHING

deception environment. This problem can be mitigated by anonymizing or scrambling traffic before it is replayed into the deception environment.

Generating synthetic data sets—those that mimic production data but do not include any real user data—is an existing approach for populating preproduction, staging, and other test environments while still complying with privacy regulations (such as HIPAA). Organizations in less privacy-conscious industries may need to adopt a similar approach for deception environments to avoid unwanted liability.

Ownership

The conventional view of deception systems is that they reside in the domain of information security. With modern advancements in deployment tooling and methodology, creating variants of production systems is a straightforward exercise—and deception environments are simply another variant of the system. Software engineers can consequently deploy and maintain effective deceptions in a more straightforward, predictable, low-effort, automated, consistent, and understandable way.

Security expertise is not a prerequisite for developing and operating deception environments. In fact, it often constrains and impairs judgment of strategic options. Engineering teams naturally gravitate toward improvements to design or workflows instead of relying on status-quo “best” practices that are seldom informed by systems thinking. Attackers think in systems; they develop systems to achieve their objectives and incorporate feedback during operations. By treating an attacker as a kindred engineer with the exact opposite goals to yours,

your mindset will be amply authentic and constructive to wield and benefit from deception environments.

HARVESTING DECEPTION'S FULL POTENTIAL

Here are a few powerful use cases to harvest the potential of a deception environment after it is deployed.

Resilient system design

The data generated from replicombs and honeyhives can inform more resilient system design. Minimizing the time to detect and respond to destructive activity that precipitates service downtime is correlated with organizational performance²—and attacks are firmly in the category of such ruinous activity. A dedicated sandbox for exploring how attacks impact systems is an invaluable tool for anticipating how production systems will behave when failure occurs and preempting it through design improvements.

Attackers will interact with monitoring, logging, alerting, failover, and service components in ways that stress their overall reliability. A resilient system must be aware of and recover from failures in any of these components to preserve availability. Deception environments can corroborate any measures implemented to support visibility into and recovery from component failure.

Deception environments can also expose opportunities for architectural improvement in operability and simplicity. For example, if spawning remote interactive shells (so attackers can write their tools to disk) is a consistent attacker behavior seen across deception environments, this evidence could motivate a design specification of host

Attack observability enables pragmatic threat modeling during design and planning without security expertise as a prerequisite.

immutability to eliminate this option for attackers.⁵

Importantly, this aligns with a future in which product and engineering teams are accountable for the resilience of the systems they develop and operate (including resilience to attacks).⁴ In the spirit of SCE (security chaos engineering), software engineers, architects, site reliability engineers, and other stakeholders can leverage a feedback loop fueled by real-world evidence—such as that produced by replicombs and honeypots—to inform improvements across the software-delivery life cycle.

Attacker tracing

Deception environments equip software engineers, architects, and other systems practitioners to “trace” the actions of attackers. Attack observability enables pragmatic threat modeling during design and planning without security expertise as a prerequisite. Since attacker behavior is traced in detail on a system with the same shape as a real system, the resulting insight is perfect for modeling likely decision patterns via frameworks such as attack trees.⁶ It can inform revisions to systems design, adjustments to monitoring and observability, or revised resilience measures.

Attack trees are a form of decision tree that graphs decision flows—how attackers will take one path or another in a system to reach their objectives. While a safe default assumption is that attackers will pursue the lower-cost decision path, the in-the-wild evidence collected from deception environments can validate or update existing hypotheses about how attackers learn and make decisions in specific systems. For example, attacker tracing can

establish which tactics (or combinations of them) nudge attackers toward certain choices.

This elucidation of behavioral patterns across the attack life cycle can be visualized as different branches on the attack tree and aid in prioritizing system design changes. (An example tool for visualizing security decision trees is Deciduous, an open-source web app created by the authors and available at <https://www.deciduous.app/>). It can also excavate the hidden flows within systems that are ordinarily discovered only upon failure and system instability. Attackers are adept at ferreting out unaccounted flows to achieve their objectives, so tracing their traversal paints a more precise picture of the system.

Attacker tracing can also inform experimentation; each branch on the attack tree represents a chain of hypotheses that elicit specific experiments. Attacker tracing could also extract thorough characterizations of attackers—their objectives, learning ability, level of risk aversion, degree of skepticism, and other behavioral factors. Such characterization could galvanize personalized deception by combining conditional logic and a “terraforming” approach.

Experimentation platform

A lifelike environment indistinguishable from a production environment maximizes success in deceiving attackers across all levels of capability. A deception environment can therefore serve as a platform for conducting experiments to test hypotheses on how attackers will behave in various circumstances. Attacker tracing—especially when accompanied by attack trees—can directly inform hypotheses for experiments.

Solution efficacy

Experimentation can test the efficacy of monitoring or resilience measures and whether they can be subverted without the operator's knowledge. For example, deception environments can reveal how attackers might respond to architecture redesigns or to substitutions of infrastructure components (swapping for an equivalent capability, as discussed further in the next section, "Fidelity Thresholds").

Similar to validating system performance with load or soak tests, it is valuable to validate system resilience under various failure scenarios (including exposure to attackers). SCE rests on this foundation of experimentation; fault injection generates evidence that builds knowledge of systems-level dynamics—informing continuous improvement of systems resilience and creating muscle memory for incident response.⁴ Through this lens, deception environments become an experimentation tool in the SCE arsenal.

Fidelity thresholds

Fidelity degradation experiments can divulge how attackers react to environments with varying levels of fidelity—uncovering the point at which a system begins to look like an "uncanny valley" to different attackers. Removing components one by one (similar to A/B testing) can surface which aspects of the environment attackers use to evaluate realism.

For example, attackers may treat systems running without monitoring tools as insufficiently important to bother ransoming, since system criticality influences the victim's willingness to pay. Conducting experiments by alternatively disabling and enabling monitoring and logging

subsystems can unveil to what extent attackers will flee from unmonitored systems.

For well-resourced attackers capable of gaining access to both the honeyhive and the production environment, swapping standard components for substitutes can disrupt their attack plans. These substitutes expose the same interface and perform the same function (similar to Coke vs. Pepsi), but the difference in brand name introduces unreliability into the attacker's operational knowledge. Swapping components and testing system behavior under simulated load is common in engineering disciplines and useful for evaluating many categories of hypotheses (such as whether a new component performs better or is easier to make operational).

Access difficulty

The difficulty involved in accessing the deception environment can be tuned to study different types of attackers and their perceptions of fidelity. Since attackers expect certain victims to have a basic level of software-patching hygiene, advertising trivially exploitable vulnerabilities can degrade a deception's fidelity. By selecting which vulnerabilities to telegraph as patched or unpatched (i.e., honeypatches), the accessibility of initial entry points can be adjusted to meet attacker expectations.

Honeytokens for flavor

Augmenting honeyhives with other deception techniques can measure the efficacy of those techniques and trace an attacker's progress in more detail. For example, deploying cloud honeytokens throughout the environment can warn operators when attackers gain access to various

systems and to what extent they attempt to access cloud resources. (An example of cloud honeytokens is the AWS [Amazon Web Services] key canarytoken by Thinkst, available for free at <https://canarytokens.org/generate>.)

FUTURE OPPORTUNITIES

The potential use cases for deception environments described thus far can be realized with contemporary practices and tools. From here, modest extensions to the underlying technology can serve as general-purpose tools that can even benefit disciplines beyond deception.

Just-in-time terraforming

Modern virtualization could support just-in-time creation of isolated deception VMs (virtual machines) via copy-on-write or page deduplication. (This was proven possible in 2005 but never adopted elsewhere.)⁸ This process is similar to how operating systems employ lazy copying of memory pages after processes fork. This could reduce costs by sharing resources among deception environments and creating them only when an attacker first gains access.

A “systems terraforming” approach could even render the illusion of an entire network of hosts that are reified only when an attacker attempts to connect to them. Cloud networking and hypervisor layers already cooperate to route network traffic within a VPC (virtual private cloud) to the physical hardware associated with the intended instance. For lower overhead of unused infrastructure, these layers could instead treat VM instances similarly to serverless functions: powering on once they receive traffic

but otherwise remaining suspended, hibernated, powered off, or paged out to disk.

Clever improvements to the network and hypervisor layers could facilitate this freezing of idle services, hosts, and infrastructure. These assets would be unfrozen upon first contact over the network and have their execution fast-forwarded to the point of interactivity. Once instances finish processing incoming traffic and return to idle states, they would be put back into deep freezes to reduce resource usage. This approach could reduce the cost of blue-green deployments, preproduction environments, and other scenarios where infrastructure mostly idles.

Instance emulation

Advancements in virtualization technology that would better emulate the proprietary hardware and local instance metadata endpoints of AWS and GCP (Google Cloud Platform) would allow creation of deception environments that appear to be real Amazon EC2 (Elastic Compute Cloud) and GCP instances. Full emulation of CSP APIs could lead to benefits such as offline testing of cloud environments, higher-density testing of an entire fleet on a single host, and fully isolated honeypots that live on a single machine.

Scalable honeypatching

Networking technologies such as CDNs (content delivery networks), routers, load balancers, service meshes, or web application firewalls can be reconfigured to support low-effort honeypatching at scale. Rather than blocking exploitation attempts, attackers could be trivially

redirected to a deception environment (with the sole new overhead of specifying where to direct suspicious traffic). Additionally, vulnerability signatures are currently treated as post-hoc security mechanisms but could be distributed alongside software updates instead for straightforward and swift implementation. If supplied to the aforementioned networking technologies, signatures could permit pattern matching in data or protocol streams.

Anonymization via mirroring

Traffic-mirroring technologies, such as those integrated into service meshes and VPCs, could be extended to include data anonymization features that operate at the application protocol layer. Current anonymization techniques operate at the packet layer,³ which is insufficient to anonymize user data being mirrored into a deception environment. Offline data anonymization techniques such as privacy-preserving encryption and pseudonymization could be integrated with traffic-mirroring systems to uphold user privacy and meet compliance requirements while replicating a completely natural traffic pattern in a deception environment.

Hypervisor-based observability

Tracing and observability are core user requirements of modern server operating systems but are often simple to subvert. Such tools commonly execute at the same level of privilege as the workloads they monitor; everything is root. Ideally, installing and running a userland agent should not be necessary to get basic metrics and telemetry out of the kernel for systems monitoring.

Toward this goal, operating systems could expose core system events such as process and file operations directly to hypervisors over a common protocol. This would surface visibility that could not be subverted by an attacker operating inside the VM and would prevent observability outages provoked by resource exhaustion (as can often occur on overloaded hosts).

Burstable memory usage

The infrastructure cost of deception environments could be further reduced if CSPs supported traditional virtualization features such as ballooning and compressed or swapped memory. CSPs could then offer burstable performance instances featuring the ability to burst memory usage to a higher level when required (while offering lower baseline performance and therefore lower cost). This is similar to AWS's existing credits-based system for workloads that require infrequent bursts of CPU usage or GCP's static customization of memory assignment.

With advanced hypervisor extensions, CSPs could migrate VMs across physical instances when their activity bursts and they require more resources. Instead of stopping idle instances on a shared physical machine when one instance bursts, idle instances could be temporarily migrated to another host machine or swapped to disk. This approach is possible with current technology but is not yet implemented by CSPs.

Per-account billing limits

To restrict the amount of money attackers can spend on your behalf, per-account billing limits are stronger than

billing alerts. Unfortunately, CSPs do not provide tools to limit spending by account or project and alert only when unusual activity occurs or thresholds are exceeded. These tools are adequate when availability eclipses cost concerns but are incapable of enforcing a true backstop for resource consumption. CSPs have effective tools for isolating every resource except for their customers' wallets; customers could ask them to add this capability.

PARTING THOUGHTS

Imagine a world in which developers and operators of systems exploit attackers as much as attackers exploit defenders. By leveraging system-design knowledge and modern computing to deploy deception environments, software engineering teams can successfully bamboozle attackers for fun and profit while deepening systems resilience.

References

1. Alderson, D. L., Brown, G. G., Carlyle, W. M., Wood, R. K. 2011. Solving defender-attacker-defender models for infrastructure defense. Center for Infrastructure Defense, Operations Research Department, Naval Postgraduate School, Monterey, CA; <https://calhoun.nps.edu/handle/10945/36936>.
2. Forsgren, N., Humble, J., Kim, G. 2018. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution.
3. Hyojoon, K., Chen, X., Brassil, J., Rexford, J. 2021. Experience-driven research on programmable networks.

- ACM SIGCOMM Computer Communication Review* 51(1), 10-17; <https://dl.acm.org/doi/10.1145/3457175.3457178>.
4. Rinehart, A., Shortridge, K. 2020. *Security Chaos Engineering*. O'Reilly Media.
 5. Shortridge, K., Forsgren, N. 2019. Controlled chaos: the inevitable marriage of DevOps & security. Presentation at Black Hat USA; https://www.researchgate.net/publication/336613723_Controlled_Chaos_The_Inevitable_Marriage_of_DevOps_Security.
 6. Shortridge, K. 2021. The scientific method: security chaos experimentation & attacker math. Presentation at RSA Conference; <https://www.rsaconference.com/Library/presentation/USA/2021/The%20Scientific%20Method%20Security%20Chaos%20Experimentation%20%20Attacker%20Math>.
 7. Veksler, V. D., Buchler, N., LaFleur, C. G., Yu, M. S., Lebiere, C., Gonzalez, C. 2020. Cognitive models in cybersecurity: learning from expert analysts and predicting attacker behavior. *Frontiers in Psychology* 11 (1049); <https://www.frontiersin.org/articles/10.3389/fpsyg.2020.01049/full>.
 8. Vrabie, M., Ma, J., Chen, J., Moore, D., Vandekieft, E., Snoeren, A. C., Voelker, G. M., Savage, S. 2005. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. *ACM SIGOPS Operating Systems Review* 39 (5), 148-162; <https://dl.acm.org/doi/10.1145/1095809.1095825>.
 9. Zhang, L., Thing, V. L. L. 2021. Three decades of deception techniques in active cyber defense - retrospect and outlook. *Computers & Security* 106 (102288); <https://www.sciencedirect.com/science/article/pii/S0167404821001127>.

Kelly Shortridge is a senior principal in product technology at Fastly. Kelly is co-author with Aaron Rinehart of *Security Chaos Engineering* (O'Reilly Media) and is an expert in resilience-based strategies for systems defense. Their research on applying behavioral economics and DevOps principles to information security has been featured in top industry publications and is used to guide modernization of information security strategy globally.

Ryan Petrich was most recently chief technology officer at Capsule8. Their current research focuses on using subsystems in unexpected ways for optimum performance and subterfuge. Their work spans developing popular and foundational jailbreak tweaks, architecting resilient distributed systems, and experimenting with compilers, eventual consistency, and frustrating instruction sets.

Copyright © 2021 held by owner/author. Publication rights licensed to ACM.